Algorithmic Composition of Music in Real-Time with Soft Constraints

Max Meier

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München
zur Erlangung des Grades Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von
Max Meier

eingereicht im September 2014

All you have to do is touch the right key at the right time and the instrument will play itself.

(J. S. Bach)

Berichterstatter:

Prof. Dr. Martin Wirsing
Prof. Dr. Elisabeth André
Prof. Dr. Andreas Butz

Tag des Rigorosums:

8.12.2014

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Max Meier

Ort, Datum Unterschrift Doktorand/in

Abstract

Music has been the subject of formal approaches for a long time, ranging from Pythagoras' elementary research on tonal systems to J. S. Bach's elaborate formal composition techniques. Especially in the 20th century, much music was composed based on formal techniques: Algorithmic approaches for composing music were developed by composers like A. Schoenberg as well as in the scientific area. So far, a variety of mathematical techniques have been employed for composing music, e.g. probability models, artificial neural networks or constraint-based reasoning. In the recent time, interactive music systems have become popular: existing songs can be replayed with musical video games and original music can be interactively composed with easy-to-use applications running e.g. on mobile devices. However, applications which algorithmically generate music in real-time based on user interaction are mostly experimental and limited in either interactivity or musicality. There are many enjoyable applications but there are also many opportunities for improvements and novel approaches.

The goal of this work is to provide a general and systematic approach for specifying and implementing interactive music systems. We introduce an algebraic framework for interactively composing music in real-time with a reasoning-technique called 'soft constraints': this technique allows modeling and solving a large range of problems and is suited particularly well for problems with soft and concurrent optimization goals. Our framework is based on well-known theories for music and soft constraints and allows specifying interactive music systems by declaratively defining 'how the music should sound' with respect to both user interaction and musical rules. Based on this core framework, we introduce an approach for interactively generating music similar to existing melodic material. With this approach, musical rules can be defined by playing notes (instead of writing code) in order to make interactively generated melodies comply with a certain musical style. We introduce an implementation of the algebraic framework in .NET and present several concrete applications: 'The Planets' is an application controlled by a table-based tangible interface where music can be interactively composed by arranging planet constellations. 'Fluxus' is an application geared towards musicians which allows training melodic material that can be used to define musical styles for applications geared towards non-musicians. Based on musical styles trained by the Fluxus sequencer, we introduce a general approach for transforming spatial movements to music and present two concrete applications: the first one is controlled by a touch display, the second one by a motion tracking system. At last, we investigate how interactive music systems can be used in the area of pervasive advertising in general and how our approach can be used to realize 'interactive advertising jingles'.

Zusammenfassung

Musik ist seit langem Gegenstand formaler Untersuchungen, von Phytagoras‘ grundlegender Forschung zu tonalen Systemen bis hin zu J. S. Bachs aufwändigen formalen Kompositionstechniken. Vor allem im 20. Jahrhundert wurde vielfach Musik nach formalen Methoden komponiert: Algorithmische Ansätze zur Komposition von Musik wurden sowohl von Komponisten wie A. Schoenberg als auch im wissenschaftlichem Bereich entwickelt. Bislang wurde eine Vielzahl von mathematischen Methoden zur Komposition von Musik verwendet, z.B. statistische Modelle, künstliche neuronale Netze oder Constraint-Probleme. In der letzten Zeit sind interaktive Musiksysteme populär geworden: Bekannte Songs können mit Musikspielen nachgespielt werden, und mit einfach zu bedienenden Anwendungen kann man neue Musik interaktiv komponieren (z.B. auf mobilen Geräten). Allerdings sind die meisten Anwendungen, die basierend auf Benutzerinteraktion in Echtzeit algorithmisch Musik generieren, eher experimentell und in Interaktivität oder Musikalität limitiert. Es gibt viele unterhaltsame Anwendungen, aber ebenso viele Möglichkeiten für Verbesserungen und neue Ansätze.

Das Ziel dieser Arbeit ist es, einen allgemeinen und systematischen Ansatz zur Spezifikation und Implementierung von interaktiven Musiksystemen zu entwickeln. Wir stellen ein algebraisches Framework zur interaktiven Komposition von Musik in Echtzeit vor welches auf sog. ‚Soft Constraints‘ basiert, einer Methode aus dem Bereich der künstlichen Intelligenz. Mit dieser Methode ist es möglich, eine große Anzahl von Problemen zu modellieren und zu lösen. Sie ist besonders gut geeignet für Probleme mit unklaren und widersprüchlichen Optimierungszielen. Unser Framework basiert auf gut erforschten Theorien zu Musik und Soft Constraints und ermöglicht es, interaktive Musiksysteme zu spezifizieren, indem man deklarativ angibt, ‚wie sich die Musik anhören soll‘ in Bezug auf sowohl Benutzerinteraktion als auch musikalische Regeln. Basierend auf diesem Framework stellen wir einen neuen Ansatz vor, um interaktiv Musik zu generieren, die ähnlich zu existierendem melodischen Material ist. Dieser Ansatz ermöglicht es, durch das Spielen von Noten (nicht durch das Schreiben von Programmcode) musikalische Regeln zu definieren, nach denen interaktiv generierte Melodien an einen bestimmten Musikstil angepasst werden. Wir präsentieren eine Implementierung des algebraischen Frameworks in .NET sowie mehrere konkrete Anwendungen: ‚The Planets‘ ist eine Anwendung für einen interaktiven Tisch mit der man Musik komponieren kann, indem man Planetenkonstellationen arrangiert. ‚Fluxus‘ ist eine Anwendung, die sich an Musiker richtet. Sie erlaubt es, melodisches Material zu trainieren, das wiederum als Musikstil in Anwendungen benutzt werden kann, die sich an Nicht-Musiker richten. Basierend auf diesen trainierten Musikstilen stellen wir einen generellen Ansatz vor, um räumliche Bewegungen in Musik umzusetzen und zwei konkrete Anwendungen basierend auf einem Touch-Display bzw. einem Motion-Tracking-System. Abschließend untersuchen wir, wie interaktive Musiksysteme im Bereich ‚Pervasive Advertising‘ eingesetzt werden können und wie unser Ansatz genutzt werden kann, um ‚interaktive Werbejingles‘ zu realisieren.

Contents

1 Introduction ... 11
2 Soft Constraints ... 16
2.1 Monoidal Soft Constraints ... 17
2.2 Approach: Coordinating Dynamically Changing Preferences ... 23
2.3 Approaches for Solving Soft Constraints ... 25
2.4 A Solver for Monoidal Soft Constraints ... 27
3 Music Theory ... 30
4 Algorithmic Composition ... 36
5 Approach: Composing Music with Soft Constraints ... 43
5.1 Music Theory ... 44
5.2 Musical Soft Constraints ... 47
5.3 Trainable Musical Models ... 53
5.4 Related Work ... 58
5.4.1 Constraints and Music ... 58
5.4.2 Style Imitation ... 60
5.4.3 Interactive Music Systems ... 61
6 Framework Design and Implementation ... 64
6.1 Requirements ... 65
6.2 Framework Components ... 66
6.2.1 Music Theory ... 66
6.2.2 Soft Constraints ... 71
6.2.3 Musical Soft Constraints ... 75
6.2.4 Trainable Musical Models ... 81
6.3 Evaluation ... 88
6.3.1 Quality Analysis ... 88
6.3.2 Performance Tests ... 89
7 Applications ... 93
7.1 'The Planets' for Microsoft Surface ... 93
7.1.1 Concept ... 94

