Connection Science

ISSN: 0954-0091 (Print) 1360-0494 (Online) Journal homepage: http://www.tandfonline.com/loi/ccos20

Towards transformational creation of novel songs

Jukka M. Toivanen, Matti Järvisalo, Olli Alm, Dan Ventura, Martti Vainio & Hannu Toivonen

To cite this article: Jukka M. Toivanen, Matti Järvisalo, Olli Alm, Dan Ventura, Martti Vainio & Hannu Toivonen (2018): Towards transformational creation of novel songs, Connection Science, DOI: 10.1080/09540091.2018.1443320

To link to this article: https://doi.org/10.1080/09540091.2018.1443320

CONNECTION SCIENCE, 2018
https://doi.org/10.1080/09540091.2018.1443320

Towards transformational creation of novel songs

Jukka M. Toivanen [a], Matti Järvisalo [a], Olli Alm[b], Dan Ventura [c], Martti Vainio [a] and Hannu Toivonen [a]

[a]University of Helsinki, Helsinki, Finland; [b]Metropolia University of Applied Sciences, Helsinki, Finland; [c]Brigham Young University, Provo, UT, USA

ABSTRACT

We study transformational computational creativity in the context of writing songs and describe an implemented system that is able to modify its own goals and operation. With this, we contribute to three aspects of computational creativity and song generation: (1) *Application-wise*, songs are an interesting and challenging target for creativity, as they require the production of complementary music and lyrics. (2) *Technically*, we approach the problem of creativity and song generation using constraint programming. We show how constraints can be used declaratively to define a search space of songs so that a standard constraint solver can then be used to generate songs. (3) *Conceptually*, we describe a concrete architecture for transformational creativity where the creative (song writing) system has some responsibility for setting its own search space and goals. In the proposed architecture, a meta-level control component does this transparently by manipulating the constraints at runtime based on self-reflection of the system. Empirical experiments suggest the system is able to create songs according to its own taste.

ARTICLE HISTORY
Received 16 July 2017
Accepted 17 February 2018

KEYWORDS
Computational creativity; transformational creativity; music; lyrics; constraint programming

1. Introduction

Computational creativity is a subfield of artificial intelligence that studies and simulates creative behaviour by computational means. Colton and Wiggins (2012) have formulated a definition of computational creativity as "the philosophy, science and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem to be creative". Thus computationally creative systems should not only generate novel and interesting results but also have some creative responsibility, so that the produced results have qualities that cannot be traced back to the designer of the system. How to build a system that has such responsibilities is a key question in computational creativity research.

In this paper, we study this issue in the context of automated song writing. Conceptually, our framework is so-called transformational creativity. Boden (1998, 2004) identifies three general categories of (computational) creativity: combinational, exploratory and

CONTACT Hannu Toivonen hannu.toivonen@cs.helsinki.fi

Supplemental data for this article can be accessed at https://doi.org/10.1080/09540091.2018.1443320

transformational. These categories differ in the level of abstraction on which the system operates and in the types of creative responsibilities the system possesses.

- A system that exhibits *combinational creativity* operates on the concrete level of artefacts, putting together existing objects or ideas in novel ways. For instance, a combinational approach to generation of melodies could take fragments from existing songs and combine them to form a new one. Combinationally generated artefacts are composed of such familiar components.
- A system that exhibits *exploratory creativity* carries out search in some space of possible artefacts, using more abstract search mechanisms than just combinations or modifications of existing artefacts. For melodies, the search space could be defined by rules such as "notes must be on the chromatic scale" (the typical Western 12-note scale) and "notes must fall within a given range of pitches". An exploratory method is thus not limited to familiar components, but has the potential of generating artefacts consisting of novel components.
- Finally, a system that exhibits *transformational creativity* modifies the search space, i.e. it operates on a meta-level with respect to pure exploratory creativity. A transformational melody generation system could change its rules, e.g. modify the admissible range of pitches. In an extreme case this allows the system to generate artefacts it was not able to generate before, e.g. if the new range of pitches contains new notes.

In this work, we study the most complex of these, transformational creativity. For us, *the defining characteristic of transformational creativity is the system's ability to modify its own search space or goals*. We argue that this an important form of creative responsibility that a system can take.

We describe the technical architecture of an implemented system for generation of songs, i.e. pieces of matching music and lyrics. The proposed architecture aims for transformational creativity in the sense that it modifies its own search space and its own preferences within that search space. The design of the system is based on using a constraint program to specify the search space and allowing the system to manipulate the constraints to modify that space.

In a nutshell, the main components of our architecture are the following: (a) a generator of lyrics and music, effected by describing the task as a constraint satisfaction problem, allowing use of off-the-shelf constraint solvers to generate songs. The constraints include an interaction model for the correspondence between musical and linguistic features, allowing the complete problem of composing a coherent song to be modelled as a constraint satisfaction problem. (b) A set of musical feature extractors used to analyse the generated songs, in order to provide system-internal feedback on its own performance independent of the constraints. (c) A meta-level control layer, in which the system uses this system-internal feedback to modify its own constraints, making the system transformationally creative.

While this paper revolves around the transformationally creative architecture outlined above, it makes scientific contributions on three different levels of abstraction:

- *Application-wise*, songs are an interesting and challenging target for creativity, as they require the production of complementary music and lyrics. Instead of mimicking existing tastes, transformational creativity allows the system to develop some of its own.

- *Technically*, we approach the problem of creativity and song generation using rule-based constraint programming paradigm of answer set programming. We show how it can be used not only for the task of generation but also in implementing transformationally creative systems by manipulation of the constraints at runtime.
- *Conceptually*, we describe an implemented software architecture for transformational creativity where the creative (song writing) system takes creative responsibility by adapting its own search space and goals at runtime.

This paper is structured as follows. The context and background for this work is provided in Sections 2 and 3, where we review the theoretical basis of transformational creativity and the related work on the topics to which we contribute, respectively. The system and its architecture are described in Sections 4–6: first the ground-level generation of songs, including the use of constraint programming for the task (Section 4); then the ability of the system to assess its own productions, which is an integral part of creative autonomy (Section 5) and finally, the way the system exercises its creative autonomy and achieves transformational creativity by meta-level operations (Section 6). Results are presented and discussed in Sections 7–9. We first give some examples of generated songs so readers can form their subjective opinions of the quality of the results (Section 7). We then present results of a qualitative evaluation, where we assess both the quality of the songs and the transformationality of the system (Section 8). Finally, we discuss the work and the results and draw conclusions (Section 9).

2. Theoretical and conceptual setting

We use Wiggins' formalisation of creativity as search (Wiggins, 2006) as our theoretical framework to discuss creative systems and transformational creativity. Wiggins' framework covers all three types of creativity as characterised by Boden, as all of them can actually be seen as search. We give a brief overview of the formalism here, sufficient to discuss how our approach is creative; for the full formulation, see Wiggins (2006).

When viewing creativity as search, there are three central components that can be used to characterise a creative system. First, the search space of the system is defined by some set of rules $\mathcal{R}$. In the case of a song generation system, the search space consists of everything the system considers structurally valid as a song. Second, the value of artefacts (songs) is determined by an evaluation function $\mathcal{E}$, essentially telling if a given artefact is good or not. To find good elements in the space defined by $\mathcal{R}$, the system uses some traversal function $\mathcal{T}$ to perform the actual search.

This formulation has a direct correspondence to Boden's exploratory creativity. Combinational creativity, in turn, is characterised by a search function $\mathcal{T}$ that takes two existing artefacts and recombines them to produce a new one. The (reachable) search space is implicitly defined by the pool of available artefacts and the combination/modification operations available to the agent.

Transformational creativity can now be achieved by modifying the conceptual space (i.e. rules $\mathcal{R}$), the evaluation function $\mathcal{E}$ or the search function $\mathcal{T}$. By changing its own rules $\mathcal{R}$ and evaluation function $\mathcal{E}$, a creative system is able to adjust its goals and to exercise creative autonomy, while modification of the search function $\mathcal{T}$ may affect how effective and efficient the system is with respect to reaching its goals.

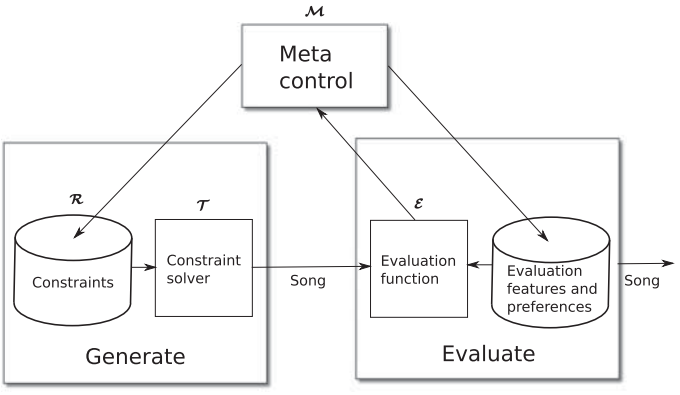

Figure 1. Generation and evaluation components of the system. Constraint programming allows the meta-level to manipulate the search space, specified by constraints to the constraint solver. The evaluation component is based on a pool of features and target values for them. These can also be modified from the meta-level.

Wiggins intended his framework as a conceptual tool, not as a software architecture (Wiggins, 2006). In this paper, however, we specifically construct an architecture where $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{T}$ have direct counterparts and where explicit representation of $\mathcal{R}$ and $\mathcal{E}$ allows their manipulation during runtime. A system's capability to adjust its own rules suggests that it also has a meta-level component which carries out such changes. We denote such a component explicitly by $\mathcal{M}$. (Wiggins has no separate notation for it since search on the meta-level can also be described by extending $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{T}$.)

In this paper we study, in the context of song generation, the question whether a practical, generic architecture for transformational creativity could be formulated. The architecture we propose is characterised by the following properties (Figure 1).

