# Automatic Composition of Melodic and Harmonic Music by Answer Set Programming

Georg Boenn[1], Martin Brain[2], Marina De Vos[2], and John ffitch[2]

[1] Cardiff School of Creative & Cultural Industries
University of Glamorgan
Pontypridd, CF37 1DL, UK
gboenn@glam.ac.uk
[2] Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
{mjb,mdv,jpff}@cs.bath.ac.uk

**Abstract.** The composition of most styles of music is governed by rules. The natural statement of these rules is declarative ("The highest and lowest notes in a piece must be separated by a consonant interval") and non deterministic ("The base note of a key can be followed by any note in the key"). We show that by approaching the automation and analysis of composition as a knowledge representation task and formalising these rules in a suitable logical language, powerful and expressive intelligent composition tools can easily be built. This paper describes the use of answer set programming to construct an automated system that can compose both melodic and harmonic music, diagnose errors in human compositions and serve as a computer-aided composition tool. The use of a fully declarative language and an "off-the-shelf" reasoning engine allows the creation of tools which are significantly simpler, smaller and more flexible than those produced by existing approaches. It also combines harmonic and melodic composition in a single framework, which is a new feature in the growing area of algorithmic composition.

## 1 Introduction

Music, although it seeks to communicate via emotions, is almost always governed by complex and rigorous rules which provide the base from which artistic expression can be attempted. In the case of musical composition, in most styles there are rules which describe the progression of a melody, both at the local level (the choice of the next note) and at the global level (the overall structure). Other rules describe the harmony, which arises from the relationship between the melodic line and the supporting instruments.

These rules were developed to guide and support human composers working in the style of their choice, but we wish to demonstrate here that by using knowledge representation techniques, we can create a computer system that can reason about and apply compositional rules. Such a system will provide a simple and flexible way of composing music automatically, but, provided that the representation technology used is sufficiently flexible to allow changes at the level of the rules themselves, it will also

help the human composer to understand, explore and extend the rules he is working with.

This paper describes ANTON, an automatic composition system based on an implementation of one set of compositional rules, those governing tonal Western music, using Answer Set Programming (ASP). The paper provides an overview of the musical context and the particular problems of algorithmic composition on both melodic and harmonic forms. This is followed by a description of the ASP that we use, before giving more details of our innovative system, its design, performance and outputs. We conclude with directions for future work in both music research and the development of our system.

## 2 Music

Creating melodies, that is sequences of pitched sounds, is not as easy as it looks. 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.

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 Section 5.

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 an appropriate task, it is actually what the human would do, creating component form 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.

## 3 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) [1] 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[3]. Two dice are used to choose the 16 minuet measures from a set of 176, and another die selects the 16 trio measures[4], this time from 96 possible. This gives a total number of $1.3 \times 10^{29}$ 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.

More recent algorithmic composition systems have concentrated on the generation of monody[5], 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* [2]. Probably the best known of the Markov chain approach is Cope's significant corpus of Mozart pastiche [3].

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.

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 were more recently developed for Patchwork's successor OpenMusic. A

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

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

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

detailed evaluation of them can be found in [4], where the author gives an example of a 1st-species counterpoint (two voices, note against note) after [5] 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 [6], which we find give better musical results.

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 [7], 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 this music of this style in either a melodic or a harmonic fashion, and are frequently associated with computer-based synthesis. We consider these two sub-problems separately.

### 3.1 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 [8], 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.

### 3.2 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 [9] 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 [10].

Clearly harmonisation is a good match to constraint programming based systems, there being accepted rules[6]. 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.

## 4 Answer Set Programming

Due to space constraints, only a brief overview of answer set semantics and Answer Set Programming (ASP) is given here. The interested reader is referred to [11] 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 [11], we refer to the language over which answer set semantics is defined as $AnsProlog$. Programs in $AnsProlog$ are sets of rules of the form:

$$a \leftarrow b, \text{not } c.$$

where $a$, $b$ and $c$ are atoms. Intuitively, this means "if b is known and c is not known, then a is known". 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 are satisfied with respect to a set of atoms if either the body is false or the head is true. Rules with empty bodies are called *facts*; their head should always true.