7.1.2 Realization 96
7.1.3 The Planets for Windows Phone 7 102
7.1.4 Related Work 104
7.2 Fluxus Pattern Sequencer 105
7.2.1 Concept 106
7.2.2 Realization 108
7.2.3 Related Work 113
7.3 Transforming Spatial Movements to Music 115
7.3.1 Approach: Generating Music based on Spatial Movements 115
7.3.2 Applications: Touch Display and Motion Tracking 119
7.3.3 Related Work 121
7.4 Interactive Advertising Jingles 122
7.4.1 Requirements and Application Areas 122
7.4.2 Approach: Interactive Advertising Jingles 124
7.4.3 Study: Interactive Music on Public Displays 126
7.4.4 Related Work 128
7.4.5 Conclusion 130
8 Conclusion and further work 131
Bibliography 133
Picture Credits 143

1 Introduction

Typically, automata accomplish tasks which are considered as 'useful', for example selling beverages, washing dishes or building cars. However, automata have also been built to entertain people for a long time: In the first century A.D., the Greek inventor Heron of Alexandria designed a mechanical theatre where puppets are automatically moved by a complex mechanism of strings, weights and axes (1). Musical automatons belong to the earliest known designs of programmable machines (2): In a work attributed to Archimedes, an automatic flute player is described which is driven by air that is compressed by a complex hydraulic system and in the 9th century, long before programmable looms and calculation machines were invented, the brothers Mūsā in Baghdad described an automatic flute player which is controlled by pins on a rotating drum that open the holes of a flute via little levers. By using other configurations of pins, the automaton can be programmed to play different melodies (3).

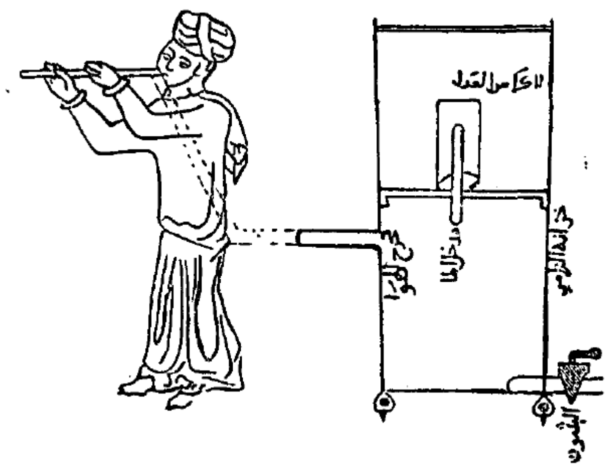


FIGURE 1 ARCHIMEDES AUTOMATIC FLUTE PLAYER (3)

In the Renaissance, entertainment automata were popular at court: besides musical automata, many other kinds of mechanical devices were used to entertain and impress, for example trick fountains, artificial animals or other technical curiosa. Programmable carillons are known since the 13th century and are often combined with visual elements. For example, the 'Strasbourg astronomical clock' from 1354 had a gilded rooster which 'opened its beak, stretched out its tongue, flapped its wings, spread out its feathers and crowed' (4). Another famous example for such automaton is the 'Rathaus-Glockenspiel' in Munich. Musical automata have also been built based on a variety of other instruments, for example so-called 'Orchestrions' which automatically play music using e.g. pipe organs, pianos or percussion instruments.

Besides entertaining people, the development of entertainment automata can also drive technology in general: in the 18th century, Jacques de Vaucanson invented several programmable musical automata before he designed the first automatic loom based on punched cards (2). This design was later adapted and improved by Joseph-Marie Jacquard and played an important role in the development of computers. A more recent example for entertainment driving technology is the video game industry which has effects on various other areas like computer graphics or artificial intelligence (5).

Today, there exist a variety of popular 'musical automata'. With music video games, one can sing ('Sing Star'), play instruments ('Guitar Hero') or dance ('Dance Dance Revolution') to popular songs. The goal of these games is to accomplish a set of predefined actions as accurate as possible in order to achieve a good score. Besides games where music is the integral part, there are also games where dynamically generated music is used to enrich the gaming experience (6). It is already common to dynamically select longer pieces of music in order to create a certain mood (e.g. when the player gets involved in a fight), but there are also games where music is generated on a more fine-grained level, e.g. directly based on his actions like shooting. Besides music games where one has to achieve a certain goal, there are also plays (i.e. invitations to less structured activities) where music can be played without any additional goal. Targeted particularly at non-musicians, these applications allow intuitively playing music in a rather simple way. Typical application areas for interactive music systems are casual games (e.g. on mobile devices or in the Internet), public installations (e.g. at an art exhibition) or as part of a professional musical performance. Interactive music is an area of research at the moment and most existing systems are rather experimental: systems which generate well-sounding music are often limited in interactivity and mostly based on pre-recorded pieces of music which can be combined in different ways. Vice versa, highly interactive systems which provide more direct and immediate control over the shape of melodies are mostly limited in musicality. There exist many applications which are fun to play with and produce appealing sound, but there are also many opportunities for improvements.

In the scientific area, a variety of approaches for algorithmically generating music have been investigated. Of particular relevance are systems based on so-called *constraints,* a technique that allows defining rules with logical formulas. Constraints have been employed for algorithmic composition of music since the late 1970's to our knowledge (7) and have been widely used to generate musical scores that fit to an existing melody and follow several general musical rules (known as the 'Automatic Harmonization Problem'). The actual rules for various musical eras are well-known, for example in the era of Baroque it is not allowed to keep the fifth interval in two successive notes (the so-called 'parallel fifths' rule). Most such rules state incompatibilities, so constraints are very adequate for formulating such music theories. These classical Boolean constraints come to their limits when there are rules that do not have to be satisfied in any case or which are hard to formulate. In the recent years, an improved technique called *soft constraints* has been of general interest that extends classical constraints and provides a far

more expressive framework for defining rules. Besides several weaker notions of soft constraints, a very general and expressive framework based on work from Stefano Bistarelli, Ugo Montanari and Francesca Rossi (8) is of particular relevance. Besides classical logical rules, this framework allows defining certain levels of acceptance for solutions and is suited well for solving problems with concurrent and contradictory optimization goals. Soft constraints with a comparable generality and expressiveness have never been used for composing music. To us, they seem very promising for algorithmically generating music, especially in interactive real-time systems with many soft and concurrent rules. Compared to classical constraints, additional types of rules can be defined with soft constraints, for example in order to maximize the harmony of musical intervals between several voices. In this work, we want to develop a general approach for generating music with soft constraints in interactive real-time systems. Another widely used technique in the area of algorithmic composition is machine learning: the rules for generating music are derived from existing music, e.g. by training a statistical model or learning grammatical rules. Machine learning techniques are often used to originally compose new music that is consistent with a previously trained musical style – whereas constraints are mostly used to generate harmonization voices to an existing melody. It would be desirable to have both (soft) constraints and machine learning techniques in one framework.