(1) We have a modular separation between
   (a) ground-level search space $\mathcal{R}$,
   (b) ground-level search $\mathcal{T}$,
   (c) ground-level evaluation $\mathcal{E}$ and
   (d) meta-level control module $\mathcal{M}$ capable of modifying ground-level search space and evaluation.
(2) Constraint programming is used for the implementation of ground-level generation (the Generate step of Figure 1). This allows declarative specification of the search space using a set $\mathcal{R}$ of constraints and application of an off-the-shelf constraint solver as the traversal function $\mathcal{T}$ to generate artefacts according to the given constraints.
(3) The evaluation function $\mathcal{E}$ considers of a variety of features and their target values (the Evaluate step of Figure 1). The internal architecture of the Evaluation step can be seen as an analog of the Generation step. They both have two subcomponents: a declarative specification that can be manipulated by the meta-level $\mathcal{M}$ and an operational component that implements the specification.
(4) The meta-level component $\mathcal{M}$ controls the generation and evaluation components. It receives evaluations of songs from the evaluation function $\mathcal{E}$, and based on them decides how to modify constraints $\mathcal{R}$ or the target values of the individual features evaluated by $\mathcal{E}$.

An interesting view to what can be considered creative is provided by Jennings (2010). He argues that the greatest challenge for computational creativity researchers is to show that their systems are not just extensions of their own creativity but instead capable of some *creative autonomy*. In his work, Jennings focuses on the system's creative responsibility in setting its own goal.

Jennings' criteria for creative autonomy are that (1) the agent is capable of independently evaluating artefacts it produces, and that (2) the agent can adjust its standards without explicit directions for when and how to do so. (Additionally, both should not be purely random.) The first criterion of creative autonomy can be mapped to the system having an evaluation function $\mathcal{E}$ enabling self-reflection, a crucial component of the notion of transformational creativity (Boden, 2004). The second condition corresponds, broadly taken, to the idea of transformationality and is in our case manifested in the meta-level component $\mathcal{M}$ adjusting $\mathcal{R}$ and $\mathcal{E}$.

Grace and Maher (2015), in turn, draw from design literature and argue that a cycle of intentionality and exploration is important for creativity: a creative system can be surprised also by its own output if it externalises and then re-perceives its creations. Surprises can then lead to specific curiosity, i.e. the agent intentionally directs its search and attention to areas close to the surprising artefact. Our work will utilise similar ideas for adjusting the system's goals.

3. Related work

We next briefly review related work in the three topics of this paper: (1) song generation, including the separate melody and lyrics generation, (2) use of constraint programming in creative systems and (3) architectures for transformational creativity.

3.1. Song generation

Automated music composition has received a lot of attention in the past (Roads, 1996). Many methods for generating different aspects of music have been developed, including various statistical methods (Simon, Morris, & Basu, 2008), L-systems (Prusinkiewicz, 1986), fractals (Sukumaran & Dheepa, 2003) and constraint satisfaction approaches (see next subsection for more information, Boenn, Brain, De Vos, & Ffitch, 2008). Automated music generation has also been rooted in many different kinds of seed information. For instance, music has been generated to match a target emotion (Monteith, Martinez, & Ventura, 2010) or sleep data measurements (Tulilaulu, Paalasmaa, Waris, & Toivonen, 2012).

Likewise, poetry generation, closely related to lyrics generation, has received a lot of attention in the field of computational creativity (Colton, Goodwin, & Veale, 2012; Gervás, 2001; Manurung, 2003; Toivanen, Toivonen, Valitutti, & Gross, 2012). Here, many different methods have been used as well. However, few systems combine music and lyrics. We next briefly review such systems.

One line of research generates music based on text. For instance, Monteith, Martinez, and Ventura (2012) have generated musical accompaniments for given lyrics. This approach concentrates on the extraction of linguistic stress patterns and composition of a melody with matching note lengths and fulfilment of certain aesthetic metrics for musical and

linguistic match. Monteith, Francisco, Martinez, Gervás, and Ventura (2011) have also generated musical accompaniments for stories, targeting emotive labels that match musical affect to emotional story content.

On the other hand, lyrics have been generated based on music. For instance, Oliveira, Cardoso, and Pereira (2007) have generated text based on rhythm, and Ramakrishnan, Kuppan, and Devi (2009) have generated lyrics for given melodies automatically.

Some systems combine music and lyrics in an interesting way but are not generative. For instance, Mihalcea and Strapparava (2012) use both music and lyrics to classify songs into different emotion categories.

Finally, there are a few existing systems that generate both the music and lyrics for songs. Toivanen, Toivonen, and Valitutti (2013) have generated simple songs by reducing the problem to a sequence of two tasks: first writing lyrics and then composing music to match the lyrics. Since both tasks are carried out by the same system, information from the lyrics writing process can be transferred to the music composition module, informing the latter of choices made, intended sentiment, etc. Sridhar, Gladis, Ganga, and Prabha (2014) generate lyrics in a number of ways, one of which is combined with also producing a melody. Here, the tune is composed first and then Tamil lyrics are generated from the tune. Scirea, Barros, Shaker, and Togelius (2015) have generated songs based on real world data, in particular academic papers. This system composes lyrics of the song by extracting important words from a given academic paper and using them in templates. Then the system composes a melody by using a Markov chain evolution method.

The method of this paper generates music and lyrics simultaneously. It makes them match by using declarative constraints that specify which lyrical and musical features should coincide and how. Since the aim is to produce songs as a whole, simultaneous generation of music and lyrics should produce better results than their sequential generation, which may result in partial songs (music or lyrics) which are very difficult to complete satisfactorily (with matching lyrics or music, respectively). We return to this in Section 9 .

3.2. Constraint programming in creative systems

Constraint logic programming is an efficient declarative programming paradigm for solving computational problems with many interacting components. The goal of the computation is described declaratively and the intermediary steps for reaching that goal are left unspecified. Finding the optimal solution can then be performed with a general-purpose constraint solver. In our work, we utilise Answer Set Programming (ASP), a programming paradigm rooted in logic programming and non-monotonic reasoning (Gelfond & Lifschitz, 1988; Niemelä, 1999; Simons, Niemelä, & Soininen, 2002).

In the area of computational creativity, constraint solvers have been used to generate both music and poems. However, we are not aware of any transformationally creative systems based on constraint programming.

Boenn et al. (2008) and Boenn, Brain, De Vos, and Ffitch (2011) have developed an extensive music composition system, Anton, using ASP to represent the musical knowledge and the rules of the system. Anton describes a model of musical composition as a collection of interacting constraints. The system can be used to compose short pieces of music as well as to assist the composer with suggestions, completions and verification. For other examples

of music composition using constraint programming, see Anders' survey (Anders, 2011) on the subject.

In recent years, many different systems for producing poetry with constraint satisfaction methods have been developed. For instance, Toivanen, Järvisalo, and Toivonen (2013) have developed a system that utilises ASP to automatically produce poems that satisfy various poetic and linguistic features. Tobing and Manurung (2015) have used logic programming to generate topical metrical poetry, and El Bolock and Abdennadher (2015) generate poetry with constraint handling rules.

In a different line of work, Papadopoulos, Pachet, Roy, and Sakellariou (2015) suggest methods combining regular and Markov constraints for imitating the style of a musical or textual corpus. These methods can efficiently sample sequences that satisfy the regular constraints and obey the distribution defined by the probabilistic Markov constraints. Such methods could potentially be used in a transformational architecture similar to ours, where a meta-level component can control the constraints under which the methods then operate, but to our knowledge this has not been done.

In contrast to the work reviewed above, in this paper we propose that constraint programming is a paradigm that lends itself to transformational creativity: declarative constraints are relatively easy to manipulate for the system itself, while the generation step (finding a solution) can be left to an off-the-shelf constraint solver.

3.3. Transformationally creative systems

Early examples of transformational systems in the field of artificial intelligence include Lenat's Automated Mathematician and Eurisko (Lenat, 1983; Lenat & Brown, 1984). They both are discovery systems consisting of heuristic rules, including heuristics about changing their own heuristics. They are clearly transformational in the sense of Boden (2004) and Wiggins (2006) as they manipulate their own search space. Automated Mathematician and Eurisko have inspired several successors; for instance, the HR system (Colton, 2001; Colton, Bundy, & Walsh, 1999) automatically forms mathematical theories using a higher level theory that contains concepts and conjectures about the lower level concepts and conjectures.

Transformational creativity is foundational in computational creativity research, and numerous systems can be argued to be transformational in one sense or another. However, even though well-known examples of transformational architectures exist, e.g. in the discovery systems mentioned above, we are not aware of implemented transformational architectures for computational creativity. We are here interested in explicit representations of the search space (essentially rules $\mathcal{R}$) and of the evaluation function $\mathcal{E}$ and of their manipulation by the system itself. In this paper, we propose such an architecture for transformational creativity, using song writing as the example application.

Ritchie (2006) discusses various possible meanings of transformational creativity in depth. He also mentions constraints as a means to formalise the conceptual space of a creative system in a manner that affords changes in that conceptual space, i.e. potential transformational creativity. Ritchie, however, is primarily interested in empirical evaluation and observation of transformational creativity, based on generated artefacts. Our evaluation goes slightly in this direction, but our emphasis is on describing an implemented architecture that allows the system to transform its own internal conceptual space.

4. Ground-level generation of songs

We now move on to the architecture of our system. In this section, we describe how the system generates songs, and in Section 5 we outline the component that evaluates them. The meta-level component is then described in Section 6.

A central challenge in computational song writing is to produce a coherent, matching pair of music and lyrics – a problem with many interacting components and properties and thus well-suited for modelling as a declarative ASP program. In the present system, ASP is used to model the process of composing the song melody and lyrics together.

In this section, we give an overview of our song generation component in general and describe the use of constraint programming for composing songs with matching musical and textual features in some more detail, as an enabler of transformational creativity in our architecture. In our implementation, the overall song structure, chord progressions and lyrical phrase candidates are generated procedurally, while the melody and lyrics are composed using ASP under the procedurally generated constraints. The procedural parts are not linked to transformational creativity in this paper, so they are intentionally kept simple.

4.1. Overview of song generation

The system aims to produce songs that are easy to sing and play. In addition, the results should be versatile and aesthetically pleasant to human listeners. More importantly, however, the system should be able to modify its preferences and to direct its attention to those subspaces of possible songs it finds interesting. The viewpoint here is purely technical; emulation of the process used by humans to write and to perceive songs is not in the scope of this work.

The system produces simple songs with a fixed number of bars and a simple overall structure. The output consists of music notation with the melody, lyrics and chords for accompaniment. Chords are presented with chord symbols, i.e. the system does not produce full-scale accompaniments. For output, we use Lilypond notation (Nienhuys & Nieuwenhuizen, 2003) that can be easily converted into pdf sheet music or transformed automatically into audio with MIDI synthesis if needed.

