Anton: Answer Set Programming in the Service of Music

Georg Boenn
Cardiff School of Creative & Cultural Industries
University of Glamorgan
Cardiff, CF24 2FN, UK
gboenn@glam.ac.uk

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

Abstract

With the increasing efficiency of answer set solvers and a better understanding of program design, answer set programming has reached a stage where it can be more successfully applied in a wider range of applications and where it attracts attention from researchers in other disciplines. One of these domains is music synthesis. In this paper we approach the automation and analysis of composition of music as a knowledge representation and advanced reasoning task. Doing so, it is possible to capture the underlying rules of melody and harmony by a very small, simple and elegant set of logic rules that can be interpreted under the answer set semantics. Our system, ANTON is the first algorithmic composer to combine both harmonic and melodic composition. In addition to describing the composition system thus created we consider the advantages of constructing an algorithmic composer this way, and also the limitations of current solvers.

Introduction

Originally computers were seen as machines to assist in numerical calculations, and it was soon realised that they could do other things, starting with commerce, but extending to symbolic operations and eventually to near-universal use in all technical areas. More recently the application of computers to artistic activity has become a subject of interest.

In this paper we report on the use of declarative logic programming as a significant component of an artistic endeavour, the composition of music. We show that it is possible to use Answer Set Programming (ASP) to create *ab initio* short musical pieces that are both melodic and harmonic. After a description of the computational basis we describe the musical context of this work, and why it is neither a trivial task, nor a tractable one. Our system, ANTON, named in honour of our favourite composer of the second Viennese School, is presented as both a design and as a practical working system. We report on our experience in using ASP for this system, and indicate a number of potentially exciting directions in which this system could develop, both musically and computationally.

Answer Set Programming

Due to space constraints, only a brief overview of the answer set semantics and Answer Set Programming (ASP) is given here. The interested reader is referred to (Baral 2003) for a more in-depth coverage of the definitions and ideas presented in this section.

The *answer set semantics* is a model based semantics for normal logic programs.

Following the notation of (Baral 2003), we refer to the language over which the answer set semantics is defined as $AnsProlog$.

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* $\mathrm{not}\ a$. If a is true then $\mathrm{not}\ a$ is false and vice versa. Atoms and literals are used to create rules of the form:

$$a \leftarrow B, \mathrm{not}\ C.$$

where a is an atom, B and C are sets of atoms. Intuitively, this means "if all element of B are known/true and no element of C is known/true, then a is known/true". The set of conditions of a rule (on the right hand side of the arrow) are known as the *body*, written as $B(r)$, and the atom that is the consequence of the rule is referenced as the *head* of the rule, written $H(r)$. The body is split further in two sets of atoms, $B^+(r)$ and $B^-(r)$ depending on whether the atom appears positively or negatively. Rules with empty bodies are called *facts*; their head should always be true.
A *program* in $AnsProlog$ is a finite set of rules.

If a program Π contains no negated atoms ($\forall r \in \Pi \,.\, B^-(r) = \emptyset$) its semantics is unambiguous and can easily be computed as the fixed point of the T_p(the immediate consequence) operator. Starting from the empty set, we check in each iteration which rule bodies are true. The heads of those rules are added to the set for the next iteration. This is a monotonic process, so we obtain a unique fixpoint, denoted $T_p^\infty(\emptyset)$. This fixpoint is called the *answer set*.

For example, given the following program:

$$\begin{aligned} a &\leftarrow b, c. \\ b &\leftarrow c. \\ c &\leftarrow . \\ d &\leftarrow e. \\ e &\leftarrow d. \end{aligned}$$

the unique answer set is $\{a, b, c\}$, as $T_p(\emptyset) = \{c\}$, $T_p(\{c\}) = \{b, c\}$, $T_p(\{b, c\}) = \{a, b, c\}$ and $T_p(\{a, b, c\}) = \{a, b, c\}$. Note that d and e are not included in the model as their is no way of concluding e without knowing d and vice versa. This is different to the classical interpretation of this program (via Clark's completion) which would have two models, one of which would contain d and e.

The natural mechanism for computing negation in logic programs in *negation as failure*, which tends to be characterised as epistemic negation ("we do not know this is true"), rather than classical negation ("we know that this is not true"). This correspondence is motivated by the intuition that we should only claim to know things that can be proven; thus anything that can not be proven is not known. To extend the semantics to support this type of negation, the *Gelfond-Lifschitz reduct* is used. This takes a set of proposed atoms and gives a reduced, positive program by removing any rule which depends on the negation of any atom in the set and dropping all other negative dependencies.

Definition 1 *Given an $AnsProlog$ program Π and a set of atoms A, the Gelfond-Lifschitz transform of Π with respect to A is the following set of rules:*

$$\Pi^A = \{H(r) \leftarrow B^+(r) | r \in \Pi, B^-(r) \cap A = \emptyset\}$$

This allows us to extend the concept of answer sets to programs with negation. Intuitively, these are sets of possible beliefs about the world which are consistent with all of the rules and have acyclic support for every atom that is known, and thus in the set.