The main goal of this work is to extend known approaches for composing music based on classical constraints with soft constraints and to provide a general and systematic approach for specifying and implementing interactive music systems. We present an algebraic framework for declaratively specifying systems which generate music in real-time based on user interaction. The desired musical output is described in a conceptual way with (soft) constraints that specify 'how the music should sound'. These constraints can be derived from multiple sources: general musical constraints can for example enforce all notes to be on a certain tonal scale or to follow certain harmonic progressions. The mapping between user interaction and desired musical result can also be expressed with constraints that are generated for example from the sensor readings of an accelerometer or motion tracker. These constraints express the user's preferences for certain aspects of the music, e.g. 'I want to play fast and high notes'. Multiple voices can be coordinated among each other by declaring soft constraints that express rules for the combination of voices and, for example, prefer harmonic intervals between them. Another typical kind of soft constraint defines rules for melodic progressions that are used to generate music similar to given melodic material in order to e.g. make it consistent with a certain musical style or an advertising jingle. These constraints are generated from a general musical transition model and can be defined by training this model with one or several existing melodies.

We implemented our framework in .NET based on four main components: *Music* models basic concepts of music theory (notes, intervals, scales…) and provides an infrastructure for sending and receiving notes and metric information. *Soft Constraints* allows modeling and solving problems with soft constraints. Based on the latter two, *Musical Soft Constraints* makes it possible to compose music with soft constraints. Using this component, *Musical Model* implements our approach for training melodic material. Besides the core functionality specified in the algebraic framework, there is also much additional functionality for conveniently developing applications with only few programming code. Much effort was spent on achieving a high level of software quality with high demands on non-functional requirements.

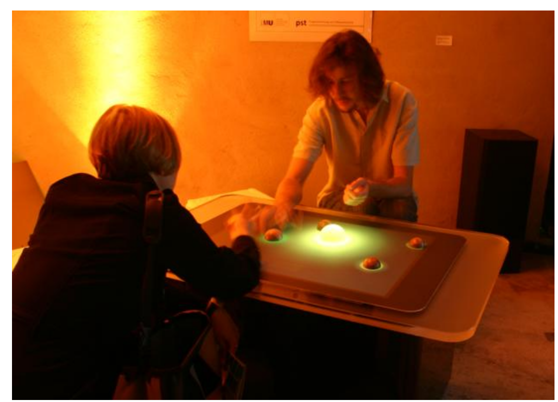


FIGURE 2 LEFT: 'THE PLANETS', RIGHT: TRANSFORMING BODY MOVEMENTS TO MUSIC

Based on the algebraic framework and its implementation, we introduce several applications: 'The Planets' is an application for the Microsoft Surface table where music can be interactively composed by arranging planet constellations. Each planet represents a certain instrument which is controlled by its relative position towards the sun: moving it closer to the sun makes it play faster; rotating it around the sun changes its pitch. Furthermore, it is also possible to control global parameters with two special planets, e.g. the global harmony between all planets. 'Fluxus' is a Windows desktop application which is geared towards musicians and makes it possible to improvise music and train melodic material which can be used in applications geared towards non-musicians. This application makes use of a pattern-based approach for training dynamic models (as well as recording static melodies) for several instruments and re-playing them in different combinations. Based on melodic material trained by the Fluxus sequencer, we present a general approach for transforming spatial movements to music and two concrete applications: the first one is controlled by two-dimensional movements on a touch display, the second one by three-dimensional movements of body parts detected by a motion tracking system (Microsoft Kinect). Intuitively, this approach is based on the following interaction paradigm: the faster one moves his fingers or body parts, the faster the resulting melodies – the

higher their position, the higher the notes. At last, we investigate how interactive music systems can be used in the area of pervasive advertising in general and how our framework can be used to realize ‘interactive advertising jingles’ which can be interactively played on the one hand but can still be recognized as a given brand melody on the other hand. We implemented a prototype where interactive advertising jingles can be played with hand movements in front of a public display and conducted a study in order to find out how novice users can interactively play music without any previous training period.

This work is organized as follows: In chapter 2, we show how problems can be modeled and solved with soft constraints and introduce a general approach for specifying systems which have to deal with dynamically changing preferences. In chapter 3, we give a short introduction to music theory with a focus on aspects relevant for this work. In chapter 4, we give an overview over the area of algorithmic composition. In chapter 5, we introduce our algebraic framework for declaratively specifying interactive music systems with soft constraints and compare our approach with related ones. In chapter 6, we introduce the .NET implementation of the framework and assess its quality. In chapter 7, we introduce the applications that have been developed using the framework so far and present results of our investigations how interactive music systems can be used in the area of pervasive advertising.

2 Soft Constraints

Constraint programming is very different from most generally known programming paradigms. Instead of specifying an algorithmic procedure which computes a desired result step-by-step, the result itself is specified in a conceptual 'declarative' way. An overly simplified analogy to this is two different ways of ordering a pizza in a restaurant.

The *procedural* way would sound like this:

'Take 250g flour, 1 packet of dry yeast, 1 cup of warm water, some salt and mix it all together. Then put a sheet over the bowl, keep it in a warm place and wait for 30 minutes. Afterwards, roll everything out on a baking tray with a paper below and brush it with olive oil. Top it with tomato sauce, cheese and ham and put it in the oven at 200°C. After half an hour, bring it to me, please!'

The *declarative* way would sound like this:

'I want a Pizza Prosciutto, please.'

The second approach is not just shorter than the first one - it is also likely that the pizza will arrive earlier and taste better because the given recipe is very primitive and most chefs will have more sophisticated ways of making a pizza, since they typically have lots of experience with that.

Similarly, declarative programming makes it easy to model and solve certain kinds of problems very efficiently by only describing the properties of a desired result and leaving the actual work of computing it to a universally applicable software component (a so-called *solver*). In many cases, it would also require much work to develop a problem-specific solver from scratch which could compete with a highly optimized solver for general problems. One approach to declarative programming is based on so-called *constraints*: In classical constraint programming, a desired result is specified with several conditions constraining its properties. These conditions are expressed as logical formulas which imply a hard border between correct and incorrect results. Constraint satisfaction problems (CSP) proved to be useful in a large field of applications ranging from cabin layout design for airplanes to proving correctness of software. Constraints are also often used for solving scheduling problems (e.g. creation of timetables, planning of production processes, logistical problems…). However, there are also many real-life problems where classical constraints come to their limits because the border between 'good' and 'bad' results is fuzzy or hard to formulate. A way to deal with such problems is *soft constraints*. Intuitively, a soft constraint represents a condition which does not have to be met in any case. There are numerous formalisms for modeling soft constraints, for example by extending classical constraints with a global cost function (constraint optimization problems, COP) or maximizing the number of satisfied constraints (Max-CSP). For detailed information about modeling and

solving problems with constraints we refer to Rina Dechter´s book ´Constraint Processing´ (9). There is also a survey of approaches for modeling soft constraints (10). Of particular relevance to us is the work of Stefano Bistarelli, Ugo Montanari and Francesca Rossi who developed a very elegant framework for soft constraints (8) which is general enough to model many of the other approaches. This framework is based on so-called ´semirings´; a detailed overview about it is given in Stefano Bistarelli´s book ´Semirings for Soft Constraint Solving and Programming´ (11). The formalism used in this work – monoidal soft constraints (12) - is a slightly modified version of this framework and will be introduced in the next section. Our contribution to this work is the extension of the existing soft constraint solver with Cartesian products of monoids and partial orders.

2.1 Monoidal Soft Constraints

A soft constraint rates different alternatives for solving a problem by assigning a certain *grade* to each possible option. This makes it possible to compare alternative solutions among each other and search for the best one. The framework introduced here is not restricted to a certain type of grades: instead, they are modeled in a very modular and versatile way with an abstract algebraic structure, i.e. an abstract description of a set of grades with associated operations for combining and comparing them. Many different types of concrete grades can be used, for example numbers (integer, real…) or Boolean values. The framework is originally based on work from Bistarelli et al. (8), who used so-called *semirings* for modeling grades and operations on them. This framework was modified and extended by Hölzl, Meier and Wirsing in (12): the semirings were replaced by *ordered monoids* in order to provide a more natural way for defining grades and being able to define certain kinds of ‘meta-preferences’ over the constraints itself. The formalism introduced here was again slightly modified in (13).