A song is generated in three phases, gradually adding detail (Figure 2). First, an overall section structure (e.g. ABAB) is selected. Then chord progressions are generated for each section type (and repeated in all sections of the same type). Finally, melody and lyrics are generated using constraint programming, in two steps: generation of so-called candidate data consisting of possible notes and segments of lyrics, and application of ASP to compose songs out of this data using the current constraints.

In the next subsections, we first describe elements produced procedurally (section structure, harmony, segments of lyrics) and then elements produced with constraint programming (melody, lyrics). Our focus will be on the latter.

4.2. Procedural generation of section structure, harmony and lyrical phrase candidates

Section structure, harmony and lyrical phrase candidates are generated with simple methods and they are not central to our research questions. We here briefly outline the generation methods used.

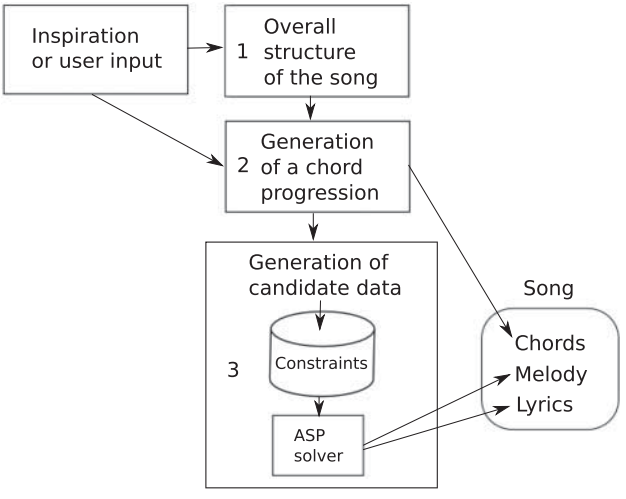

Figure 2. Ground-level song composition architecture (see Section 4.3 for constraint programming).

Section structure. The system uses simple section structures, such as ABBA or AABB, where letters A and B denote different musical sections. Their main function here is to provide musical coherence to the songs by means of repetition. In the current setting, the structural simplicity makes it easier to concentrate on the interaction of musical and textual features in the ASP program (and their transformationality). More elaborate musical structures and detailed information of the qualities of conventional musical elements, such as introduction, verse, pre-chorus and chorus, are outside the scope of this work but could be incorporated into the system to improve the results.

Harmony. The harmony is composed either in a major or minor scale based on user input or system's own choice. The system database contains different sets of harmonic patterns regularly found in diatonic Western music for major and minor keys. The construction of harmony is based on selecting harmonic patterns from this set and expressing these as chord sequences in a given key. A typical harmonic pattern is, for instance, the chord sequence I, IV, V. When dealing with minor keys, harmonic minor scale is used. In the end of the music sections, the last chord is constrained to be the first scale degree chord in order to provide harmonic resolution. The harmony generation procedure also assigns time values for each of the chords in a probabilistic manner. The chord progressions could easily be learned from a corpus to be imitated.

Lyrical phrases. The constraint program composes the lyrics of the song from a set of lyrical phrase candidates. We here describe how the phrase candidates are generated procedurally.

Initial phrases are first extracted automatically from text corpora based on specific patterns, such as being a full sentence, having a trochee meter (a metrical foot which consists of a stressed syllable followed by an unstressed syllable), or starting with the word "I". These phrases are then modified in order to provide more variability and novelty to the results. The modification is carried out by substituting words in the original phrases. In the word substitution process, the goal is to roughly keep the original meaning of the phrase by

replacing, for instance, nouns or verbs in the phrases by nouns or verbs that occur in similar contexts.

The word substitution process is based on mining word co-occurrences from large text corpora and utilising word associations of two kinds. *Syntagmatic relations* refer to words that co-occur in the text. For example, the phrase "*yellow moon*" forms a syntagmatic expression. On the other hand, a *paradigmatic relation* refers to words that occur in the same context and can be technically substituted with each other. For instance, "*moon*" is paradigmatic to "*sun*" because these words can be used in similar contexts. Word substitution process preserves the original patterns of the phrases.

For coherence of the lyrics, a set of phrases is selected as follows. One phrase is first selected at random. Then a collection of semantically most similar phrases with respect to that seed phrase is selected. As a measure of similarity between two phrases, we utilise a modified Hausdorff distance (Dubuisson & Jain, 1994), where phrase similarity is the average distance over shortest word distances between phrases. For measuring the distance between two words, we utilise the Word2Vec method (Mikolov, Chen, Corrado, & Dean, 2013) trained with paradigmatic word similarity.

The set of phrases generated in the above manner is given to the constraint solver that selects and combines the phrases based on all the constraints defined for the music and lyrics.

4.3. Constraint-based generation of melody and lyrics

We continue by describing our constraint satisfaction formulation for generating melody and lyrics. Solutions to the constraint formulation, found by calling an off-the-shelf constraint solver, correspond to pieces of melody and lyrics that "match" together, as specified by the constraints.

Answer set programming. As the constraint programming paradigm of choice, we use answer set programming (ASP). While we here rely on ASP due to the high-level, data-driven declarative language it offers, together with relatively efficient optimisation solver technology available for the language, we acknowledge that other constraint optimisation paradigms could be alternatively applied.

Answer set programming (Gelfond & Lifschitz, 1988; Niemelä, 1999; Simons et al., 2002) is a rule-based, data-driven constraint satisfaction paradigm. The conceptual separation between rules and data is important for understanding the approach. *Rules* are the generic part of the constraint program, used for any problem instance, while each specific instance is represented by *input data* given to the rules. In our case, the procedurally generated parts described in the previous subsections serve to specify a problem instance. This instance specification includes candidate phrases for lyrics and candidate notes generated based on the harmonic progression.

In ASP, input data is represented as first-order predicates that express the problem instance; the constraint declaration (the actual ASP program) is expressed in terms of first-order rules over the input predicates. The rules formalise how additional knowledge can be inferred from the input data, to be expressed via output predicates, and the rules also define constraints over the solutions of interest. In our system, input predicates express basic properties of the form of the composed songs, and the computational problem of generating a song is expressed via the generic rule-based constraint declaration.

After the generic constraint declaration is designed, state-of-the-art generic ASP solvers, such as Clingo (Gebser, Kaminski, Kaufmann, & Schaub, 2014), provide an efficient way of finding solutions to problem instances (different input data) under the constraint declaration. Such solvers are *complete*: given enough computational resources they are guaranteed to find the solutions to any input given that solutions exist or to prove that no solutions exist in the negative case.

We will not provide formal details on answer set programming and its underlying semantics; the interested reader is referred to other sources (Gelfond & Lifschitz, 1988; Niemelä, 1999; Simons et al., 2002) for a detailed account. Instead, we will in the following provide a step-by-step intuitive explanation on how the task of song generation can be expressed in the language of ASP. For more hands-on examples on how to express different computational problems in ASP, we refer the interested reader to Gebser, Kaminski, Kaufmann, and Schaub (2012).

Input predicates. Below, we briefly describe the central input predicates used to specify the input data to the ASP program. Following the data-driven view, the idea is that possible modifications to the generation process are controlled by changing the input via changing the parameters of the input predicates.

- `bars/1`: `bars`(*b*) represents the fact that a song consists of *b* bars.
- `units/1`: `units`(*u*) represents the fact that each bar in a song consists of *u* atomic units.
- `note_candidate/5`: `note_candidate`(*i*,*j*,*p*,*d*,*c*) represents the fact that a note with pitch *p*, duration *d* units and consonance *c* is a note candidate at unit *j* of bar *i*, i.e. such a note may be selected to begin at unit *j* of bar *i* in a song.
- `phrase_candidate/2`: `phrase_candidate`(*p*,*b*) represents the fact that phrase *p*, spanning *b* bars, is a phrase candidate for the lyrics of the song.
- `syllable/4`: `syllable`(*p*,*i*,*s*,*b*) represents the fact that the *i*th syllable of phrase candidate *p* is *s* and has stress $b \in \{0, 1\}$ (0 =unstressed, 1 =stressed).
- `phrase_should_start_at/1`: `phrase_should_start_at`(*b*) represents the fact that a phrase should start at the beginning of the *b*th bar.
- `interval/2`: `interval`(*l*,*u*) represents the fact that the sum of intervals of the melody, i.e. the sum of differences of pitches between consecutive notes, should be at least *l* and at most *u*.
- `max_duration_unstressed/1`: `max_duration_unstressed`(*d*) represents the fact that each unstressed note should have a maximum duration of *d* units.
- `min_duration_stressed/1`: `min_duration_stressed`(*d*) represents the fact that each stressed note should have a minimum duration of *d* units.
- `consonance/2`: `consonance`(*l*,*u*) represents the fact that the total consonance, i.e. the sum of the consonances of notes, should be at least *l* and at most *u*.

Since the approach is declarative, the set of input predicates can be easily extended with further predicates to obtain even tighter control over the generation process.

Output predicates. Next we describe the output predicates of the ASP program. The ASP rules specified in the following subsections use these predicates to encode the generated melody and lyrics.

- `phrase_starts_at/2`: `phrase_starts_at`(*p*,*b*) represents the fact that phrase *p* begins at (the beginning of) bar *b*. The ASP program rules will enforce that songs with overlapping phrases are not generated.
- `syllable_at/4` provides the syllable positions of the selected phrases, `syllable_at`(*p*,*i*,*b*,*u*) representing the fact that the *i*th syllable of phrase *p* takes place at unit *u* of bar *b* of a song.
- `pluck/5`: `pluck_at`(*b*,*u*,*p*,*d*,*c*) represents the fact that a note with pitch *p*, duration *d* and consonance *c* takes places at unit *u* of bar *b* in a song.

Projections. We start the description of the ASP program rules with useful projections of the input predicates (Figure 3) to be used as auxiliary predicates in rules described later on. Rule `r1` infers the set of pitches available in the note candidates, represented via the `pitch/1` predicate. Similarly, Rules `r2`—`r5` infer the set of note consonances, bars, units and phrases available in the input data.