Definition 2 *Given an $AnsProlog$ program Π, A is an answer set of Π $\iff$ A is the unique answer set of Π^A.*

For example, the following program has two answer sets:

$$\begin{aligned} a &\leftarrow \text{not } b. \\ b &\leftarrow \text{not } a. \\ c &\leftarrow \text{not } d. \\ d &\leftarrow b. \\ d &\leftarrow e, \text{not } a, \text{not } c. \\ e &\leftarrow d, \text{not } a. \end{aligned}$$

$\{a, c\}$ and $\{b, d, e\}$. Computing the reduct with respect to $\{a, c\}$ gives:

$$\begin{aligned} a &\leftarrow . \\ c &\leftarrow . \\ d &\leftarrow b. \end{aligned}$$

which results in $T_p^\infty(\emptyset) = \{a, c\}$.

A given program will have zero or more answer sets. With $AnsProlog$ we can represent and reason about NP-complete problems in such a way that the answer sets of the program correspond to the solutions of the problem.

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 choice; if the conditions in the body are met then a number of atoms in the head may (a non-deterministic choice) be contained in 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*).

Answer set programming (ASP) is a programming paradigm in which a problem is *represented* as an $AnsProlog$ program in such a way that the answer sets can be *interpreted* to give the solutions. A reasoning engine is then used to produce the answer sets of the program. Typically these are composed of two components, a *grounder* which removes the variables from the program by instantiation and an *answer set solver* which compute answer sets of the propositional program. GRINGO(Gebser, Schaub, and Thiele 2007) and LPARSE(Syrjänen 2000) are the grounders most commonly used and CLASP(Gebser et al. 2007), SMODELS(Syrjänen and Niemelä 2001), CMODELS(Lierler and Maratea 2004) and DLV(Eiter et al. 1998) represent the state of the art of solver development.

ASP has been used to tackle a variety of problems, including: planning and diagnosis (Eiter et al. 2002; Lifschitz 2002; Nogueira et al. 2001), modelling and rescheduling of the propulsion system of the NASA Space Shuttle (Nogueira et al. 2001), multi-agent systems (Baral and Gelfond 2000; Buccafurri and Caminiti 2005; Cliffe, De Vos, and Padget 2006), Semantic Web and web-related technologies (Polleres 2005; Ruffolo et al. 2005), superoptimisation (Brain et al. 2006), reasoning about biological networks (Grell, Schaub, and Selbig 2006), voting theory (Konczak 2006) and investigating the evolution of language (Erdem et al. 2003).

The Musical Background

Music is a world-wide phenomenon across all cultures. The details of what constitutes music may vary from nation to nation, but it is clear that music is an important component of being human.

In this paper we are concentrating on western traditional musics, but as we will consider later in the section on future music research, much of the technology can be translated to other traditions.

The particular area of interest here is composition; that is creating new musical pieces.

Creating melodies, that is sequences of pitched sounds, is not as easy as it looks (sounds). We have cultural preferences for certain sequences of notes and preferences dictated by the biology of how we hear. This may be viewed as an artistic (and hence not scientific) issue, but most of us would be quick to challenge the musicality of a composition created purely by random whim. Students are taught rules of thumb to ensure that their works do not run counter to cultural norms and also fit the algorithmically definable rules of pleasing harmony when sounds are played together.

"Western tonal" simply refers to what most people in the West think of as "classical music", the congenial Bach through Brahms music which feels comfortable to the modern western ear because of its adherence to familiar rules. Students of composition in conservatoires are taught to write this sort of music as basic training. They learn to write melodies and to harmonise given melodies in a number of sub-versions. If we concentrate on early music then the scheme often called "Palestrina Rules" is an obvious example for the basis of this work. Similarly, harmonising Bach chorales is a common student exercise, and has been the subject of many computational investigations using a variety of methods.

In this paper, we take the somewhat arid technical rules and embed them within a modern computational system, which enables us to contemplate many original ways of exploiting the fact that they are simultaneously available; the rules themselves can be explored, extended and refined, or student exercises can be evaluated to ensure that they are indeed "valid". We will be able to complete partial systems, such as producing a melody consonant with a given harmony structure, as well as, more adventurously, to create new melodies.

For this paper we have opted to work with a sub-type of the Palestrina Rules called Renaissance Counterpoint. This style was used by composers like Josquin, Dufay or Palestrina and is very distinct from the Baroque Counterpoint used by composers like Bach, Haendel.

We have used the teaching at one conservatoire in Köln to provide the basic rules, which were then refined in line with the general style taught. The point about generating melodies is that the "tune" must be capable of being accompanied by one or more other lines of notes, to create a harmonious whole. The requirement for the tune to be capable of harmonisation is a constraint that turns a simple sequence (a *monody*) to a *melody*.

Our experience with this work is to realise how many acceptable melodies can be created with only a few rules, and as we add rules, how much better the musical results are. This concept is developed further in the section on ANTON.