If a program $\Pi$ contains no negated atoms ($\forall r \in \Pi \,.\, B^-(\Pi) = \emptyset$) 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. This fixpoint is called the *answer set*. For example, given the following program:

$$a \leftarrow b, c.$$
$$b \leftarrow c.$$
$$c \leftarrow .$$
$$d \leftarrow e.$$
$$e \leftarrow d.$$

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

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 known 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 $\Pi$ and a set of atoms $A$, the Gelfond-Lifschitz transform of $\Pi$ with respect to $A$ is the following set of rules:*

$$\Pi^A = \{H(r) \leftarrow B^+(r) | r \in \Pi, B^-(r) \cap A = \emptyset\} \tag{1}$$

This allows us to define the concept of *answer sets*. 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 $\Pi$, $A$ is an answer set of $\Pi$ $\iff$ $A$ is the unique answer set of $\Pi^A$.*

For example, the following program has two answer sets:

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

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

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

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

A given program will have zero or more answer sets and computing an answer set is NP complete.

When used as a knowledge representation language, $AnsProlog$ is enhanced to contain constraints (e.g. : $-b, \text{not } c$) and choice rules (*e.g.* $\{a, b, c\} : -b, \text{not } c$). The former are rules with an empty head, stating that a valid answer set should not make the body true. The latter is a short hand notation for expression that a certain number of atoms need to be true under certain circumstances. These are syntactic sugar and can be removed with linear, modular transformations (see [11]). Variables and predicated rules are also used and are handled, at the theoretical level and in most implementations, by instantiation (referred to as *grounding*).

ASP is a programming paradigm in which a problem is *represented* as an $AnsProlog$ program in such a way that the answer sets correspond to 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. These answer sets are then *interpreted* to give solutions of the original problem. GRINGO[12] and LPARSE[13] are the grounders most commonly used and CLASP[14], SMODELS[15], CMODELS[16] and DLV[17] represent the state of the art of solver development.

ASP has been used to tackle a variety of problems, including: planning and diagnosis [18–20], modelling and rescheduling of the propulsion system of the NASA Space Shuttle [20], multi-agent systems [21–23], Semantic Web and web-related technologies [24, 25], superoptimisation [26], reasoning about biological networks [27], voting theory [28] and investigating the evolution of language [29].

## 5 The ANTON 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 a system which can be applied more widely and freely than has previously been possible. ASP is used to create a description of the rules that govern the melodic and harmonic properties of correct piece of music. The ASP program works as a model for music composition that can be used to assist the composer by suggesting, completing and verifying short pieces.