Constraint declaration for generating lyrics. We now continue with detailing the rules that form the central computational problem at hand, starting with phrase selection (Figure 4). Rule `r6` expresses that for each bar from which a phrase should start (as specified via the input predicate `phrase_should_start_at/1`), exactly one phrase from the set of phrase candidates (as specified via the input predicate `phrase_candidate/1`) should be selected to start at the beginning of the bar. The phrase selected to start at the position is represented via the output predicate `phrase_starts_at/2`. In order to prevent overlapping phrases to be selected, Rule `r7` infers the bars occupied by each selected phrase, expressed via the auxiliary predicate `phrase_at/2`. Using this knowledge, Rule `r8` enforces that `phrase_at/2` should be true at each bar for exactly one phrase candidate.

In our formulation, the atomic elements of phrases are syllables, and they will be aligned with notes of the melody. Rules for inferring the positions of syllables are given in Figure 5. Rule `r9` declares for each selected phrase that each syllable in the phrase must be positioned at exactly one position within the bars that the phrase occupies. Rules `r10` and `r11` enforce that the syllables are positioned in the right order, following the phrase. Rule `r10`

```
(r1)  pitch(P)      :- note_candidate(_,_,P,_,_).
(r2)  consonance(C) :- note_candidate(_,_,_,_,C).
(r3)  bar(I)        :- bars(B), I=1..B.
(r4)  unit(J)       :- units(U), J=1..U.
(r5)  phrase(P)     :- phrase_candidate(P,_).
```

Figure 3. Useful projections of the input predicates.

```
(r6)  1 { phrase_starts_at(I,P) : phrase_candidate(P) } 1 :- phrase_should_start_at(I).
(r7)  phrase_at(I+X,P) :- phrase_starts_at(I,P), phrase_candidate(P,B),
                          bar(I), X=0..B-1.
(r8)  1 { phrase_at(I,P) : phrase_candidate(P) } 1 :- bar(I).
```

Figure 4. Selecting phrases.

```
(r9)  1 { syllable_at(P,I,B+BI,JI) : BI=0..L-1 : JI=U : unit(U) } 1 :-
              syllable(P,I,_,_), phrase_starts_at(B,P), phrase_candidate(P,L).

(r10) :- syllable_at(P,I,B,J1), syllable_at(P,I+1,B,J2), J1>=J2,
         phrase_at(B,P).
(r11) :- syllable_at(P,I,B1,_), syllable_at(P,I+1,B2,_), B1>B2,
         phrase_at(B1,P), phrase_at(B2,P).
```

Figure 5. Selecting syllable positions.

```
(r12) pluck_at(B,J) :- syllable_at(_,_,B,J).
(r13) pluck_duration(B,J1,D) :- syllable_at(P,I,B,J1), syllable_at(P,I+1,B,J2),
                                D=J2-J1.
(r14) pluck_duration(B,J1,D) :- syllable_at(P,I,B,J1),
                                0 { syllable_at(P,I+1,B,J2) : J2>J1 } 0,
                                D=U-J1+1, units(U).

(r15) 1 { pluck(I,J,P,D,C) : note_candidate(I,J,P,D,C) } 1 :-
                                pluck_at(I,J), pluck_duration(I,J,D).
```

Figure 6. Matching melody and lyrics.

enforces that any two consecutive syllables in a phrase, if positioned to a same bar, must be ordered according to the phrase. Rule `r11` declares analogously that, if two consecutive syllables are not positioned in the same bar, the latter syllables must not be positioned to a bar before the bar to which the former syllable is positioned.

Constraints for matching lyrics and notes. A key part of the constraint formulation are constraints which align syllables of the lyrics with notes of the melody. They ensure that for each syllable, a note occurs at the same position, and *vice versa*. In our formulation, the positions and lengths of notes are inferred from the positions of syllable and distances between them, respectively (Figure 6). Rule `r12` ensures that a note will take place exactly when a syllable occurs. Rules `r13`–`r14` infer the lengths of notes by computing the distance to the syllable that occurs next within the song, which may be either within the same bar (Rule `r13`) or in the following bar (Rule `r14`). Finally, Rule `r15` enforces that exactly one concrete note (with a specific pitch and consonance, and of the correct duration) is selected from the set of note candidates given as input.

Constraint declarations for controlling melody. We move on to describing further constraints which allow for obtaining tighter control of properties of the generated songs.

As a first example, we consider controlling the matching of stressed (unstressed) syllables with longer (shorter) notes (Figure 7). To this end, Rule `r16` enforces that the duration of a node associated to a stressed (stress = 1) syllable should not be below the value controlled via the input predicate `min_duration_stressed`. Rule `r17` analogously enforces a maximum duration for notes associated with unstressed syllables. Finally, Rule `r18` ensures that syllables declared as unstressed in the input data should not be considered stressed.

Rules `r19`–`r21` in Figure 8 enforce a global constraint on the consonance of the song. The consonance of each selected note is represented via the auxiliary predicate `pluck_consonance` and inferred by Rule `r19`. Rules `r20` and `r21` then enforce a

```
(r16) :- syllable_at(P,I,B,J), syllable(P,I,S,1), pluck_duration(B,J,L),
         L < min_duration_stressed(D).
(r17) :- syllable_at(P,I,B,J), syllable(P,I,S,0), pluck_duration(B,J,L),
         L > max_duration_unstressed(D).

(r18) :- syllable(P,I,_,0), syllable_at(P,I,B,1).
```

Figure 7. Controlling stress.

```
(r19) pluck_consonance(I,J,C) :- pluck(I,J,_,_,C).
(r20) :- #sum [ pluck_consonance(I,J,C) = C ] L-1, consonance(L,_),  bar(I).
(r21) :- U+1 #sum [ pluck_consonance(I,J,C) = C ], consonance(_,U),  bar(I).
```

Figure 8. Controlling note consonance.

```
(r22) sound(B,J+L,P) :- pluck(I,J,P,_,_), pluck_duration(B,J,D), L=0..D-1.
(r23) sound(B,J) :- sound(B,J,_).
(r24) :- not sound(B,J), bar(B), unit(J).
```

Figure 9. Rules for enforcing a continuous melody.

```
(r25) pitch_change(I,J,P) :- sound(I,J,P1), sound(I,J-1,P2),  P=#abs(P1-P2).
(r26) pitch_change(I,1,P) :- sound(I,1,P1), sound(I-1,16,P2), P=#abs(P1-P2).
(r27) :-     #sum [ pitch_change(I,J,P) : bar(I) = P ] L-1,   interval(L,_).
(r28) :- U+1 #sum [ pitch_change(I,J,P) : bar(I) = P ],       interval(_,U).
```

Figure 10. Controlling changes in note pitch.

global lower and upper bound of song consonance, here considered as the sum of the consonances of individual notes in the generated melody.

In order to enforce a continuous melody as well as to obtain global control of pitch changes within the melody, Rules `r22` and `r23` infer the positions at which some note is sounding, represented via the auxiliary predicate `sound/2`, from the actual notes and their durations. Rule `r24` then enforces a continuous melody, declaring that there should not be a bar and a unit at which there is no sound.

Finally, Rules `r25`–`r28` in Figure 10 control the global cumulative absolute pitch change within a generated melody. Rules `r25` and `r26` infer the absolute difference between pitches of consecutive notes, should they occur in the same bar (Rule `r25`) or consecutive bars (Rule `r26`), represented via the auxiliary predicate `pitch_change/3`. Rules `r27` and `r28` then enforce that the cumulative sum of pitch changes in the whole melody should be bounded according to the lower and upper bounds provided as input via the `interval/2` input predicate.

Together, the rules in Figures 3–10 constitute the ASP program that produces songs when provided with the procedurally generated input data described earlier in this section.

5. Ground-level evaluation of songs

Each generated song is passed to the evaluation function $\mathcal{E}$ (cf. Figure 1) which makes an aesthetic judgement of the song using a number of musical and lyrical features. These features are normalised to compute values in the range [0, 1] and are selected from a pool of features that cover a range of musical and lyrical characteristics (largely adopted from Murray and Ventura, 2012).

For example, the percentage of melody notes that are in the *j*th key is given by feature

$$\text{KeyPrevalence}_j(I) = \frac{|K_j|}{|I|}, \tag{1}$$

where I is the sequence of notes in the melody, $K_j = \{i \in I \mid i \in Key_j\}$ and Key_j is the set of notes in the *j*th key.[1] An example feature for lyrics is

$$\text{PositiveLyricsSentiment(W)} = \frac{1}{|W|} \sum_{w \in W} \sigma(w), \tag{2}$$

which computes the average positive sentiment over individual words w in given lyrics W.

The other features in our system are based on self-similarity of the melody and of the rhythm, on linearity and pitch range of the melody, on interval class prevalences, on variability of rhythm, on average duration of a note, on variability of harmony, on phoneme structure of the lyrics, on rarity of the words in the lyrics, on correspondence of note duration to syllable stress and on average consonance value of melody notes with respect to the song chords. Appendix gives definitions of all features implemented in the system.

Given these features, an aesthetic value could be assigned in many ways. Our system uses a linear combination $\sum_i \alpha_i f_i$, where f_i are the features and α_i are non-negative coefficients (weights) such that $\sum_i \alpha_i = 1$. A feature i with weight $\alpha_i = 0$ is ignored, and the evaluation focuses on those features with (high) weights. For example, given a sequence of notes I representing the melody, a simple and highly focused song aesthetic is

$$\mathcal{E}(I) = \tfrac{2}{5}\text{KeyPrevalence}_1(I) + \tfrac{3}{5}(1 - \text{SelfSimilarMelody}(I)). \tag{3}$$

This aesthetic only considers two features: all other features have zero weights and are excluded for clarity. When the system now aims to maximise $\mathcal{E}$, it effectively values songs that are in G Major, that are not melodically self-similar (i.e. they have complex melodies), and where melodic similarity is somewhat more important than the key.[2]

In this way, the system can represent a large variety of musical aesthetics by linear combinations of subsets of musically meaningful features. As a result, the system has the potential to change its aesthetic, which has been identified as a necessary condition for autonomy in a computationally creative system (Jennings, 2010). At the same time, new aesthetics developed by the system will be defensible (at least at some level, even by the system itself) because they are always composed as combinations of musically meaningful features. This level of system autonomy is unusual in state-of-the-art computational creativity systems. It also supports transformational creativity, as will be discussed in Section 6.

The goal here is not to justify a particular musical aesthetic, which would be difficult under any circumstances. Rather, the goal is to provide the system with autonomy to explore not only the space of possible songs but also a space of (arguably justifiable) musical aesthetic eventually necessary for any autonomous creative system. In future work, it

would be interesting to have the system not only explore this space (which we believe is already a significant advance compared to current music creation systems) but also be able to provide framing information to justify its decisions.