In this particular style of music complete pieces are not usually created in one go. Composers create a number of sections of melody, harmonising them as needed, and possibly in different ways, and then structuring the piece around these basic sections. Composing between 4 bars and 16 bars is not only a computationally convenient task, it is actually what the human would do, creating components from which the whole is constructed. So although the system described here may be limited in its melodic scope, it has the potential to become a useful tool across a range of sub-styles.

Automatic Composition

A common problem in musical composition can be summarised in the question "where is the next note coming from?". For many composers over the years the answer has been to use some process to generate notes. It is clear that in many pieces from the Baroque period that simple note sequences are being elaborated in a fashion we would now call algorithmic. For this reason we can say that algorithmic composition is a subject that has been around for a very long time. It is usual to credit Mozart's Musikalisches Würfelspiel (Musical Dice Game) (Chuang 1995) as the oldest classical algorithmic composition, although there is some doubt if the game form is really his. In essence the creator provides a selection of short sections, which are then assembled according to a few rules and the roll of a set of dice to form a Minuet[1]. Two dice are used to choose the 16 minuet measures from a set of 176, and another die selects the 16 trio measures[2], this time from 96 possible. This gives a total number of 1.3×10^{29} possible pieces. This system however, while using some rules, relies on the coherence of the individual measures. It remains a fun activity, and recently web pages have appeared that allow users to create their own original(ish) "Mozart" compositions.

In the music of the second Viennese school ("12-tone", serial music) there is a process in action, rotating, inverting and use of retrograde, but usually performed by hand.

More recent algorithmic composition systems have concentrated on the generation of monody[3], either from a mathematical sequence, chaotic processes, or Markov chains, trained by consideration of acceptable other works. Frequently the systems rely on a human to select which monodies should be admitted, based on judgement rather than rules. Great works have been created this way, in the hands of great talents. Major descriptions of mathematical note generators can be found for example in *Formalized Music* (Xenakis 1992). Probably the best known of the Markov chain approach is Cope's significant corpus of Mozart pastiche (Cope 2006).

In another variation on this approach, the accompanist, either knowing the chord structure and style in advance, or using machine-listening techniques, infers a style of accompaniment. The former of these approaches can be found in commercial products, and the latter has been used by some jazz performers to great effect, for example by George E. Lewis.

A more recent trend is to cast the problem as one of constraint satisfaction. For example PWConstraints is an extension for IRCAM's Patchwork, a Common-Lisp-based graphical programming system for composition. It uses a custom constraint solver employing backtracking over finite integer domains. OMSituation and OMClouds are similar and are more recently developed for Patchwork's successor OpenMusic. A detailed evaluation of them can be found in (Anders 2007), where the author gives an example of a 1st-species counterpoint (two voices, note against note) after (Fux 1965 orig 1725) developed with Strasheela, a constraint system for music built on the multi-paradigm language Oz. Our musical rules however implement the melody and counterpoint rules described by (Thakar 1990), which we find give better musical results.

[1] A dance form in triple time, *i.e.* with 3 beats in each measure

[2] A Trio is a short contrasting section played before the minuet is repeated

[3] A monody is a single solo line, in opposition to homophony and polyphony

One can distinguish between *improvisation* systems and *composition* systems. In the former the note selection progresses through time, without detailed knowledge of what is to come. In practice this is informed either by knowing the chord progression or similar musical structures (Brothwell and ffitch 2008), or using some machine listening. In this paper we are concerned with *composition*, so the process takes place out of time, and we can make decisions in any order.

It should also be noted that these algorithmic systems compose pieces of music of this style in either a melodic or a harmonic fashion, and are frequently associated with computer-based synthesis. The system we will propose later is unique as it deals with both simultaneously.

Melodic Composition

In melodic generation a common approach is the use of some kind of probabilistic finite state automaton or an equivalent scheme, which is either designed by hand (some based on chaotic oscillators or some other stream of numbers) or built via some kind of learning process. Various Markov models are commonly used, but there have been applications of n-grams, genetic algorithms and neural nets. What these methods have in common is that there is no guarantee that melodic fragments generated have acceptable harmonic derivations. Our approach, described below is fundamentally different in this respect, as our rules cover both aspects simultaneously.

In contrast to earlier methods, which rely on learning, and which are capable of giving only local temporal structure, a common criticism of algorithmic melody (Leach 1999), we do not rely on learning and hence we can aspire to a more global, whole melody, approach. In addition we are no longer subject to the limitations of the kind of process which, because it only works in time in one direction, is hard to use in a partially automated fashion; for example operations like "fill in the 4 notes between these sections" is not a problem for us.

We are also trying to move beyond experiments with random note generation, which we have all tried and abandoned because the results are too lacking in structure. Predictably, the alternative of removing the non-determinism at the design stage (or replacing with a probabilistic choice) runs the risk of 'sounding predictable'! There have been examples of good or acceptable melodies created like this, but the restriction inherent in the process means it probably works best in the hands of geniuses.