FIGURE 3 PIZZA TOPPINGS

As a running example in this section, we model preferences for ordering a pizza with two toppings and a matching beverage. The first topping can be ham, pepperoni or tuna; the second one onions, pineapple or mushrooms. The corresponding beverage can be wine, beer or water. We want to express preferences for topping combinations (e.g. ‘I like ham together with onions’) as well as preferences for the beverage depending on the toppings (e.g. ‘ham and wine fits quite well’).

In general, possible solutions of a problem are modeled by assigning values to variables. $Variable$ is a finite set of problem variables and $Value$ a finite set of problem values. Values can be assigned to variables with a $Valuation$, a function from variables to values:

$$Valuation = Variable \rightarrow Value$$

In our example, we define the sets of variables and values like this:

$$Variable = \{topping_1, topping_2, beverage\}$$

$$Value = \{ham, pepperoni, tuna, onion, pineapple, mushrooms, beer, wine, water\}$$

We now want to define preferences which express the properties of desired variable assignments. In classical constraint programming, legal variable assignments are defined with Boolean expressions which draw a hard line between ‘good’ (true) and ‘bad’ (false) solutions. In our example, we define a constraint which restricts all variables to a certain domain:

$$domain : Valuation \rightarrow \{\text{false}, \text{true}\}$$

$$\begin{aligned}
&domain(val) = \\
&val(topping_1) \in \{ham, pepperoni, tuna\} \\
&\wedge\ val(topping_2) \in \{onion, pineapple, mushrooms\} \\
&\wedge\ val(beverage) \in \{beer, wine, water\}
\end{aligned}$$

(constraint for variable domains)

We could for example also define a constraint which requires the beverage to be wine when the first topping is ham:

$$hamAndWine : Valuation \rightarrow \{\text{false}, \text{true}\}$$

$$hamAndWine(val) = (val(topping_1) = ham) \rightarrow (val(beverage) = wine)$$

(order wine to ham)

Boolean constraints come to their limits when we want to express a preference like ‘beer is ok, but wine is better’. In this case, we need to assign different levels of acceptance to a valuation. These different levels are modeled with an ordered monoid, an abstract definition of a set of grades with a binary operation and a relation. The operation combines grades; it has to be associative and commutative and requires an identity element from the set of grades. The relation has to be a partial order and is used for comparing grades, i.e. finding out if one grade is better than another.

To sum it up, an ordered monoid is an algebra $(G, *, \leq, 1)$ with:

- carrier set G,
- an associative and commutative operation $_ * _ : G \times G \rightarrow G$
- a partial order $_ \leq _ \subseteq G \times G$
- and an identity element 1 ($\forall g \in G. g * 1 = g$)

Examples for monoids are natural or real numbers (with addition or multiplication) or Boolean values (with conjunction):

$$Nat_{+} = (\mathbb{N}, +, \leq, 0)$$

$$Real_{*} = (\mathbb{R}, *, \leq, 1)$$

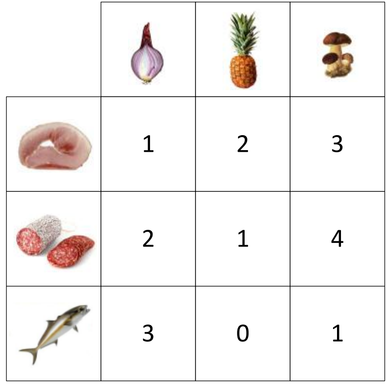
$$Bool = (\{\text{false}, \text{true}\}, \wedge, \rightarrow, \text{true})$$

These grades can now be used for rating variable assignments with a grade. A soft constraint over a monoid $(Grade, *, \leq, 1)$ is a function from valuations to grades:

$$SoftConstraint = Valuation \rightarrow Grade$$

Based on the order relation over grades, a soft constraint induces a partial order over valuations. This makes it possible to compare valuations and search for the best ones with the highest grades. There can be several best valuations having either the same grade or distinct grades which are incomparable among each other due to the partial order.

As an example, we define a soft constraint which expresses our preferences for topping combinations with natural numbers:



	1	2	3
	2	1	4
	3	0	1

FIGURE 4 TOPPING CONSTRAINT

Valuations with a high grade are preferred to valuations with a lower grade, e.g. ‘pepperoni with onions (grade 2) is better than tuna with mushrooms (1) but not as good as ham with mushrooms (3)’. In this case, the best combination would be pepperoni with mushrooms. Constraints can be defined over an arbitrary number of variables - although in our example we only use constraints over two variables since these can easily be displayed with a matrix.

Defining only a single soft constraint is rather senseless and makes it trivial to find the best valuation: soft constraints become expressive and complex when there are multiple constraints at the same time. Each constraint expresses a separate preference and several constraints can also be contradictory among each other, i.e. a valuation which is considered as good by one constraint can get a rather bad grade by another one.

Combining soft constraints over the same monoid can be done by combining their grades:

$$_ * _ : SoftConstraint \times SoftConstraint \rightarrow SoftConstraint$$

$$(s_1 * s_2)(v) = s_1(v) * s_2(v)$$

(combine two constraints by combining their grades)

As an example, we define another soft constraint which expresses our preferences for a beverage depending on the pizza´s first topping:

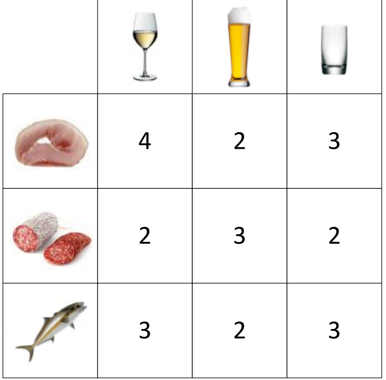


FIGURE 5 BEVERAGE CONSTRAINT

We can now combine this constraint with the constraint for topping combinations defined above and each valuation is rated with a grade which reflects both preferences. For example, using the monoid of natural numbers with addition, the following valuation is rated like this:

$$val(topping_1) = ham$$
$$val(topping_2) = onion$$
$$val(beverage) = wine$$

$$(topping * beverage)(val) = topping(val) * beverage(val) = 1 * 4 = 1 + 4 = 5$$

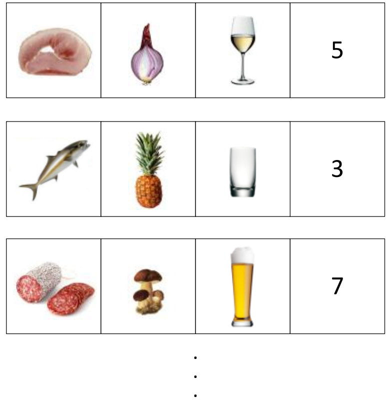


FIGURE 6 SOME VALUATIONS AND THEIR GRADE

Typically, one is interested in the maximal valuations, i.e. the valuations with the highest grades:

$$solve : SoftConstraint \to \mathrm{P}(Valuation)$$

$$solve(s) = \{v \in Valuation \mid \nexists v' \in Valuation. s(v') > s(v)\}$$

In our example, there are two optimal valuations:

FIGURE 7 VALUATIONS WITH THE HIGHEST GRADES

Soft constraints can also be combined in a much more expressive way; (12) introduces a very general way for combining constraints along with additional meta-preferences rating the constraints itself. This is realized by embedding the grades of several constraints into a single ordered monoid $(Rank, *, \leq, 1)$. For each constraint, an embedding function has to be defined which merges the constraint's grade with the global rank. Embedding functions are defined with a function space $Embed$:

$$Embed = Grade \times Rank \to Rank$$

The order of embedding has to be irrelevant, i.e. for all embedding functions $e, e' \in Embed$, $g, g' \in Grade$ and $r \in Rank$ the following has to hold:

$$e(g, (e'(g', r))) = e'(g', (e(g, r)))$$

Simple examples for embedding functions are injections into a Cartesian product of the single constraints' grade sets. The order over the Cartesian product can for example be defined by component or lexicographically, which already allows defining simple meta-preferences. Many more complex combinations can be defined using this technique and various meta-preferences can be expressed in a uniform way.