6. Meta-level control

The previous sections explained how songs are generated using constraint programming (see Section 4) and outlined the evaluation process (see Section 5). We now move on to discuss the system's meta-level control $\mathcal{M}$ over song generation and evaluation, i.e. the transformational aspects of the architecture.

On an abstract level, the constraints $\mathcal{R}$ define the search space while the evaluation function $\mathcal{E}$ assesses the quality of songs found in the search space; the meta-level component $\mathcal{M}$ receives evaluations and then decides how to modify the constraints as well as the targets for evaluation (cf. Figure 1). We outline below the interactions of the meta-level with the other components: use of evaluations of songs, setting target values for features of songs and modification of constraints. The techniques are intentionally kept relatively simple so that the internal working of the meta-level component is transparent and also easy to adjust if needed. More elaborate optimisation techniques could be employed for higher efficiency; these are outside the scope of this paper.

Overview. The system works iteratively: it generates a song or several songs, evaluates them, adjusts its goal and constraints, then generates new songs using the new constraints and the new goal, evaluates them, etc. The goal, specified using features and their target values, is adjusted in each iteration depending on the results achieved; the changes in the constraints aim to help the system better reach its new goal.

The operation is initialised by generating a single song using a default setup of the constraints. Since song generation is stochastic, different runs of the system will produce different songs. The first song will then be used as the seed for setting the goal of the system. This can be compared to a human who starts composing music by humming a tune, apparently randomly but actually strongly influenced by all the background information and knowledge she has, and then works from this initial tune towards better and more interesting ones. However, our system does not evolve a single song during its process, it rather works on the level of characteristics of songs.

Evaluation of songs and setting the target. Each song is evaluated using a pool of features as described in Section 5 and the appendix. The result of the evaluation on all of the features is transmitted to the meta-level component.

For the first song produced (Algorithm 1, Line 2), the meta-level component analyses the feature vector and looks for combinations of features that appear specific to the song (Line 4). It considers features which have extreme values in the song, i.e. values close to 0 or 1, and picks a small set of such features to constitute the goal for song generation, with the aim of further emphasising the already extreme values in the next generation of songs. This operation can be seen as specific curiosity (Grace & Maher, 2015) that makes the system adapt its search space (see below for details).

In later iterations, the system already has a specific goal in mind. Songs produced by different constraint configurations (Lines 6–8) are evaluated with respect to the goal (Line 10), and the configuration that produces best results is selected for the next iteration

(Lines 12–13). The meta-level component also studies the best individual song among the ones produced by the best configuration (Line 14). If, despite the goal-oriented selection of the song, the song actually exhibits a combination of extreme features different from the goal, then the goal is adjusted to reflect those features (Line 4). This decision reflects the system's capabilities in reaching its own goal – if it finds a goal too difficult to reach, it chooses a related goal that looks more feasible.

Algorithm 1 Outline of the meta-level algorithm

```
1:  R ← default constraint configuration
2:  generate one song s ← GenerateSong(R)
3:  repeat
4:      F ← set of features that have extreme values in song s
5:      for i = 1 to n do
6:          randomly generate a variant R_i of the current constraint configuration R
7:          for j = 1 to m do
8:              generate song s_j ← GenerateSong(R_i)
9:          end for
10:         compute averages of the features in F over set S_i = {s_j | 1 ≤ j ≤ m} of songs
11:     end for
12:     select the constraint configuration R_i that produced best results w.r.t. F on average
13:     R ← R_i
14:     select s ← the best song in S_i w.r.t. F
15: until stopping condition
16: return song s
```

Modification of constraints. After the system has set its goal (in terms of features which should have extreme values), it aims to produce a new set of songs that go in the direction of the goal. This is achieved by modifying the constraints and their parameters (Line 6). Few of the constraints are directly related to the features measured in songs, so direct optimisation is not feasible. Instead, we resort to simple stochastic optimisation, randomly producing a number of variants of the current constraint configuration, i.e. constraint satisfaction problems.

The constraints are modified in several different ways. The simplest form of modification is changing parameters of the constraints, e.g. the required consonance values for the melody notes with respect to the underlying chords. Also, some constraints can be ignored altogether. For instance, in some songs a constraint might rule all the melody notes to belong to a certain musical key whereas in some other songs this rule can be left out.

Next, all the generated constraint configurations are fed to the constraint solver (Line 8). If the corresponding constraint satisfaction problem is satisfiable, a set of new songs is produced using it, and the songs are evaluated (Line 10). The constraint configuration that has improved the results most with respect to the chosen features (Line 12) is then selected as the basis for the next generation of constraint configurations (Line 13).

7. Examples

We have implemented a system according to the description in Sections 4–6 . Some practical choices need to be made when applying it; we here give the specifics. In the rest of this section, we then give example songs produced by the system. Some example songs with midi synthesis in mp3 format and respective music sheets can be found as electronic supplemental material (see footnote on the first page).

In experiments for this paper, we used the following settings. (1) In ground-level generation, the pitch range was constrained to two octaves in order to maintain singability of the songs. For the same reason, note duration was always at least a 16th note. The song melody and lyrics were composed in chunks of four bars in order to limit the size of the constraint problem. (2) In the meta-level algorithm (cf. Algorithm 1), a constant number of three features were selected to be maximised and another three features to be minimised, i.e. the number of features in set F was always 6. This method was selected in order to have several different features to be optimised but also to keep the meta-level optimisation process simple. In each iteration of the algorithm, five different constraint configurations were considered ($n = 5$), and with each constraint configuration five songs were generated ($m = 5$). Finally, the number of iterations of the main algorithm was two. These values were selected in order to allow monitoring of all steps and choices, as well as to keep the running time of the algorithm in reasonable limits (few days with a desktop computer having an Intel E8500, 3.16 MHz CPU with a 6MB L2 cache and 4 GM of RAM).

Example songs produced by one run of the system can be seen in Figures 11 and 12. The initial song in Figure 11 was evaluated as relatively high in the following features: note pitch consonance (with respect to the underlying chord), pitch range of the melody and usage of the prime interval. In contrast, the song was evaluated as relatively low in the following features: prevalence of a fifth interval, prevalence of a major ninth interval and prevalence of the note B. These six most extreme features were selected by the system to guide the next iteration of song creation: it set itself the goal of further strengthening the three high-scoring features and further weakening the low-scoring ones.

The best song from the next iteration of the system run can be seen in Figure 12. In this song, five of the six features were improved; the system was unsuccessful only in its

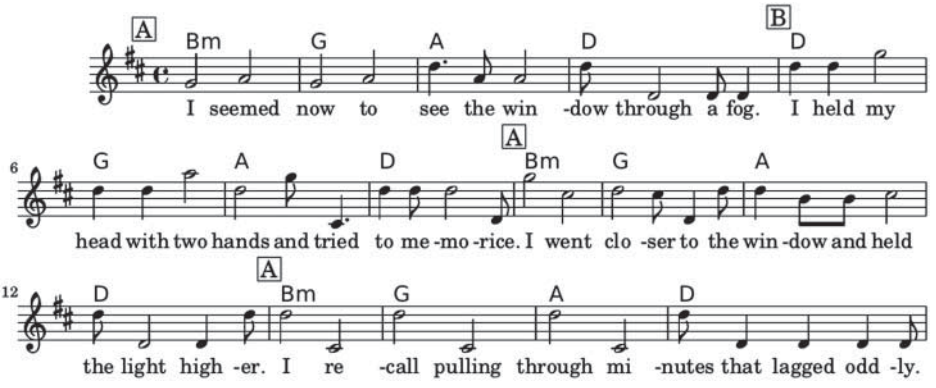

Figure 11. An example initial song generated by the system. A synthesised midi version in mp3 format can be listened to online (song number 6 in the electronic supplemental material).

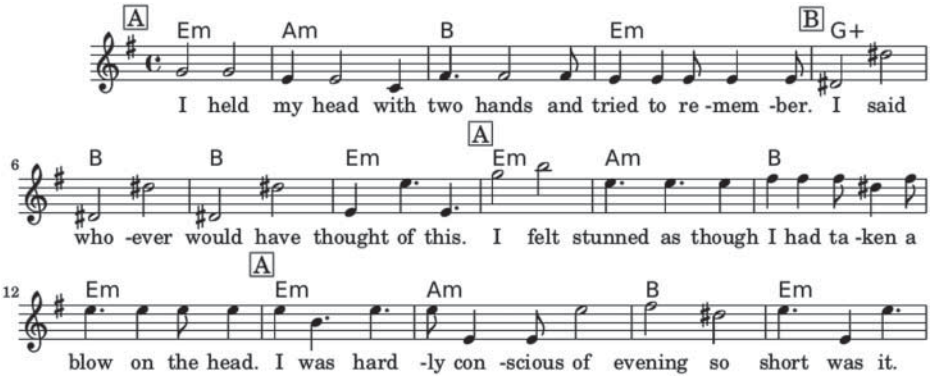

Figure 12. Example of a song generated using the specific goal to further strengthen features of the song of Figure 11: to increase the consonance of the melody notes w.r.t. the underlying chord, the width of the pitch range, and the use of the prime interval, and to reduce the prevalence of the fifth and major ninth intervals and the note B. A synthesised midi in mp3 format can be listened to online (song number 7 in the electronic supplemental material).

attempt to further reduce the prevalence of note B. For the next iteration, the system could then adjust its goal, possibly selecting another extreme feature of the song to replace the subgoal of minimising the number of Bs. The example illustrates how the system decides on its own goal, tries to reach it and adjusts the goal if it seems too challenging.

In our experiments, the song generation process usually converged in few iterations in the sense that the system did not improve the results anymore with respect to its goal. However, its constant modification of the constraints guarantees that it keeps on exploring different spaces of songs as long as it is allowed to run.

8. Empirical evaluation

The evaluation of computationally creative systems is inherently difficult (Jordanous, 2009). The appreciation of the songs is subjective, even emotional, it is context dependent and can change with the mood of the listener. Our focus is especially on potentially creative properties of the system, not so much on creativity of the results. Analysis of the results alone can be misleading since impressive results could be produced by trivial, non-creative systems: a system that has been provided with 100 highly creative songs by its developer could appear very creative by just randomly outputting one of them at a time. Hence, for computational creativity research, creativity of the generative *processes* is of special importance, in addition to the quality of the generated results (Colton, 2008).