Harmonic Composition

A common usage of algorithmic composition is to add harmonic lines to a melody; that is notes played at the same time as the melody that are in general consonant and pleasing. This is exemplified in the harmonisation of 4-part chorales, and has been the subject of a number of essays in rule-based or Markov-chain systems. Perhaps a pinnacle of this work is (Ebcioğlu 1986) who used early expert system technology to harmonise in the style of Bach, and was very successful. Subsequently there have been many other systems, with a range of technologies. There is a review included in (Rohrmeier 2006).

```
% At every time step, every part either steps
% to the next note in the key or leaps to a
% further note in the key
1 { stepUp(P,T), stepDown(P,T), leapUp(P,T),
   leapDown(P,T) } 1 :- part(P), time(T).

% A leap can only be over a consonant interval
% (3,4,5,7 or 12 semitones)
1 { leapBy(P,T,I) : consonantInterval(I) } 1
    :- leapUp(P,T).

% When a part leaps up by I, the note at time T+1
% is I steps higher than the current note
chosenNote(P,T+1,N+I) :- chosenNote(P,T,N),
                         leapBy(P,T,I).

% Every note must be in the chosen mode
% (major, minor, etc.)
:- chosenNote(P,T,N), mode(M), not inMode(N,M).

% The interval between parts must not be dissonant
:- chosenNote(P1,T,N1), chosenNote(P2,T,N2),
   interval(N1,N2,C), not consonantInterval(C).
```

Figure 1: A simplified ANTON fragment

Clearly harmonisation is a good match to constraint programming based systems, there being accepted rules[4]. It also has a history from musical education.

But these systems all start with a melody for which at least one valid harmonisation exists, and the program attempts to find one, which is clearly soluble. This differs significantly from our system, as we generate the melody and harmonisation together, the requirement for harmonisation affecting the melody.

The ANTON Composition System

What we are seeking to do, which is a new application in both music and computing, is to apply ASP techniques to compositional rules to produce an algorithmic composition system which can be applied more widely and freely than has previously been possible. $AnsProlog$ is used to create a description of the rules that govern the melodic and harmonic properties of a correct piece of music. The $AnsProlog$ program works as a model for music composition that can be used to assist the composer by suggesting, completing and verifying short pieces.

The rules of composition are modelled so that the $AnsProlog$ program defines the requirements for a piece to be valid, and thus every answer set corresponds to a valid piece. In generating an new piece, the composition system simply has to generate an (arbitrary) answer set. Rather than the traditional problem/solution mapping of answer set programming, this is using an $AnsProlog$ program to create a 'random' (arbitrary) example of a complex, structured object.

Figure 1 presents a simplified fragment of the $AnsProlog$ program used in ANTON. The model is defined over a number of time steps, given by the variable `T`. The key proposi-

[4]For example see: `http://www.wikihow.com/Harmonise-a-Chorale-in-the-Style-of-Bach`

tion is `chosenNote(P,T,N)` which represents the concept "At time `T`, part `P` plays note `N`". To encode the options for melodic progress ("the tune either steps up or down one note in the key, or it leaps more than one note"), choice rules are used. To encode the melodic limits on the pattern of notes and the harmonic limits on which combinations of notes may be played at once, constraints are included.

To allow for verification and diagnosis, each rule is given an error message:

```
% No tri-tones:  No note can be within two notes
% of a tritone (a note +/- 6 semitones)
#const err_tt="Tri-tone".
reason(err_tt).
error(P,T,err_tt) :-  chosenNote(P,T,N1),
                      chosenNote(P,T+2,N1+6).
error(P,T,err_tt) :-  chosenNote(P,T,N1),
                      chosenNote(P,T+2,N1-6).
```

Depending on how you want to use the system, composition or diagnosis, you will either be interested in those pieces that do not result in errors at all, or in an answer set that mentions the error messages. For the former we simply specify the constraint `:- error(P,T,R).`, effectively making any error rule into a constraint. For the latter we include the rules: `errorFound :- error(P,T,R).` and `:- not errorFound.`, requiring that an error is found (i.e. returning no answers if the diagnosed piece is error free).

By adding constraints on which notes can be included, it is possible to specify part or all of a melody, harmony or complete piece. This allows ANTON to be used for a number of other tasks beyond automatic composition. By fixing the melody it is possible to use it as an automatic harmonisation tool. By fixing part of a piece, it can be used as computer aided composition tool.

The complete system consists of three major phases; building the program, running the solver and interpreting the results. As a simple example suppose we wish to create a 4 bar piece in E major one would use the Perl wrapper and write

```
$ programBuilder.pl --task=compose \
                    --mode=major \
                    --time=16 > program
```

which builds the ASP program, giving the length and mode. Then

```
$ lparse -W all < program | \
  shuffle.pl 6298 | \
  smodels 1 > tunes
```

runs the grounder and solver and generates a representation of the piece. Using another Perl script we provide a number of output formats, one of which is a CSOUND (Boulanger 2000) program with a suitable selection of sounds.

```
$ parse.pl --fundamental=e --output=csound \
  < tunes > tunes.csd
```

generates the CSOUND input from the generic format, and then

```
$ csound tunes.csd -o dac
```

plays the melody. We provide in addition to CSOUND, output in text, $AnsProlog$ facts or the LILYPOND score language(Nienhuys and Nieuwenhuizen 2003). Naturally we provide scripts for all main ways of using the system.

```
keyMode(lydian).
chosenNote(1,1,25).
chosenNote(1,2,24).
chosenNote(1,8,19).
chosenNote(1,9,20).
chosenNote(1,10,24).
chosenNote(1,14,29).
chosenNote(1,15,27).
chosenNote(1,16,25).
#const t=16.
configuration(solo).
part(1).
```