2.2 Approach: Coordinating Dynamically Changing Preferences

In our approach for composing music with soft constraints, we do not only want to solve a static constraint problem but rather want to coordinate system behavior with respect to preferences which are dynamically changing over time (e.g. based on user interaction). In general, we want to assign actions to actors (e.g. in our case notes to instruments):

$$Variable = Actor$$

$$Value = Action$$

$$Valuation = Actor \rightarrow Action$$

(assign actions to actors)

We use a discrete set of points in time at which an action should be assigned to each actor. We model this set with natural numbers which allows performing operations on times in a simple way (for example, given a time t, the preceding time is $t - 1$).

$$Time = \mathbb{N}$$

The comparison relation has to be compatible with the occurrence of the points in time in real time, e.g. $t_1 < t_2$ implies that t_1 has happened before t_2. The real duration between two points in time does not have to be constant; the assignment of actions to actors can also be triggered by some irregular event (e.g. user interaction). A very elegant and general approach for dealing with time is introduced in Leslie Lamport's well-known work on the ordering of events in a distributed system (14).

Dynamically changing preferences over action assignments are modeled with a set of soft constraints. A dynamic preference defines a distinct soft constraint for each point in time (denoted with an index):

$$preference_{Time} \in SoftConstraint$$

(a dynamic preference)

At each time, several dynamic (as well as static) preferences are combined to a single constraint problem. This can be done in various ways as described in the last subsection.

$$problem_{Time} \in SoftConstraint$$

Solving this problem yields one or several optimal solutions from which one has to be chosen (this can be done randomly since every solution has an optimal grade). This solution then assigns an action to each actor which satisfies the given preferences best:

$$solution_{t \,\epsilon\, Time} \in solve(problem_t)$$

$$action_{a \,\epsilon\, Actor, t \,\epsilon\, Time} = solution_t(a)$$

(action assignment with respect to preferences)

As an example, we now want to define preferences for ordering several pizzas on successive days. Besides our general preferences as defined above, we define an additional preference which is dynamically changing over time based on the last day´s pizza. This preference prefers toppings and beverages which were not ordered the day before by adding a penalty of minus one for each action which was chosen on the last day:

$$balancedDiet_t(val) = \sum_{actor \,\in\, Actor} \begin{cases} -1\; if\;\; action_{\,actor,t-1} = val(actor) \\ 0\; else \end{cases}$$

We combine this dynamic preference with the static preferences from the last section:

$$topping * beverage * balancedDiet_{Time}$$

This dynamic constraint problem yields several optimal sequences; one of them is visualized in the picture below. In this example, we define a dynamic soft constraint depending on previous actions. In addition to that, dynamic constraints can also be defined based on current events, e.g. the sensor readings of a user interface or any other external information.

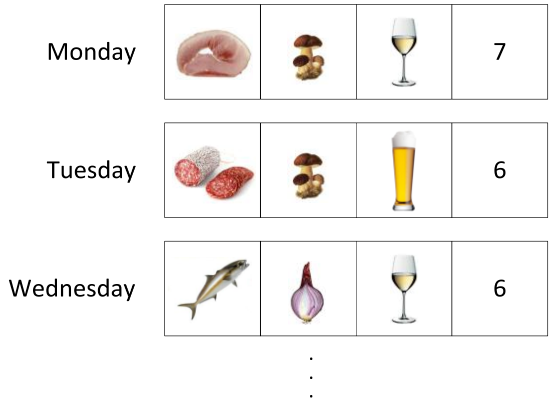


FIGURE 8 BALANCED DIET

2.3 Approaches for Solving Soft Constraints

There are several approaches and implementations for solving semiring-based soft constraints, which is an NP-hard problem. Discrete optimization problems have been intensely investigated since the 1950´s and several techniques are known for tackling them. Optimal solutions can be found or approximated for example with branch-and-bound algorithms, heuristics, dynamic programming approaches or greedy algorithms. In the area of constraint optimization, so-called consistency techniques have been investigated in the recent time.

Local consistency techniques are widely used in classical constraint programming and have been generalized for semiring-based soft constraint problems with certain properties (8). The basic idea behind these techniques is to iteratively eliminate inconsistency in subproblems until consistency of the full problem is obtained. This is done by transforming sets of constraints such that redundancy is removed and the problem´s search space gets smaller. Such transformation of a set of constraints is called ´constraint propagation´ and has to assure that the solutions of the problem remain unchanged. In (15), the concept of local consistency for soft constraints is generalized and adapted to a general framework for constraint propagation. The work (16) introduces local consistency techniques making it possible to solve additional types of problems. Recently, the concept of local consistency for soft constraints has been adapted for so-called constraint hierarchies (17). Another approach for solving soft constraints is dynamic programming: Dynamic programming is a technique for solving a problem by separately solving subproblems of it and combining their solutions to a solution for the whole problem. Dynamic programming can always be used for solving soft constraints without any condition; they can even be solved in linear time in some cases (8). In the dissertation (18), an approach for relaxing soft constraints is presented (19) and several algorithms for solving them are introduced (20). In the recent time, the combination of algebraic structures with lexicographic orders has been investigated in general: the work (21) shows under which conditions two partially ordered

domains can be combined to a new one that corresponds to the lexicographic order of those two which allows modeling additional types of scenarios.