We evaluate the song-writing system empirically with qualitative methods, using them to answer the following questions:

(1) *Are the songs produced by the system of good quality (recognisable as songs, singable, even fun)?* This is an elementary requirement for a system creating songs.
(2) *Is the system able to produce a range of novel songs?* Since the system sets its own goal at runtime, we are particularly interested to evaluate if the songs produced in different

runs are novel with respect to other runs, i.e. if the system develops different "tastes" at different runs.

(3) *Is the dynamic adjustment of the system's goal observable in the results?* Here we look at results during one run of the system, asking if the transformation of the goal is observable to the audience.

In the rest of this section, we address these questions with a qualitative user study where five people (two males and three females, mean age 28.0 years, all native Finnish speakers) assessed examples produced by the system with the parameters given in Section 7. In these experiments, we ran the system for two full iterations after the generation of the initial song.

In the evaluation, the subjects listened to midi synthesised versions of the songs. They were also provided sheet music with lyrics. All of the subjects evaluating the songs had years of experience and formal training in music theory and playing some instrument. However, most of them were not professional musicians.

Quality of the produced songs. Subjects were asked to evaluate the musical qualities, textual qualities and singability of the songs with open-ended questions. The songs for the evaluation were produced with varying constraint parameters and with lyrics in Finnish. In the first part of the tests, each subject evaluated three different songs. The subjects were asked (1) what the song sounds like, (2) what the song text looks like and (3) how singable the song is.

The songs were described with adjectives ranging from fun, surprising, interesting, and dramatic to monotonous and peculiar. Some subjects said that the songs are singable while others were more critical. For instance, in some songs the high variance in note durations was criticised, and so were short notes in some ends of phrases. Overall, the conclusion from the feedback is that the products of the system are recognisable as songs, they are singable (with some reservations) and at best they are even fun and interesting.

Variability of the generated songs. Creativity is often characterised as the ability to produce novel and valuable results. Above, we already addressed quality as a proxy for value. Now, as a proxy for novelty, we consider variation between songs produced by the system. (This also relates to transformationality, as the system sets its own objective during runtime and therefore aims to different kinds of results each time.)

In the second part of the evaluation, the subjects were asked if and how the previous three songs, produced by different runs of the system, are different. The general consensus of the subjects was that there is considerable variation in the music and lyrics of the produced songs. The most notable differences were seen in the building and release of tension in the music, as well as in the rhythm of the melody. We can thus conclude that the generated songs have novelty not only in the sense of being new songs but also having a good degree of variability between the songs generated.

Visibility of transformationality in the results. In the third part of the user study, we asked the subjects to take a look at two different pairs of sequentially produced songs where the second song has been produced to maximise features that already were extreme in the first song.

With these questions we wished to find out whether the differences between the two songs were also observable to humans. We addressed this first by asking the subjects to

report differences between the two songs, without any indication of the generation process and its goal. A positive result is that the consensus of the subjects was that there were notable differences between the two songs in both song pairs.

We then listed the system's subgoals (features to be maximised/ minimised) when generating the second song and asked how well the system had managed to fulfil these specific goals. The subjects reported that the system had managed to fulfil its internal goals well, i.e. the shift of the goals in song production was visible in the results. However, the subjects paid attention to the low-level nature of the features and noted that perhaps it could be more interesting to have more abstract, higher level features such as moods and emotions that would be composed of many low-level ones.

In summary, the evaluation confirms that the songs have a good quality, that they are diverse, and that transformationality of the system is visible in the results. For the purposes of this paper, the two first results are prerequisites for being successful in the last and most important point: transformational creation of songs.

9. Discussion and conclusion

We have presented an approach for automatic, transformational generation of songs, i.e. of coherent pieces of music and lyrics. The architecture of the system is based on the use of constraint programming as a song generation method. A separate component is included in the system for self-evaluation of the produced songs. A meta-level component is in control of transforming the constraints, i.e. transforming song generation, based on the evaluations obtained. In its implementation, the system relies on an off-the-shelf constraint solver.

We next briefly discuss our results and experiences from the point of views of the application (song generation) and the technology (constraint programming for creativity), and then focus more on transformational creativity. We then wrap up with future work.

9.1. Application: song generation

In many existing generative systems that combine language and music, the process is inherently sequential, i.e. either music or language is generated or extracted first and then matching language or music is generated respectively. In our approach, both the lyrics and melody are generated together without one of them being a starting point for generating the other. Also, since the constraint programming framework is declarative, it can directly be used to complement a given melody with lyrics or given lyrics with a melody. Likewise, the constraint programming framework can be used to complete the missing parts of a song based on certain fixed musical or lyrical fragments. These capabilities offer interesting opportunities, e.g. for interactive song-writing systems (but require substantial user interface development work).

According to the user evaluation, the present system is able to produce songs with a reasonably good match between the text and music. In the current system, the basic idea has been to fit longer note durations with stressed syllables and shorter note durations with unstressed syllables. Also, stressed beats are generally accompanied by stressed syllables. Durations of both stressed and unstressed syllables in the songs can be easily constrained further to improve the fit between melody and lyrics.

9.2. Technology: constraint programming for (transformational) creativity

We have proposed constraint programming as a tool for composing songs flexibly in different forms. The immediate benefit of the present approach is that this technology makes it easy to describe and control many interacting characteristics of a given creative domain, songs in our case. Also the match between the linguistic and musical characteristics of songs can be achieved using constraints.

The main benefit of the constraint programming approach for computational creativity is that it enables an easy transformation of the system's internal goal. As the search space is described explicitly by constraints, the shape of the search space can be directly modified by changing the constraints. We will return to transformationality of the architecture in the next subsection.

When running our system, we noticed that many constraint combinations lead to unsatisfiable constraint satisfaction problems. A related issue is that there are few direct relations between constraints and song features, and the search for interesting new conceptual spaces is essentially blind and based on trial and error.

9.3. Transformational creativity

A transformationally creative system does not only produce artefacts but it also generates or modifies the structures that are used to produce these artefacts (Boden, 2004). The present system achieves this transformationality as follows.

First, the system has an explicit representation, as constraints, of its ground-level search space for generating songs, corresponding to the rules $\mathcal{R}$ of Wiggins (2006). Then there is a separate module $\mathcal{E}$ for evaluating qualities of the songs. Finally, a meta-level control module $\mathcal{M}$ modifies the search space (the constraints) and the evaluation function based on the system's own evaluation of the results.

This architecture clearly is transformational, as defined by Boden (2004), since it transforms its own search space. In the terms of Wiggins (2006), it achieves $\mathcal{R}$-transformational creativity. In addition, it can be described as $\mathcal{E}$-transformational since it also modifies its own evaluation function.

Creativity of the meta-level. Transformations in creative systems can take many forms and some forms can be argued to be more transformational or more creative than others. An interesting question is what is transformed. In our case, modifications to the set of constraints transform the search space of songs. These transformations are based on self-reflection and involve reasoning about the generation process on a meta-level, a condition under which "real creativity" may arise (Bundy, 1994).

A possible follow-up question is how novel or creative the modifications themselves are, where they come from, how they are evaluated, and so on – in a sense asking how creative the meta-level is. Our system works with a set of constraints written by the developers, allowing the meta-level to choose which constraints to use, and to modify their parameters in foreseen ways. One can argue that the meta-level operations are simple. However, we emphasise the importance of operating on different conceptual levels, from the ground-level generation to declarative constraints and then further to the meta-level control of these constraints, over the complexity of the meta-level operations.

There are obvious ways to make the meta-level more autonomous. We could add a constraint generator, making the system less dependent on its developers, and thus potentially more creative in some sense. However, as mentioned before, already now our modifications to the constraints can lead to unsatisfiable combinations; obviously, generation of new constraints would have even higher risk for this.

Creative autonomy. A possible dependence of the system on its developers is closely related to the concept of creative autonomy (Jennings, 2010). The current system can be argued to exercise creative autonomy: (1) it can evaluate its liking without an outside source using the evaluation module. (2) It carries out changes to its standards (the constraints and evaluation function) without being explicitly directed when and how to do so. (3) The evaluation and changes above are not purely random.

The requirement of changing the standards without being told *how* to do it deserves further discussion. During runtime, no external agent has control over how the system changes its standards, supporting the argument that the system is autonomous. However, the way standards are changed is based on our specification and the system works without external influence. It can therefore be argued that what the system produces is implicitly present in its initial set of instructions and, in some sense, the results could be traced back to the programmer. The solution to this issue, promoted by Jennings (2010), is social interaction and influence between creative agents.

Inspired by this, we can envision a number of mechanisms to affect the program's execution in the approach that we have proposed. These mechanisms may include interaction with users or other computational systems, for instance in the form of receiving criticism from other agents (giving grounds for adjusting the system's own preferences), observation of songs others have produced (giving grounds to change the evaluation function) or possibly even exchange of constraints between agents.

Intentionality and transparency. Finally, intentionality is a property often associated to intelligence or creativity. The intention of the system is modelled by a combination of the selected features and their target values. The system sets this intention and adjusts it during the process, based on its observations of the songs it has produced.

Given the design of the system, its intent to maximise or minimise certain features can also be presented to the user, as we did in the final part of the empirical evaluation. This aspect clearly raised the interest of the subjects and modified their views of the songs and probably also of the system, but this was not assessed in the evaluation. Making the system more transparent by showing the users the intent of the system is a form of *framing*, i.e. background information about the created artefact that helps put it into context (Charnley, Pease, & Colton, 2012). Transparency and framing are potentially very important factors in the appreciation of results of creative systems. At best, they can help users understand how and why an artefact was produced, and what kind of creative responsibilities the computer had in the process.

The transparent architecture proposed in this paper provides lot of information that could be used beneficially for framing. The rules and evaluation function are represented explicitly and declaratively, easing their communication (but the readability of constraints admittedly is arguable). Perhaps more interestingly, the system can describe its *intent*, as mentioned above, its *motivation* for choosing the intent (the system observed a song with an interesting combination of features and wanted to explore other songs where these

features are emphasised), as well as the *process* of trying to meet the intent (by manipulation of certain constraints).

9.4. Future work

Ground-level song generation can obviously be improved ad infinitum with better lyrics and music production procedures and constraints. In particular, the match between music and lyrics could take advantage of more fine-grained alignment of stresses and lengths of music and lyrics. The set of feature extractors could also be extended to cover even a larger variety of aspects.