Figure 2: musing.lp: An example of a partial piece

Alternatively we could request the system to complete part of a piece. In order to do so, we provide the system with a set of $AnsProlog$ facts expressing the mode (major, minor, etc.), the notes which are already fixed, the number of notes in your piece, the configuration and the number of parts. Figure 2 contains an example of such file. The format is the same as the one returned from the system except that all the notes in the piece will have been assigned.

We then run the system just as before with the exception of adding `--piece=musing.lp` when we run `programBuilder.pl`. The system will then return all possible valid compositions that satisfy the criteria set out in the partial piece.

The $AnsProlog$ programs used in ANTON contains less than 200 lines (not including comments, empty lines and user defined pieces) and encodes 28 melodic and harmonic rules. Once instantiated, the generated programs range from 3,500 atoms and 13,400 rules (a solo piece with 8 notes) to 11,000 atoms and 1,350,000 rules (a 16 note duet). The system is licensed under the GPL and is available, along with example pieces, from `http://www.cs.bath.ac.uk/~mjb/` . Figure 3 contains an extract from a series of simple duets composed by ANTON.

It should be noted that ANTON's 200 lines of code contrast with the 8000 lines in Strasheela (Anders 2007) and 88000 in Bol (Bel 1998). For this reason we claim that our representation of the musical problem is easily read and understood.

Evaluation of ANTON

Practical Use

All this construction is of little use if the system is not practical to use, so we benchmarked a variety of solvers using the programs ANTON generated. Table 1 contains the times taken by a number of answer set solvers (SMODELS (Syrjänen and Niemelä 2001), SMODELS-IE (Brain, De Vos, and Satoh 2007), SMODELSCC (Ward and Schlipf 2004), CMODELS (Lierler and Maratea 2004) and CLASP (Gebser et al. 2007)) in composing a single piece of a given length. Likewise Table 2 contains the times taken to compose a two part piece of a given length. LPARSE (Syrjänen 2000) was used to ground the programs and its run time, typically around 30-60 seconds, is omitted from the results.

All times were recorded using a 2.4GHz AMD Athlon X2 4600+ processor, running a 64 bit version of OpenSuSE 10.3. All solvers were built in 32 bit mode. Each run was limited to 20 minutes of CPU time and 2Gb of

	smodels 2.32		smodels-ie 1.0.0		smodelscc 1.08	cmodels 3.75		clasp 1.0.5
Length	Default	Restarts	Default	Restarts	No lookahead	w/ zchaff	w/ MiniSAT	Default
4	1.02	1.03	0.09	0.09	1.17	0.33	0.39	0.22
6	2.43	2.43	0.38	0.38	2.58	0.64	0.85	0.46
8	5.16	5.16	1.03	1.04	4.94	1.06	1.62	1.01
10	12.25	11.72	2.58	2.59	8.55	1.54	2.63	1.33
12	28.25	46.13	8.08	15.14	11.36	2.42	4.04	2.27
14	40.62	140.00	10.50	43.54	18.78	3.14	6.05	3.48
16	101.05	207.25	29.40	69.53	27.94	4.01	9.40	4.62

Table 1: Time taken (in seconds) for a number of solvers generating a solo piece

	smodels 2.32		smodels-ie 1.0.0		smodelscc 1.08	cmodels 3.75		clasp 1.0.5
Length	Default	Restarts	Default	Restarts	No lookahead	w/ zchaff	w/ MiniSAT	Default
4	3.77	3.77	0.31	0.32	4.08	1.18	1.26	0.77
6	10.36	11.24	1.89	1.89	13.90	2.17	2.81	1.60
8	54.64	77.10	14.71	21.84	26.07	3.88	5.93	3.73
10	Time out	Time out	Time out	500.26	78.72	9.51	11.12	9.34
12	Time out	Time out	Time out	Time out	103.81	14.50	18.14	16.84
14	Time out	Time out	Time out	Time out	253.92	32.41	32.34	25.59
16	Time out	Time out	Time out	Time out	452.38	82.64	49.29	29.63

Table 2: Time taken (in seconds) for a number of solvers generating a duet

RAM. The $AnsProlog$ programs used are available from http://www.cs.bath.ac.uk/~mjb/ .

These results show that the system, when using the more powerful solvers, is fast enough to be used as a component in an interactive composition tool. Further work would be needed to support real time generation of music, but we are not too far away. We also note that the only solvers able to generate longer sequences in two parts all implement clause learning strategies, which suggests that the problem is particularly susceptible to this kind of technique.