Rather than create a procedural or probabilistic algorithm for producing music, ANTON takes the approach of representing the rules of what constitutes a valid piece and then searching for pieces that meet this specification. 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 a 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' 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 proposition is `chosenNote(P,T,N)` which represents the concept "At time `T`, part

```
% 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
choosenNote(P,T+1,N+I) :- choosenNote(P,T,N), leapBy(P,T,I).

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

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

**Fig. 1.** A simplified ANTON fragment

`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) :- choosenNote(P,T,N1), choosenNote(P,T+2,N1+6).
error(P,T,err_tt) :- choosenNote(P,T,N1), choosenNote(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 results into errors or in an answer set that mentions the error messages. For the former we simply specify the constraint `:- error(P,T,R).` For the latter we include the rules: `errorFound :- error(P,T,R).` and `:- not errorFound.`.

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. By fixing a complete piece, it is possible to check its conformance to the rules, for marking student compositions or harmonisations.

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

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

**Fig. 2.** musing.lp: An example of a partial piece

runs the ASP phase and generates a representation of the piece. We provide a number of output formats, one of which is a CSOUND [30] 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, ASP facts or the Lilypond score language, with MIDI under development. Naturally we provide scripts for all main ways of using the system.

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 ASP facts expressing the keyMode, 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 composition that satisfy the criteria set out in the partial piece.

The $AnsProlog$ programs used in ANTON contains just 191 lines (not including comments and empty lines) 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). Scripts are provided to convert the answer sets generated into output for the CSOUND synthesis system and the LILYPOND notation tool. 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 produced by the system.

It should be noted that the 500 lines of code here contrast with the 8000 lines in Strasheela[4] and 88000 in Bol[31]. For this reason we claim that our representation of the musical problem is easily read and understood.

| | 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

# 6 Evaluation of ANTON

## 6.1 Practical Use

To assess the practicality of using answer set programming to create a composition system a number of tests were performed. Table 1 contains the times taken by a number of answer set solvers (SMODELS [15], SMODELS-IE [32], SMODELSCC [33], CMODELS [16] and CLASP [14]) 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 [13] was used to ground the programs and its run time, typically around 30-60 seconds, is omitted from the results.

All times where 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 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. It is also interesting to note that the only solvers able to generate longer sequences using two parts all implement clause learning strategies, suggesting that the problem is particularly susceptible to this kind of technique.

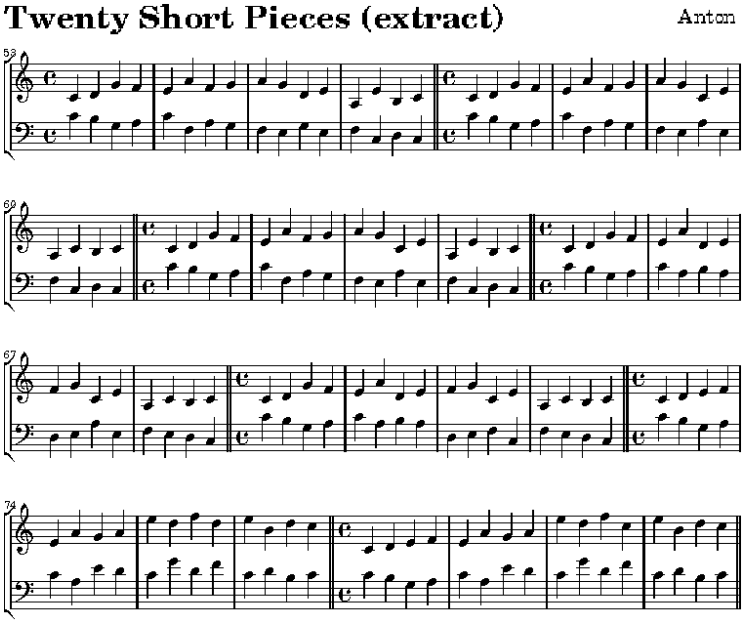


**Fig. 3.** Part of a set of pieces composed by the system

### 6.2 Music Quality

The other way to evaluate the system is to judge the music it produces. This is less certain process, involving personal values. However we feel that the music is acceptable, at least of the quality of a student of composition, and at times capable of moments of excitement. The first composition, part of which is shown in Figure 3, consisting of twenty short melodies[7] shows promise with real musical moments, but the only real evaluation is for the reader to listen; the web site provides the sounds.

### 6.3 ASP as the Knowledge Representation Language

In constructing ANTON a number of advantages of using answer set programming have become clear; as have a number of limitations.

Firstly, $AnsProlog$ is very fast to write and very compact. As well as the obvious benefits, this means it is possible to develop the system at the same time as undertaking knowledge capture and to prototype features in the light of the advice of domain experts. Part of the reason why it is so fast to use is that rules are fully declarative. Programming thus focuses on expressing the concepts that are being modelled rather than having to worry about which order to put things in - such as which rules should come first, which concepts have higher priority, which choices should be made first. This also makes