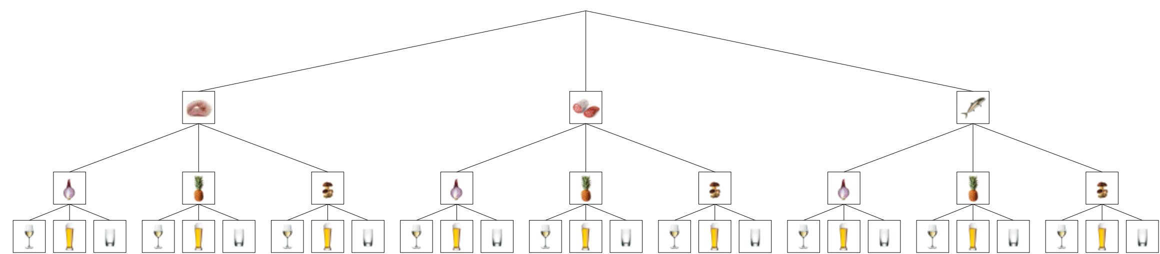


FIGURE 9 SEARCH SPACE IN THE EXAMPLE

Besides theoretical approaches for solving soft constraints, there are also several implementations. In (22), a local search framework is presented which adopts problem transformation and consistency techniques. Based on this framework, a prototype solver is implemented. The constraint programming system clp(FD) (23) for constraint problems with a finite domain has been extended by constraint semirings: clp(FD, S) (24) is based on a dedicated implementation resulting in good efficiency whereas softclp(FD) (25) is implemented on top of the mature clp(FD) library of SICStus Prolog. In (26), an implementation of semiring-based constraints is introduced for the concurrent constraint language Mozart. (27) presents a framework for designing soft constraint solvers with constraint handling rules (CHR). This language allows specifying constraint programming systems and propagation algorithms on a high level of abstraction. Constraint handling rules have been implemented in several programming languages, for example Prolog or C. In (28), Wirsing et al. present a framework for prototyping soft constraints in the rewriting logic language Maude. A branch-and-bound algorithm with several search optimizations is implemented and applied for optimizing parameters in software-defined radio networks. The solver used in the work at hand is based on this Maude prototype. It is realized in .NET with C# and directly implements the theory of monoidal soft constraints as defined above. The basic search algorithm and optimizations of this solver will be introduced in the next section; a short description of the .NET implementation can be found in section 6.2.2; for a more detailed description we refer to the original work (13).

2.4 A Solver for Monoidal Soft Constraints

The .NET solver used in this work is based on a verified Maude prototype (28) and implements the theory of monoidal soft constraints (12). It is realized with a typical branch-and-bound backtracking algorithm which enumerates and rates possible variable valuations in order to find an optimal solution. Several search optimizations are employed for pruning search paths such that it is guaranteed that no optimal solutions are omitted.

Soft constraints can be defined in two different ways: *functional soft constraints* are defined with an arbitrary function; *explicit soft constraints* with a discrete map from valuations to grades:

$$Map = \{M \subseteq Valuation \times Grade \mid \nexists g, g' \in Grade. g \neq g' \wedge (v,g) \in M \wedge (v,g') \in M\}$$

(maps which assign a unique grade to valuations)

Such map $M \in Map$ represents a soft constraint like this:

$$M(v) = \begin{cases} g \text{ if } (v,g) \in M \\ \bot \text{ else} \end{cases}$$

The solver enumerates valuations by combining all explicit constraints' map entries in a consistent way, i.e. such that every entry assigns the same value to a certain variable. Whenever a new constraint's map entry could be combined, the constraint's grade is embedded into an overall rank. Then, the solver compares this partial solution with existing total solutions: if it can be assured that all total solutions on this search path will always yield a lower grade, the path can be pruned. Typical embedding functions are for example injections into positions of a Cartesian product of grade sets $Grade_1 \times \ldots \times Grade_n$ which have the following form (inj_i is the i-th injection into a Cartesian product, π_i the i-th projection):

$$embed_i : Grade_i \times Rank \rightarrow Rank \; (i \in \{1,..n\})$$

$$embed_i(g,r) = inj_i(g * \pi_i(r), r)$$

The search algorithm iterates over all soft constraints and over all explicit constraints' map entries like this:

```
for each soft constraint
    if is explicit constraint
        for each map entry
            if is consistent with partial solution
                embed into partial solution
                if not pruning possible
                    store backtracking continuation
                    continue with the next constraint
        // complete map processed
        backtrack
    else if is functional constraint
        embed into partial solution
        if pruning possible
            backtrack

// all constraints processed, total solution found!
delete other solutions with lower rank
store solution
backtrack
```

At first, the solver initializes an empty partial solution with a minimal rank and no variables assigned. Then, all constraints are embedded into it like this: when processing an explicit constraint, the algorithm searches its map for the first valuation which is consistent with the current partial solution's valuation and embeds its grade into the solution's rank; a successive map entry (if existing) is stored as a backtracking continuation. Functional soft constraints compute their grade directly from the current partial solution's valuation. This new rank can then be compared with all existing total solutions and the search path can be pruned under certain conditions (see below). It is required that all variables from a functional constraint's domain have been assigned previously by an explicit constraint. In the simplest case, an explicit Boolean constraint enumerating only the variable's domain can be defined. When all constraints have been processed, a new total solution has been found: it is stored in a global list and all other existing total solutions with a lower rank are deleted. Then, the solver starts searching from the next backtracking continuation. If there are no continuations left anymore, the search is finished.

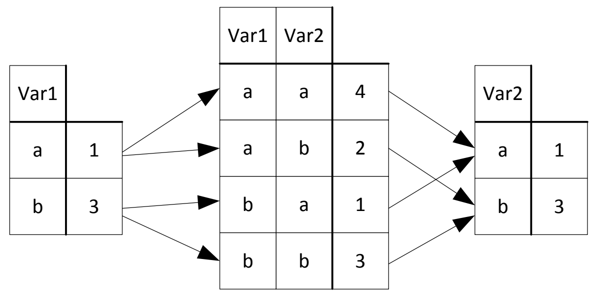


FIGURE 10 SEARCH PATHS FOR EXPLICIT CONSTRAINTS (ARROWS INDICATE CONSISTENT VALUATIONS)

The solver uses several search optimizations which are applicable under certain conditions. Proofs can be found in (13). A monoid $(G,*,\leq,1)$ is called *intensive* if and only if $\forall g, g' \in G\,.\, g > g * g' \vee g = g * g'$, i.e. the monoid's combination operation always yields a result which is not greater than its arguments. Let $P_{sub} \subseteq P \subseteq SoftConstraint$, i.e. P_{sub} is a subproblem of P. Then the following holds for every $v \in Valuation$ and $g \in G$: $g > P_{sub}(v) \Rightarrow g > P(v)$, i.e. the grade of a partial solution will never exceed that of a corresponding total solution. This allows pruning search paths with a grade that is already lower than any previously found total solution's grade. When an explicit constraint's map is in descendant order, it is then also possible to prune all remaining map entries (28). This can be extended for monoidal constraints with embedding functions: An embedding function is called intensive if it always computes a lower or equal rank: $e(g,r) \leq r$. Let $P_{sub} \subseteq P \subseteq SoftConstraint$ a problem with intensive embedding functions. Then the following holds for every $v \in Valuation$ and $r \in Rank$: $r > P_{sub}(v) \Rightarrow r > P(v)$.

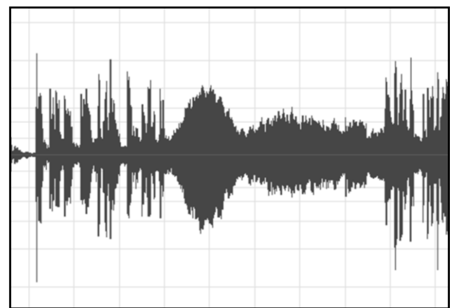
The structure of hierarchical problem combinations can allow additional divide & conquer optimizations. Proofs can again be found in (13). A typical combination injects each constraint's grade into a certain position of a Cartesian product of grade sets. The order over this Cartesian product can for example be defined by component or lexicographically. When intensive monoids are used, the corresponding embedding functions will also be intensive. When an order by component is used, it is possible to separately solve problems that do not share variables. When a lexicographic order is used, the top problem can be solved first. Then, the bottom problem can be solved with respect to the top problem's solutions. These divide & conquer optimizations directly break into the problem's exponential structure and can lead to highly reduced search times.