Music Quality

Judging quality is a subjective process; after all we do not all like the same music. However we assert that the music is acceptable, at least by the standards of a student of composition, and at times there are moments of excitement. For the reader to judge we show in Figure 3 part of ANTON's Opus 1: *Twenty Short Pieces*; the audio files of the complete work can be found in http://cs.bath.ac.uk/~mjb.

ASP as the Representation and Reasoning Language

One of the main results reported in this paper is how easy it was to encode the rules in terms both of ease of expression and of ease of capturing the rules. Composers can think of ANTON as a testbed for experimentation with musical expertise that can be formulated as essential musical rules for all musical parameters[5] in order to build relations between those that either comply with a certain musical style or that open up new musical experiences. We have made the case that a sub-style of Renaissance Counterpoint and its melodic style can be represented with $AnsProlog$. The flexibility of creating different solutions based on the same rules offers the composer the opportunity to discover areas that he might have never thought of. It is recognised among musicians that an important component of composition is the use of the listener's expectations to obtain an effect. The composer plays on the listener by either satisfying his expectations or surprising him by not doing what he was anticipating. This facet of music is not confined to the experience of someone listening to a completed piece of music. It is inherent in the entire creative process, since a composer is also his own first listener. Subtle changes of $AnsProlog$ facts given to ANTON can give surprising results, as can small and skillful adjustments to the set of rules to produce a break with the previous set, thus allowing the composer to create the moldings for his own creations in a step-by-step process. This is a multi-faceted feedback-loop between writing the rules, listening to and examining the musical outcomes and modifying the rules if necessary. One of the most important musical tasks is to be able to work with impulse and resolution on multiple levels and with different voices simultaneously. This gathering and dissipation of musical energy (Thakar 1990) can happen in an infinite number of ways where all musical lines, defined by their various parameters, participate together. This process has to take into account all past musical events as well as the fragile balance between different voices and their parameters. Therefore, the modelling of rules that can capture musical impulse and resolution proves to be the most challenging aspect of writing programs with ANTON. There are currently no other programs known to us offering solutions in this direction and much of our future work will need to focus on this particular challenge.

There are also some negative points. One persistent problem was the lack of mature development support

[5]For a single note commonly known as pitch, loudness, timbre (sound quality) and duration.

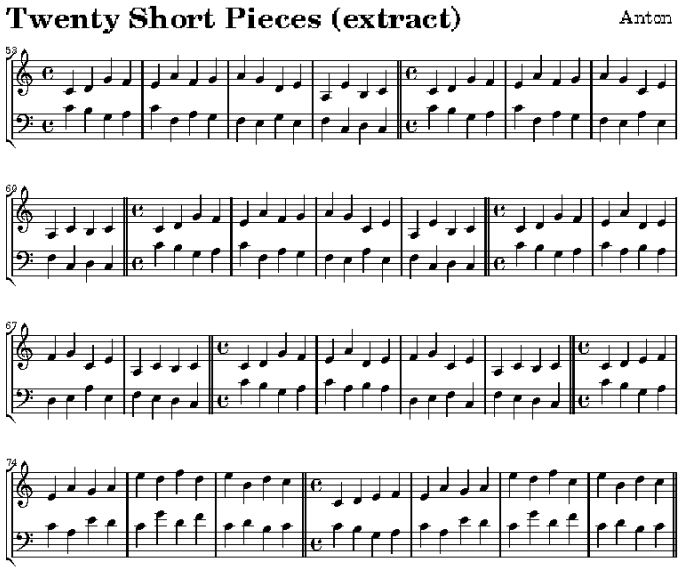

Figure 3: Part of a set of pieces composed by the system

tools, particularly debugging tools. SPOCK (Brain et al. 2007) was used, but as its focus is on computing the reasons behind the error, rather than the interface issues of explaining these reasons to the user, it was normally quicker to find bugs by looking at the last changes made and which regression tests failed. Generally, the bugs that were encountered where due to subtle mismatches between the intended meaning of a rule and the declarative reading of the rule used. For example the predicate `stepUp(P,T)` is used to represent the proposition "At time T, part P steps up to give the note at time T+1"; however, it could easily be misinterpreted as "At time T-1, part P steps up to give the note at time T". Which of these is used is not important, as long as the same declarative reading is used for all rules. With the first "meaning" selected for ANTON, the rule:

```
chosenNote(P,T,N+S) :-  chosenNote(P,T-1,N),
                        stepUp(P,T),
                        stepBy(P,T,S).
```

would not encode the intended progression of notes. To avoid these errors it would be possible to develop a system that translated rules into natural language, given the declarative reading of the propositions involved. It should then be relatively straightforward to check that the rule encoded what was intended.

The Future

Music Research

This system could develop in a number of novel ways. For example we might throw light on the compositional process by learning aspects of the rules, finding which are inconsistent or redundant, or determining the importance of rules. We could investigate whether there are "unspoken" rules, and experiment to find unacknowledged rules of composition. One particularly interesting possibility is using the system to generate a large set of pieces, acquiring human evaluations of the 'quality' of each and then using techniques such as inductive logic programming to infer rules for composing 'good' pieces.