The constraint programming framework is effective, but with our set of constraints it is unfortunately not feasible to generate large numbers of songs in the meta-level algorithm to explore a large part of the search space. Modification of constraints at runtime can also lead to unsatisfiable settings. One option to tackle these problems would be to apply constraint optimisation, readily available in ASP, instead of hard constraints.

On the transformational or meta-level, there are several fundamental aspects that deserve further research. Regarding the artefacts generated, the features used by the meta-level component to direct the search and generation of songs should be on a more abstract level, such as moods, emotions and tones. This requires setting or learning mappings between these more abstract features and the lower detail features that can be evaluated from songs. Once the goal is set, heuristics would be useful for choosing suitable constraints to optimise the relevant features; machine learning could possibly be used here to help automate the process. A more challenging setting would be to let the system modify the constraints more freely, even generate new ones. In another direction, to reach full creative autonomy (Jennings, 2010), interaction with the user or other agents would help the system cultivate its own taste even more independently of the developers; this should also happen across sessions. Finally, the architecture would allow generation of better framing information for the produced songs to increase their appreciation by the audience of the system.

Notes

1. Keys are identified by *j* as the number of sharps in a given key signature. Key signatures of seven or more sharps are treated as their enharmonic equivalents, e.g. Db major is an enharmonic equivalent of C# major.
2. Note that here we construct aesthetic objective functions that seek to maximise or minimise the included features, using extreme targets of 0 and 1. This is done intentionally to decrease the degrees of freedom of the system and to help the system overcome competing objectives. However, because the objective functions *are* typically multi-objective, the resulting musical artifacts will rarely exhibit such extremes. Also, it is possible to generalise the approach to incorporate non-binary targets into the objective function as in Murray and Ventura (2012).
3. Baccianella, S., Esuli, A., & Sebastiani, F. (2010). Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In LREC (Vol. 10, pp. 2200–2204).
4. Carnegie Mellon University, "The CMU pronunciation dictionary", 2000, http://www.speech.cs.cmu.edu

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work has been supported by the European Commission under the FET grant 611733 (ConCreTe) and the Academy of Finland under grants 276897 (CLiC), 293411, 1293348 (DLT), 251170 (COIN), 276412, 284591 and 312662.

ORCID

Jukka M. Toivanen http://orcid.org/0000-0001-7297-2501
Matti Järvisalo http://orcid.org/0000-0003-2572-063X
Dan Ventura http://orcid.org/0000-0002-3111-2238
Martti Vainio http://orcid.org/0000-0003-2570-0196
Hannu Toivonen http://orcid.org/0000-0003-1339-8022

References

Anders, T. (2011). Constraint programming systems for modeling music theories and composition. *ACM Computing Surveys*, *43*(4), Article 30.

Boden, M. A. (1998). Creativity and artificial intelligence. *Artificial Intelligence*, *103*(1–2), 347–356.

Boden, M. A. (2004). *The creative mind: Myths and mechanisms* (2nd ed.). London: Psychology Press.

Boenn, G., Brain, M., De Vos, M., & Ffitch, J. (2008). Automatic composition of melodic and harmonic music by answer set programming. In M. Garcia de la Banda & E. Pontelli (Eds.), *Logic programming, 24th International Conference* (Lecture Notes in Computer Science Vol. 5366, pp. 160–174). Udine, Italy: Springer.

Boenn, G., Brain, M., De Vos, M., & Ffitch, J. (2011). Automatic music composition using answer set programming. *Theory and Practice of Logic Programming*, *11*(2–3), 397–427.

Bundy, A. (1994). What is the difference between real creativity and mere novelty?. *Behavioral and Brain Sciences*, *17*(03), 533–534.

Charnley, J., Pease, A., & Colton, S. (2012). On the notion of framing in computational creativity. In M. L. Maher, K. J. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, & G. A. Wiggins (Eds.), *Proceedings of the third international conference on computational creativity* (pp. 77–81). Dublin, Ireland: Association for Computational Creativity.

Colton, S. (2001). Experiments in meta-theory formation. In A. Cardoso & G. A. Wiggins (Eds.), *Proceedings of AISB symposium on artificial intelligence and creativity in arts and science* (pp. 100–109). York, UK: Society for the Study of Artificial Intelligence and Simulation of Behaviour.

Colton, S. (2008). Creativity versus the perception of creativity in computational systems. In D. Ventura, M. L. Maher & S. Colton (Eds.), *Proceedings of the AAAI spring symposium on creative intelligent systems* (pp. 14–20). Stanford, CA: AAAI.

Colton, S., Bundy, A., & Walsh, T. (1999). Automatic concept formation in pure mathematics. In T. Dean (Ed.), *Proceedings of the 16th international joint conference on artificial intelligence* (Vol. 2, pp. 786–791). Stockholm, Sweden: Morgan Kaufmann.

Colton, S., Goodwin, J., & Veale, T. (2012). Full-FACE poetry generation. In M. L. Maher, K. J. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, & G. A. Wiggins (Eds.), *Proceedings of the third international conference on computational creativity* (pp. 95–102). Dublin, Ireland: Association for Computational Creativity.

Colton, S., & Wiggins, G. A. (2012). Computational creativity: The final frontier? In L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, & P. J. F. Lucas (Eds.), *Proceedings of the European conference on artificial intelligence* (pp. 21–26). Montpellier, France: IOS Press.

Dubuisson, M.-P., & Jain, A. K. (1994). A modified Hausdorff distance for object matching. In S. Peleg & S. Ullman (Eds.), *Proceedings of the 12th IAPR international conference on pattern recognition* (Vol. 1, pp. 566–568). Jerusalem, Israel: IEEE.

El Bolock, A., & Abdennadher, S. (2015). Towards automatic poetry generation using constraint handling rules. In R. L. Wainwright, J. M. Corchado, A. Bechini, & J. Hong (Eds.), *Proceedings of the 30th annual ACM symposium on applied computing* (pp. 1868–1873). Salamanca, Spain: ACM.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). *Answer set solving in practice*. San Rafael, CA: Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2014). Clingo = ASP + control: Preliminary report. *CoRR*, *abs/1405.3694*.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In R. A. Kowalski & K. A. Bowen (Eds.), *Proceedings of the fifth international conference and symposium on logic programming* (pp. 1070–1080). Seattle, WA: MIT Press.

Gervás, P. (2001). An expert system for the composition of formal Spanish poetry. *Journal of Knowledge-Based Systems*, *14*(3–4), 181–188.

Grace, K., & Maher, M. L. (2015). Specific curiosity as a cause and consequence of transformational creativity. In H. Toivonen, S. Colton, M. Cook, & D. Ventura (Eds.), *Proceedings of the sixth international conference on computational creativity* (pp. 260–267). Park City, UT: Association for Computational Creativity.

Jennings, K. E. (2010). Developing creativity: Artificial barriers in artificial intelligence. *Minds and Machines*, *20*(4), 489–501.

Jordanous, A. K. (2009). Evaluating machine creativity. In N. Bryan-Kinns, M. D. Gross, H. Johnson, J. Ox, & R. Wakkary (Eds.), *Proceedings of the seventh ACM conference on creativity and cognition* (pp. 331–332). Berkeley, CA: ACM.

Lenat, D. B. (1983). Eurisko: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, *21*(1–2), 61–98.

Lenat, D. B., & Brown, J. S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence*, *23*(3), 269–294.

Manurung, H. M. (2003). *An evolutionary algorithm approach to poetry generation* (Doctoral dissertation). University of Edinburgh, College of Science and Engineering, School of Informatics.

Mihalcea, R., & Strapparava, C. (2012). Lyrics, music, and emotions. In J. Tsujii, J. Henderson, & M. Pasca (Eds.), *Proceedings of the conference on empirical methods in natural language processing* (pp. 590–599). Jeju Island, South Korea: ACL.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Monteith, K., Francisco, V., Martinez, T., Gervás, P., & Ventura, D. (2011). Automatic generation of emotionally-targeted soundtracks. In D. Ventura, P. Gervás, D. F. Harrell, M. L. Maher, A. Pease, & G. A. Wiggins (Eds.), *Proceedings of the second international conference on computational creativity* (pp. 60–62). Mexico City, Mexico: Association for Computational Creativity.

Monteith, K., Martinez, T., & Ventura, D. (2010). Automatic generation of music for inducing emotive response. In D. Ventura, A. Pease, R. Pérez y Pérez, G. Ritchie & T. Veale (Eds.), *Proceedings of the first international conference on computational creativity* (pp. 140–149). Lisbon, Portugal: Association for Computational Creativity.

Monteith, K., Martinez, T., & Ventura, D. (2012). Automatic generation of melodic accompaniments for lyrics. In M. L. Maher, K. J. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, & G. A. Wiggins (Eds.), *Proceedings of the third international conference on computational creativity* (pp. 87–94). Dublin, Ireland: Association for Computational Creativity.

Murray, S. J., & Ventura, D. (2012). Algorithmically flexible style composition through multi-objective fitness functions. In P. Pasquier, A. Eigenfeldt, & O. Bown (Eds.), *Proceedings of the first international workshop on musical metacreation* (pp. 55–62). Stanford, Palo Alto, CA: AAAI Press.

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, *25*(3–4), 241–273.

Nienhuys, H. W., & Nieuwenhuizen, J. (2003). Lilypond, a system for automated music engraving. In N. Bernardini, F. Giomi, & N. Giosmin (Eds.), *Proceedings of the XIV colloquium on musical informatics* (pp. 167–172). Florence, Italy: AIMI.

Oliveira, H. G., Cardoso, F. A., & Pereira, F. C. (2007). Tra-la-lyrics: An approach to generate text based on rhythm. In A. Cardoso & G. A. Wiggins (Eds.), *Proceedings of fourth international joint workshop on computational creativity* (pp. 47–55). London, UK: Goldsmiths, University of London.

Papadopoulos, A., Pachet, F., Roy, P., & Sakellariou, J. (2015). Exact sampling for regular and Markov constraints with belief propagation. In G. Pesant (Ed.), *Principles and practice of constraint programming* (Lecture Notes in Computer Science Vol. 9255, pp. 341–350). Springer International Publishing.

Prusinkiewicz, P. (1986). Score generation with L-systems. In P. Berg (Ed.), *Proceedings of the 1986 international computer music conference* (pp. 455–457). Den Haag, Netherlands: Michigan Publishing.