3 Music Theory

In this section, a short introduction to acoustics is presented with a focus on musical aspects which are relevant for this work:

- Sound
- Tonal systems
- Harmony

A great resource for further and more detailed information is the 'Springer Handbook of Acoustics' (29). Sound is always transmitted over a medium (e.g. air or water). In case of silence, the medium's particles are aligned rather homogenously. But whenever something moves within the medium it causes a disturbance in it. Like in a game of billiards, particles get accelerated and bump into other particles, which then again bump into others and so on. This way, the disturbance is propagated through the medium as a sound wave. From the point of view of an observer (e.g. an ear or a microphone), a sound wave is a change of pressure over time. The amount of pressure at a given time is called the amplitude; a sound wave can be represented as a function from time to amplitude. Sound waves are often illustrated as transverse waves with time being on the horizontal axis and amplitude on the vertical axis.



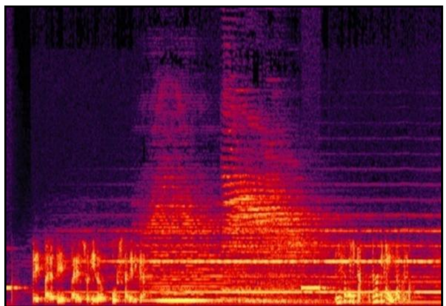


FIGURE 11 A SOUND WAVE AND ITS FREQUENCY SPECTRUM

It is difficult to imagine how such illustration of a wave actually sounds and indeed: the perception of sound is very different. Humans do not perceive sound directly as amplitude changes but rather on the more abstract level of 'rate of amplitude changes' (frequency, 'high' or 'low'). Sound waves are called cyclic (or periodic) when they consist of an always repeating pattern; they are called acyclic when no such pattern can be found. Acyclic sounds are perceived as 'noise', whereas cyclic sounds are perceived as 'tones' having a certain frequency. In the beginning of the 19th century, Joseph Fourier found out that every cyclic function can be decomposed into several of the simplest cyclic functions: sine waves with different frequency, amplitude and phase. The individual frequencies of a complex sound wave can be visualized in a

diagram with time being on the horizontal axis and frequency on the vertical axis (low frequencies are at the bottom). Each frequency's amplitude is indicated with a certain color (here, dark is low, red medium and yellow high).

The problem of decomposing a sound wave into its basic frequencies is called Fourier analysis. The human ear acts as a 'mechanical Fourier analyzer' and directly senses a sound's frequency structure: sound waves arrive at the ear, get bundled by the ear cup and arrive at the ear-drum. Then they are transmitted to the cochlea, a spiral structure which is filled with a fluid and has sensory hairs on it (it can be seen in the right of Figure 12). The spiral is very small in the center and becomes increasingly larger in the outer areas leading to different resonance properties: When a sound is transmitted to the cochlea, it gets in resonance on areas depending on the sound's frequencies, which is detected by the sensory hairs. High frequencies lead to resonance on the smaller areas in the cochlea's center, whereas low frequencies are located in the bigger outer areas. Thus, the nerve impulses arriving at the brain do already correspond to the sound's frequency structure.

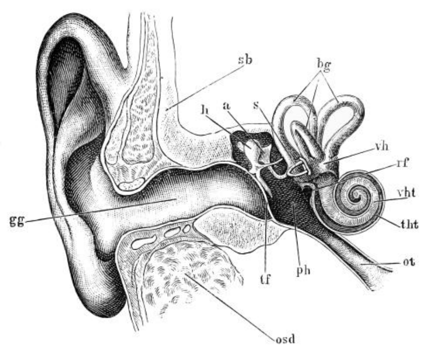


FIGURE 12 HUMAN AUDITORY SYSTEM

Many naturally generated periodic sounds consist of frequencies which have distinct ratios among each other. This is a result from the way many physical objects naturally oscillate: a sound source (e.g. a human's vocal cord or a guitar string) has certain resonance frequencies in which it tends to oscillate. The lowest such frequency is called the *fundamental frequency*; all higher frequencies are called *overtones*. When an overtone has a frequency which is in an integer ratio to the fundamental frequency, it is called a *harmonic* - and in fact, harmonic frequency ratios sound pleasant. Most melodic instruments mainly have harmonic overtones (e.g. plucked strings or blown pipes), but there are also instruments with many disharmonic overtones (e.g. drums or bells).

An oscillating string serves as a good example for demonstrating why many objects tend to oscillate in harmonics. The simplest oscillation is spread over the whole string, making it going up and down over its full length. This is the string's fundamental frequency:

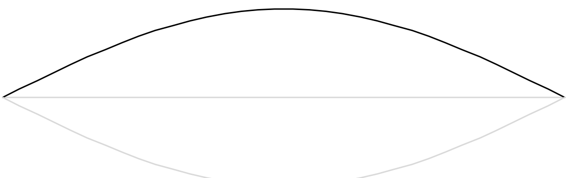


FIGURE 13 FUNDAMENTAL FREQUENCY

The second simplest possible oscillation divides the string in two halves, thus doubling the frequency. This is the string's first harmonic:

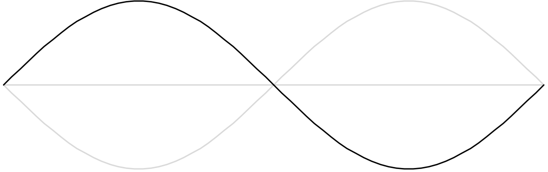


FIGURE 14 FIRST HARMONIC

There are infinitely many further harmonics, here are the next two:

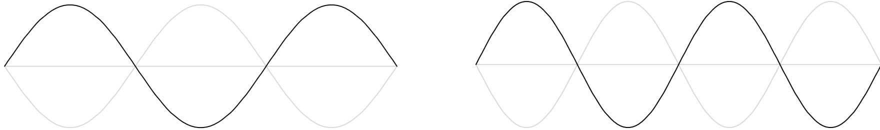


FIGURE 15 SECOND AND THIRD HARMONIC

Although every cyclic sound not being a sine wave itself is composed of many frequencies, it is mostly perceived as having only one pitch (the fundamental frequency) and a certain timbre (the overtone structure). When pressing a single key on a piano, only a single pitch is perceived - although it actually consists of many different frequencies at the same time. A sound's timbre is not perceived as several distinct frequencies but rather as the 'color' of the sound. The timbre can also evolve over time, for example a hammered string of a piano has many high frequencies in the beginning which decrease over time.