So far we have only considered a particular style of Western music. However the framework should be applicable to other styles, especially formal ones. *e.g.* the rules of Hindustani classical music are taught in a traditional, oral, fashion, but we see no reason why our framework could not capture these. Recent work (Endrich 2008) indicates that there are indeed universal melodic rules, and the combination of the ASP methodology with this musical insight is an intriguing one.

In real life pieces some of the rules are sometimes broken. This could be simulated by one of a number of extensions to answer set semantics (preferences (Brain and De Vos 2003), consistency restoring rules, defensible rules, etc.). However how to systematise the knowledge of when it is acceptable to break the rules and in which contexts it is 'better' to break them is an open problem.

A major deficiency of the current system is the lack of rhythm, as all parts play all the time (with no rests), with notes of equal duration, which, while usual in some styles, stands in the way of a whole range of interesting variety. We have not considered rhythm so far, but one of us is already researching rhythmic structures and performance gesture (Boenn 2007), so in the longer term this may be incorporated.

Systems Development

The current system can write short melodies effectively and efficiently. Development work is still needed to extend this to entire pieces; we can start from these melodic fragments but a longer piece needs a variety of different harmonisations for the same melody, and related melodies with the same harmonic structure and a number of similar techniques. We have not solved the difficult global structure problem but our system is a starting point on which we can build a structure

that is hierarchical over time scales; we have a mechanism for building syntactically correct sentences, but these need to be built into paragraphs and chapters, as it were.

Our results seem to suggest that a real-time composition system is possible, which would open up the possibility for performance and improvisation. Profiling of the current system has indicated that some conceptually simple tasks, like parsing, are taking a disproportionate fraction of the runtime, and some engineering would assist in removing these problems. Clearly this is one of a number of system-like issues that need to be addressed. Also, the availability of a parallel answer set solver that implements clause learning would help in building this type of application.

An obvious extension to the composition of duets is to expand this to three and four parts, by adding inner voices, with their different rules.

Answer Set Synthesis

What we are doing is not answer set *programming* in the classical sense because we are not solving a problem *per se*; we are generating an arbitrary representative instance. Although this may seem like a subtle shift of emphasis, it has a number of interesting implications. Firstly for applications, this takes NMR into 'procedural synthesis' of all kinds of things — by describing what objects are, we can construct arbitrary examples. This has some interesting possibilities for computer games and virtual worlds where 'randomly generated' content is needed. This content has to be non-repetitive; increasingly it needs to be complex and structured (and have some fixed, hard properties — such as there must be a way out of the maze). In the case of things such as background music, high 'artistic merit' is not as important as consistent and non-repetitive.

This also raises questions for solver and language design. Most solvers are calibrated towards 'hard' problems, that are large (but tractable search spaces) with relatively few answer sets. However programs from this answer set synthesis school of application tend to have huge (intractable) search spaces with a very large number of answer sets. Thus the emphasis in solving shifts towards getting to an answer quickly (assuming that many paths lead to solutions) rather than trying to reduce and cover the search space efficiently. This is likely to influence the choice of branching heuristic. There are also a number of other interesting areas which come to mind. A metric for distance between answer sets would allow a solver to generate "things similar to the last solution", "things similar to the last solution but not too similar", "things as different as possible to the last solution", *etc*. Also there is a need for schemes for handling preferences ("it's not impossible to have A and B, but it should be avoided if possible") and probabilities ("there is a choice between A and B but in 90% of solutions it should be A"), particularly as part of the solving process, rather than needing to compute multiple solutions and then refining or optimising the choice between them, again probably in the heuristic.

Conclusion

In this paper, we have presented ANTON, the first algorithmic composing system that is capable of both melodic and harmonic composition. By using answer set programming as our modelling language for the technical rules that underpin music composition, we have obtained a highly flexible, extremely compact and efficient system. As all the rules are simultaneously available the system enables us to explore the rules themselves, to evaluate pieces for rule compliance, to complete partial systems, such as producing a melody consonant with a given harmony structure, as well as, more adventurously, to create new melodies.

We have demonstrated that current ASP systems can be used to generate aesthetically acceptable music within an appropriate time frame.

The development of ANTON opens up interesting research ideas both in the musicological direction and in declarative programming in general and more specifically answer set programming. In particular we have identified a number of ways in which ASP solvers could be extended so as to widen their application.

References

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., and Gelfond, M. 2000. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence*, 257–279. Kluwer Academic Publishers.

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

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

Brain, M., and De Vos, M. 2003. Implementing OCLP as a Front End for Answer Set Solvers: From Theory to Practice. In *Proceedings of Answer Set Programming: Advances in Theory and Implementation (ASP'03)*. Ceur-WS.

Brain, M.; Crick, T.; De Vos, M.; and Fitch, J. 2006. TOAST: Applying Answer Set Programming to Superoptimisation. In *International Conference on Logic Programming*, LNCS. Springer.