[7] We call this ANTON's Opus 1: Twenty Short Pieces

incremental development easy as new constraints can be added one at a time, without having to consider how they affect the search strategy.

Being able to add rules incrementally during development turns out to be extremely useful from a software engineering view point. During the development of ANTON, we experimented with a number of different development methodologies. The most effective approach was found to be first writing a script that translates answer sets to human readable score or output for a synthesiser. Next the choice rules were added to the $AnsProlog$ program to create all possible pieces, valid or not. Finally the constraints were incrementally added to restrict the output to only valid sequences. By building up a library of valid pieces it was possible to perform regression testing at each step and thus isolate bugs as soon as they were introduced.

Using answer set programming was not without issue. One persistent problem was the lack of mature development support tools, particularly debugging tools. SPOCK [34] 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 where 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. One possible way of supporting a programmer in avoiding these subtle errors would be 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.

## 7 Conclusions and Directions for Future Work

We have built a sophisticated composition system with adequate run time performance.

Using knowledge representation techniques it is possible to create an automatic composition system that is significantly smaller, simpler and more expressive than the current state of the art. The choice of using a pure declarative language, $AnsProlog$, allows the system to be flexible enough to be used as a platform for research into the rules of composition.

There are a number of possible lines of development for the ANTON system, both in terms of the musical rules it contains and the supporting system.

### 7.1 Music Research

The system provides a platform for a novel approach to music research. We can learn aspects of the rules, finding which are inconsistent or redundant, and can determine the importance of rules. We hope that this will throw light on the compositional process. We can see if there are any "unspoken" rules of composition, and also the related, finding unknown 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.

The work so far has been limited to a particular style of Western music. However the framework should be applicable to other styles, especially formal ones. The rules of Hindustani classical music are taught to pupils in a traditional, oral, fashion, but we see no reason why this framework cannot capture these. Recent work [35] 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 pieces some of the rules are sometimes broken. This could be simulated by one of a number of extensions to answer set semantics (preferences [36], consistency restoring rules, defensible rules, etc.). 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.

One deliberate simplification of the current system is the lack of rhythm as the style of composition we are implementing traditionally contain few explicit restrictions. So all parts play all the time, with notes of equal duration. While usual in some styles, this obviates a whole range of interesting variety. We have not yet considered rhythm, but one of us is already researching rhythmic structures and performance gesture [37], so in the longer term this may be incorporated.

### 7.2 Systems Development

The current system can write short melodies effectively and efficiently. Development work is still needed to take 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 it does create a starting point on 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.

The system performance currently seems 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 run-time, 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. It should perhaps be noted that inner voices obey different rules, and these need to be implemented.

## References


1. Chuang, J.: Mozart's Musikalisches Würfelspiel. `http://sunsite.univie.ac.at/Mozart/dice/` (1995)
2. Xenakis, I.: Formalized Music. Bloomington Press, Stuyvesant, NY, USA (1992)
3. Cope, D.: A Musical Learning Algorithm. Computer Music Journal **28**(3), pp 12-27 (Fall 2006)
4. Anders, T.: Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System. PhD thesis, Queen's University, Belfast, Department of Music (2007)
5. Fux, J.: The Study of Counterpoint from Johann Joseph Fux's Gradus ad Parnassum. W.W. Norton (1965, orig 1725)
6. Thakar, M.: Counterpoint. New Haven (1990)
7. Brothwell, A., ffitch, J.: An Automatic Blues Band. In Barknecht, F., Rumori, M., eds.: 6th International Linux Audio Conference, Kunsthochscule für Medien Köln, LAC2008, pp 12-17 (March 2008)