Ramakrishnan, A., Kuppan, S., & Devi, S. L. (2009). Automatic generation of Tamil lyrics for melodies. In A. Feldman & B. Lönneker-Rodman (Eds.), *Proceedings of the workshop on computational approaches to linguistic creativity* (pp. 40–46). Boulder, CO: ACL.

Ritchie, G. (2006). The transformational creativity hypothesis. *New Generation Computing*, *24*(3), 241–266.

Roads, C. (1996). *The computer music tutorial*. Cambridge, MA: The MIT Press.

Scirea, M., Barros, G. A., Shaker, N., & Togelius, J. (2015). SMUG: Scientific music generator. In H. Toivonen, S. Colton, M. Cook, & D. Ventura (Eds.), *Proceedings of the sixth international conference on computational creativity* (pp. 204–211). Park City, UT: Association for Computational Creativity.

Simon, I., Morris, D., & Basu, S. (2008). MySong: Automatic accompaniment generation for vocal melodies. In M. Czerwinski, A. M. Lund, & D. S. Tan (Eds.), *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 725–734). Florence, Italy: ACM.

Simons, P., Niemelä, I., & Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, *138*(1), 181–234.

Sridhar, R., Gladis, D. J., Ganga, K., & Prabha, G. D. (2014). Automatic Tamil lyric generation based on ontological interpretation for semantics. *Sadhana*, *39*(1), 97–121.

Sukumaran, S., & Dheepa, G. (2003). Generation of fractal music with Mandelbrot set. *Global Journal of Computer Science and Technology*, *1*(1), 127–130.

Tobing, B. C., & Manurung, R. (2015). A chart generation system for topical metrical poetry. In H. Toivonen, S. Colton, M. Cook, & D. Ventura (Eds.), *Proceedings of the sixth international conference on computational creativity* (pp. 308–314). Park City, UT: Association for Computational Creativity.

Toivanen, J. M., Järvisalo, M., & Toivonen, H. (2013). Harnessing constraint programming for poetry composition. In M. L. Maher, T. Veale, R. Saunders, & O. Bown (Eds.), *Proceedings of the fourth international conference on computational creativity* (pp. 160–167). Sydney, Australia: Association for Computational Creativity.

Toivanen, J. M., Toivonen, H., Valitutti, A., & Gross, O. (2012). Corpus-based generation of content and form in poetry. In M. L. Maher, K. J. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, & G. A. Wiggins (Eds.), *Proceedings of the third international conference on computational creativity* (pp. 175–179). Dublin, Ireland: Association for Computational Creativity.

Toivanen, J. M., Toivonen, H., & Valitutti, A. (2013). Automatical composition of lyrical songs. In M. L. Maher, T. Veale, R. Saunders, & O. Bown (Eds.), *Proceedings of the fourth international conference on computational creativity* (pp. 87–91). Sydney, Australia: Association for Computational Creativity.

Tulilaulu, A., Paalasmaa, J., Waris, M., & Toivonen, H. (2012). Sleep musicalization: Automatic music composition from sleep measurements. In J. Hollmén, F. Klawonn & A. Tucker (Eds.), *Advances in intelligent data analysis XI* (Lecture Notes in Computer Science Vol. 7619, pp. 392–403). Helsinki: Springer.

Wiggins, G. A. (2006). A preliminary framework for description, analysis and comparison of creative systems. *Knowledge-Based Systems*, *19*(7), 449–458.

Appendix. Musical and lyrical feature extractors

Let $\mathcal{I}$ be the set of all possible sequences of note pitches, $\mathcal{D}$ be the set of all possible note durations, $\mathcal{W}$ be the set of all possible sequences of words and $\mathcal{C}$ the set of all possible chord sequences. We can represent a song as the four-tuple (I, D, W, C) where $I \in \mathcal{I}$, $D \in \mathcal{D}$, $W \in \mathcal{W}$ and $C \in \mathcal{C}$ represent the melody pitches, melody note durations, lyrics and chords, respectively. Then, for instance, $I = i_1, i_2, \ldots, i_n$, where i_j is the pitch of an individual note, and $W = w_1, w_2, \ldots, w_m$, where w_j is an individual word of the lyrics. An individual note i can take any pitch value in a 2 octave range, with

note values represented by numbers in the range [0, 23], i.e. $i_j \in [0, 23],\ 1 \leq j \leq n$. An individual word w can be any member of the vocabulary V of words, so $w_j \in V,\ 1 \leq j \leq m$.

An evaluation function $\mathcal{E} : \mathcal{I} \times \mathcal{D} \times \mathcal{W} \times \mathcal{C} \rightarrow [0, 1]$ can be constructed using a weighted linear combination of the following 14 features to evaluate songs produced by the generation component. See Sections 5 and 6 for how the features are used.

(1) Self-similarity of melody (Murray & Ventura, 2012). Measures how often repeating intervals occur in sequence I.

$$\text{SelfSimilarMelody}(I) = \frac{\mu - 1}{|I| - 3}, \tag{A1}$$

where $\mu = (1/|S|) \sum_{s \in S} count_s(I)$ and S is the set of all interval sequences of length 2 that appear in I. $count_s(I)$ is the number of times interval sequence s occurs in I.

(2) Self-similarity of rhythm. Measures how often repeating patterns appear in a sequence of durations D.

$$\text{SelfSimilarRhythm}(D) = \frac{\mu - 1}{|D| - 2}, \tag{A2}$$

where $\mu = (1/|S|) \sum_{s \in S} count_s(D)$ and S is the set of all duration sequences of length 2 that appear in D. $count_s(D)$ is the number of times duration sequence s appears in D.

(3) Note prevalence. Twelve different functions for each possible pitch class. Measures the proportion of notes in I that represent a specific pitch class.

$$\text{NotePrevalence}_j(I) = \frac{count_j(I)}{|I|}, \tag{A3}$$

where $0 \leq j \leq 11$ denotes the notes C, C#, ..., B, and $count_j(I)$ is the number of times pitch class j appears in I (pitches above a one-octave range are reduced to their first-octave equivalents).

(4) Key prevalence (Murray & Ventura, 2012). Twelve different functions for each possible key centre. Measures the proportion of notes in I that represent a specific key.

$$\text{KeyPrevalence}_j(I) = \frac{|K_j|}{|I|}, \tag{A4}$$

where $K_j = \{i \in I \mid i \in Key_j\}, 0 \leq j \leq 11$, and j indicates the number of sharps in the key signature. Key signatures with seven or more sharps are treated as equivalent enharmonic key signatures with flats, e.g. C# major is an enharmonic equivalent of Db major. Thus Key_0 is C Major, Key_1 is G Major, Key_7 is Db major, etc.

(5) Pitch range. Measures how much of the possible pitch range (two octaves) is used by I, with 0 meaning none of the range is used and 1 meaning all of the range is used.

$$\text{PitchRange}(I) = \frac{|S|}{24}, \tag{A5}$$

where S denotes the set of unique pitches in I and 24 is the total number of notes in a two-octave range.

(6) Interval class prevalence (Murray & Ventura, 2012). Measures the proportion of intervals in a pitch sequence $I = i_1, i_2, \ldots, i_n$ that represent a particular interval class j. Thus there are 24 different functions, $0 \leq j \leq 23$.

$$\text{IntClassPrev}_j(I) = \frac{\sum_{k=1}^{n-1} \delta(j, i_{k+1} - i_k)}{n - 1}, \tag{A6}$$

where $\delta()$ is the Kronecker delta function.

(7) Variability of rhythm. Measures the proportion of *unique* note duration pairs (d_i, d_{i+1}) in a note duration sequence $D = d_1, d_2, \ldots, d_n$:

$$\text{RhythmVariability}(D) = \frac{1}{n - 1} \sum_{j=1}^{n-1} \delta(d_j, d_{j+1}), \tag{A7}$$

where $\delta()$ is the Kronecker delta function.

(8) Average duration of a note. For sequence of note durations D

$$\text{AverageDuration}(D) = \frac{1}{|D|}\sum_{d\in D} d. \tag{A8}$$

(9) Variability of harmony. Measures the proportion of unique chords in a given chord sequence C.

$$\text{HarmonyVariability}\,(C) = \frac{|S|}{|C|}, \tag{A9}$$

where $|S|$ denotes the set of unique chords in C.

(10) Positive sentiment of lyrics. Computes the average sentiment value of individual words w_i in the lyrics W based on SentiWordnet[3].

$$\text{PositiveLyricsSentiment}(W) = \frac{1}{|W|}\sum_{w\in W} \sigma(w), \tag{A10}$$

where $\sigma(w)$ returns a value in the range [0, 1], with 0 meaning not at all positive and 1 meaning completely positive.

(11) Phoneme structure of lyrics. Measures the proportion of consonant and vowel phonemes in the lyrics using two different functions:

$$\text{ProportionConsonants}(W) = \frac{1}{|L|}\sum_{p\in L} \text{con}(p) \tag{A11}$$

and

$$\text{ProportionVowels}(W) = \frac{1}{|L|}\sum_{p\in L} \text{vow}(p), \tag{A12}$$

where L is the sequence of phonemes $p_1, p_2, \ldots, p_n$ formed by the lyrics W, con() returns 1 if its argument is a consonant and 0 otherwise, and vow() returns 1 if its argument is a vowel and 0 otherwise. (In the case of English, the phonetic representation is formed with the CMU pronunciation dictionary.[4])

(12) Word rarity. Computes the average frequency of the words in lyrics W.

$$\text{Rarity}(W) = \frac{1}{|W|}\sum_{w\in W} \text{freq}(w), \tag{A13}$$

where freq() returns the frequency of its argument in Wikipedia (normalized to the range [0, 1]).

(13) Stressed note lengths. Computes the proportion of note durations associated with stressed lyric syllables:

$$\text{Stressed}(D, W) = \frac{\sum_{d\in D \wedge \text{str}(d)} d}{\sum_{d\in D} d}, \tag{A14}$$

where str() is TRUE if its argument is associated with a stressed syllable in W.

(14) Melody consonance. Measures the average consonance of the melodic pitches w.r.t. the underlying chords.

$$\text{Consonance}(I, C) = \frac{1}{|I|}\sum_{i\in I} \text{cons}(i, c_i), \tag{A15}$$

where c_i is the chord in C that holds during note i, and cons(i, c) returns a value in the range [0, 1] based on consonance values of the intervals between i and the pitches constituting c.