Brain, M.; Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2007. "That is illogical captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description. In *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, 71–85.

Brain, M.; De Vos, M.; and Satoh, K. 2007. Smodels-ie : Improving the Cache Utilisation of Smodels. In Costantini, S., and Watson, R., eds., *Proceedings of the 4th Workshop on Answer Set Programming*, 309–314.

Brothwell, A., and ffitch, J. 2008. An Automatic Blues Band. In Barknecht, F., and Rumori, M., eds., *6th International Linux Audio Conference*, 12–17. Kunsthochscule für Medien Köln: LAC2008.

Buccafurri, F., and Caminiti, G. 2005. A Social Semantics for Multi-agent Systems. In *8th International Conference of Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *LNCS*, 317–329. Springer.

Chuang, J. 1995. Mozart's Musikalisches Würfelspiel. `http://sunsite.univie.ac.at/Mozart/dice/`.

Cliffe, O.; De Vos, M.; and Padget, J. 2006. Specifying and Analysing Agent-based Social Institutions using Answer Set Programming. In Boissier, O.; Padget, J.; Dignum, V.; Lindemann, G.; Matson, E.; Ossowski, S.; Sichman, J.; and Vazquez-Salceda, J., eds., *Selected revised papers from the workshops on Agent, Norms and Institutions for Regulated Multi-Agent Systems (ANIREM) and Organizations and Organization Oriented Programming (OOOP) at AAMAS'05*, volume 3913 of *LNCS*, 99–113. Springer Verlag.

Cope, D. 2006. A Musical Learning Algorithm. *Computer Music Journal* 28(3):12–27.

Ebcioğlu, K. 1986. *An Expert System for Harmonization of Chorales in the Style of J.S. Bach*. Ph.D. Dissertation, State University of New York, Buffalo, Department of Computer Science.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In *KR'98: Principles of Knowledge Representation and Reasoning*. San Francisco, California: Morgan Kaufmann. 406–417.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2002. The DLV^K Planning System. In Flesca, S.; Greco, S.; Leone, N.; and Ianni, G., eds., *European Conference, JELIA 2002*, volume 2424 of *LNAI*, 541–544. Cosenza, Italy: Springer Verlag.

Endrich, A. 2008. *Building Musical Relationships*. In preparation. *seen in manuscript*.

Erdem, E.; Lifschitz, V.; Nakhleh, L.; and Ringe, D. 2003. Reconstructing the Evolutionary History of Indo-European Languages Using Answer Set Programming. In Dahl, V., and Wadler, P., eds., *PADL*, volume 2562 of *LNCS*, 160–176. Springer.

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

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. S., eds., *LPNMR*, volume 4483 of *LNCS*, 266–271. Springer.

Grell, S.; Schaub, T.; and Selbig, J. 2006. Modelling biological networks by action languages via answer set programming. In Etalle, S., and Truszczyński, M., eds., *Proceedings of the International Conference on Logic Programming (ICLP'06)*, volume 4079 of *LNCS*, 285–299. Springer-Verlag.

Konczak, K. 2006. Voting Theory in Answer Set Programming. In Fink, M.; Tompits, H.; and Woltran, S., eds., *Proceedings of the Twentieth Workshop on Logic Programming (WLP'06)*, number INFSYS RR-1843-06-02 in Technical Report Series, 45–53. Technische Universität Wien.

Leach, J. L. 1999. *Algorithmic Composition and Musical Form*. Ph.D. Dissertation, University of Bath, School of Mathematical Sciences.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, 346–350. Springer.

Lifschitz, V. 2002. Answer set programming and plan generation. *J. of Artificial Intelligence* 138(1-2):39–54.

Nienhuys, H.-W., and Nieuwenhuizen, J. 2003. Lilypond, A System For Automated Music Engraving. In *Proceedings of the XIV Colloquium on Musical Informatics*.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. A A-Prolog Decision Support System for the Space Shuttle. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Represenation and Reasoning*. Stanford (Palo Alto), California, US: American Association for Artificial Intelligence Press.

Polleres, A. 2005. Semantic Web Languages and Semantic Web Services as Application Areas for Answer Set Programming. In *Nonmonotonic Reasoning, Answer Set Programming and Constraints*. IBFI.

Rohrmeier, M. 2006. Towards modelling harmonic movement in music: Analysing properties and dynamic aspects of pc set sequences in Bach's chorales. Technical Report DCRR-004, Darwin College, University of Cambridge.

Ruffolo, M.; Leone, N.; Manna, M.; Saccà, D.; and Zavatto, A. 2005. Exploiting ASP for Semantic Information Extraction. In De Vos, M., and Provetti, A., eds., *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Syrjänen, T., and Niemelä, I. 2001. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*.

Syrjänen, T. 2000. *Lparse 1.0 User's Manual*. Helsinki University of Technology.

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

Ward, J., and Schlipf, S. 2004. Answer Set Programming with Clause Learning. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*. Springer.

Xenakis, I. 1992. *Formalized Music*. Stuyvesant, NY, USA: Bloomington Press.