8. Leach, J.L.: Algorithmic Composition and Musical Form. PhD thesis, University of Bath, School of Mathematical Sciences (1999)
9. Ebcioğlu, K.: An Expert System for Harmonization of Chorales in the Style of J.S. Bach. PhD thesis, State University of New York, Buffalo, Department of Computer Science (1986)
10. Rohrmeier, M.: 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 (2006)
11. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. 1st edn. Cambridge University Press (2003)
12. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A New Grounder for Answer Set Programming. In Baral, C., Brewka, G., Schlipf, J.S., eds.: LPNMR 2007. LNCS, vol 4483, pp 266-271. Springer Heidelberg (2007)
13. Syrjänen, T.: Lparse 1.0 User's Manual. Helsinki University of Technology. (2000)
14. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: Proceeding of IJCAI07, pp 386-392 (2007)
15. Syrjänen, T., Niemelä, I.: The Smodels System. In: ICLP 2001 (2001)
16. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR 2004 LNCS, vol 2923, pp 346-350. Springer Heidelberg (2004)
17. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Cohn, A.G., Schubert, L., Shapiro, S.C., eds.: KR'98: Principles of Knowledge Representation and Reasoning, pp 406–417 Morgan Kaufmann, San Francisco, California (1998)
18. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: The DLV$^K$ Planning System. In Flesca, S., Greco, S., Leone, N., Ianni, G., eds.: JELIA 2002, LNAI, vol 2424, pp 541-544. Springer Verlag (September 2002)
19. Lifschitz, V.: Answer set programming and plan generation. J. of Artificial Intelligence **138**(1-2), pp 39-54 (2002)
20. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: A A-Prolog Decision Support System for the Space Shuttle. In: Answer Set Programming: Towards Efficient and Scalable Knowledge Represenation and Reasoning. American Association for Artificial Intelligence Press, Stanford (Palo Alto), California, US (March 2001)

21. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Logic-based artificial intelligence, pp 257-279. Kluwer Academic Publishers (2000)
22. Buccafurri, F., Caminiti, G.: A Social Semantics for Multi-agent Systems. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: LPNMR 2005, LNCS, vol 3662, pp 317-329 Springer, Heidelberg (2005)
23. Cliffe, O., De Vos, M., Padget, J.: 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., 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, LNCS, vol 3913, pp 99-113. Springer Verlag (2006)
24. Polleres, A.: Semantic Web Languages and Semantic Web Services as Application Areas for Answer Set Programming. In Brewka, G., Niemelä, I., Schaub, T., Truszczynski, M., eds.: Nonmonotonic Reasoning, Answer Set Programming and Constraints. Volume 05171 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
25. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In De Vos, M., Provetti, A., eds.: Answer Set Programming. Volume 142 of CEUR Workshop Proceedings., CEUR-WS.org (2005)
26. Brain, M., Crick, T., De Vos, M., Fitch, J.: TOAST: Applying Answer Set Programming to Superoptimisation. In: International Conference on Logic Programming. LNCS, Springer Heidelberg (August 2006)
27. Grell, S., Schaub, T., Selbig, J.: Modelling biological networks by action languages via answer set programming. In Etalle, S., Truszczyński, M., eds.: ICLP'06, LNCS, vol 4079, pp 285-299. Springer-Verlag (2006)
28. Konczak, K.: Voting Theory in Answer Set Programming. In Fink, M., Tompits, H., Woltran, S., eds.: Proceedings of the Twentieth Workshop on Logic Programming (WLP'06). Number INFSYS RR-1843-06-02, pp 45-53 in Technical Report Series, Technische Universität Wien (2006)
29. Erdem, E., Lifschitz, V., Nakhleh, L., Ringe, D.: Reconstructing the Evolutionary History of Indo-European Languages Using Answer Set Programming. In Dahl, V., Wadler, P., eds.: PADL, LNCS, vol 2562, pp 160-176. Springer (2003)
30. Boulanger, R., ed.: The Csound Book. MIT Press (2000)
31. Bel, B.: Migrating Musical Concepts: An Overview of the Bol Processor. Computer Music Journal **22**(2) (1998), pp 56–64
32. Brain, M., De Vos, M., Satoh, K.: Smodels-ie : Improving the Cache Utilisation of Smodels. In Costantini, S., Watson, R., eds.: Proceedings of the 4th Workshop on Answer Set Programming, pp 309-314 (2007)
33. Ward, J., Schlipf, S.: Answer Set Programming with Clause Learning. In: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning. Volume 2923 of LNCS., Springer (2004)
34. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: "That is illogical captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description. In De Vos, M., Schaub, T., eds.: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 71–85
35. Endrich, A.: Building Musical Relationships. In preparation (2008) *seem in manuscript*.
36. Brain, M., De Vos, M.: 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 (September 2003)
37. Boenn, G.: Composing Rhythms Based Upon Farey Sequences. In: Digital Music Research Network Conference. (July 2007)