When several instruments play together, the problem of creating harmonic (and, if desired, disharmonic) frequency ratios comes up. Pythagoras was one of the first who experimented with harmonics and developed a tuning system for musical instruments. Unexpectedly, the problem turned out to be a very hard one which is still an area of research. A tuning system defines which frequencies (*pitches*) an instrument can play. Two goals are important: First, there should be many pitches which sound harmonic together and second, the number of pitches should be still manageable. Pythagoras developed a very natural way to construct pitches: He observed that there are certain *pitch classes* which are perceived somehow as 'the same'. These can be constructed by doubling or halving frequencies. For example, a pitch with 200 Hz is in the same pitch class as 400 Hz, 800 Hz, 1600 Hz and also 100 Hz (these are called *octaves*). When women and men sing the same song together, they typically sing in different octaves, but still the same pitch classes. Given a certain pitch, a tuning system should include all other pitches having the same pitch class. Pythagoras started with a certain pitch (e.g. 200 Hz). The first harmonic (400 Hz) is already in the tuning system since it belongs to the same pitch class, so the second one (600 Hz) seems a good choice to add. This defines a new pitch class: 300 Hz, 600 Hz, 1200 Hz and so on. By always taking the second harmonic (multiplying the frequency with 3), new pitch classes can be generated (the next would be 450 Hz, 900 Hz, 1800 Hz,...). Unfortunately, this construction does not terminate and creates infinitely many pitch classes; another problem is that the pitches are not separated by the same ratios. In so-called western music (which should not be confused with country music), 12 ordered pitch classes are used: C, C#, D,…, B which can be played in different octaves. The distance between pitches is called an *interval*; the interval with a distance of 7 steps corresponds to the second harmonic used in Pythagoras' construction (commonly known as a fifth). When playing ascending fifths on a keyboard, the sequence C, G, D, A,…. is generated reaching all 12 pitch classes and finally returning at another C again (the well-known 'circle of fifths'). Unfortunately, when constructing the pitches as Pythagoras, the frequency of the last 'C' ends up being slightly higher. This difference with a ratio of 1.01364 is known as the 'Pythagorean comma' and is a fundamental problem: It is inherently not possible to find a non-trivial tuning system where every fifth (exactly: the interval that should be a fifth) has a frequency ratio corresponding to the second harmonic (a pure fifth). Today, a system is commonly used which divides all pitches with the same ratio (equal temperament) and thus distributes the comma equally over all pitches. This makes it possible to play in every different tonal scale close to pure intervals but it is not possible to play an actually pure fifth or any other pure interval (except the octave). Many other tuning systems distribute the comma only over certain tonal scales, thus getting pure intervals on some scales but also very detuned intervals on others. Some instruments (e.g. bowed strings or electronic instruments) can play pitches with continuous frequency, and many string ensembles indeed dynamically adapt their tuning to get pure intervals. There are also automatic approaches for use in electronic music instruments (e.g. Hermode tuning (30)) which continually change the tuning system in order to tune intervals as pure as possible. This technology is already integrated in several products like Apple's audio workstation software Logic and some synthesizers. A nice resource for further information on tuning systems is Ross W. Duffin's book

‘How Equal Temperament Ruined Harmony’ (31). This book claims that much ‘older’ music heard today (e.g. Baroque music) is not being played as intended by the composer since different tuning systems were used at the time of composition. Duffin argues that many composers purposely used tonal scales which sound bad and disharmonic to create tension which is often later resolved by scales with pure intervals.

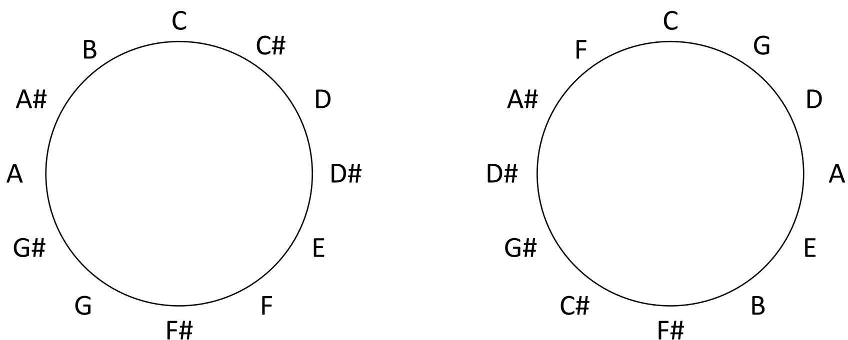


FIGURE 16 PITCH CLASSES IN WESTERN MUSIC (LEFT: SEQUENTIAL ORDER, RIGHT: CIRCLE OF FIFTHS)

Tuning is difficult, but in practice equal temperament is good enough most of the time. As mentioned above, there are 12 different pitch classes in western music. The following table lists all possible intervals (distances) between pitches within one octave along with the frequency ratio of the corresponding pure interval:

Distance	Interval	Ratio
0	Unison	1:1
1	Minor second	16:15
2	Major second	9:8
3	Minor third	6:5
4	Major third	5:4
5	Fourth	4:3
6	Tritone	64:45
7	Fifth	3:2
8	Minor sixth	8:5

9	Major sixth	5:3
10	Minor seventh	16:9
11	Major seventh	15:8
12	Octave	2:1

TABLE 1 INTERVALS

It can be observed that intervals with simple ratios sound more pleasant than those with complex ratios. Fifth have of ratio of 3:2 and are the most harmonic interval (besides the trivial intervals unison and octave), whereas the tritone is the most dissonant interval.

Another important concept in music theory is *tonal scales*. A tonal scale is a subset of pitch classes in certain order and is defined with a starting pitch class (the *tonic*, e.g. C) and a list of intervals (the *mode*, e.g. major). The mode defines which other pitch classes beside the tonic belong to the scale. Each mode has a certain character; the most common modes in western music are major and minor but there are also many other modes (e.g. church or Jazz modes). As an example, the major mode is defined with the following intervals:

Unison – Major second – Major third – Fourth – Fifth – Major sixth – Major seventh

The C-major scale contains the following pitch classes:

C – D – E – F –G –A – B

Another example for a major scale is E-major which contains the following pitch classes:

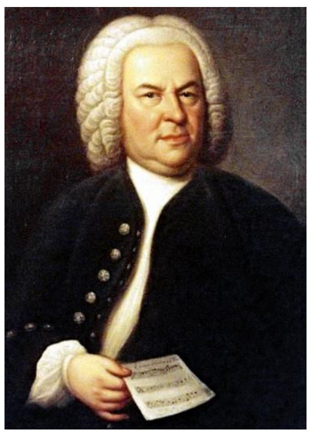
E - F# - G# - A – B – C# - D#

4 ALGORITHMIC COMPOSITION

Composing music with formal rules has a long tradition. Although most composers do not explicitly use algorithms - the techniques they use can often be modeled with formal systems. Godfried Toussaint discovered that even many traditional ethnical rhythms are generated by the so-called 'Euclidean Algorithm' (32). It is very unlikely that the 'composers' of these rhythms knew about this algorithm – the more surprising is the large number of rhythms from all over the world generated by it. In the cited work, Toussaint lists over 100 traditional rhythms from e.g. India, Brazil, Greece or Turkey. In general, the Euclidean algorithm solves the problem of distributing a number of k units to a number of n intervals as even as possible; it is named after Euclid's similar algorithm for computing the greatest common divisor of two integers. This algorithm can be found in Euclid's Elements and was called by Donald Knuth the 'granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day'. Besides generating rhythms, it is also used for timing neutron accelerators or drawing straight lines in graphics software. When it comes to rhythm, we want to assign a number of notes (k) to a number of time units (n) as even as possible. The rhythms generated by the algorithm are called 'Euclidean rhythms' and are denoted with E(k,n). Rhythmic sequences can be represented as a binary sequence of notes ('x') and silence ('.'), for example [x . x .] corresponding to the trivial Euclidean rhythm E(2,4). A very famous example for an Euclidean rhythm which is widely distributed is E(3,8) corresponding to [x . . x . . x .]. This rhythm is known as the 'Tresillo' in Cuba or as the 'Habanero' in the USA but it can also be found in West African music. The rhythms can start at any position; E(3,8) for example also generates the rhythms [x . x . . x . .] and [x . . x . x . .]. Many more - surprisingly complex – rhythms are presented in Toussaint's work, e.g. E(7,12) = [x . x x . x . x x . x .] from Ghana or E(15,34) = [x . . x . x . x . x . . x . x . x . . x . x . x . x . . x . x .] from Bulgaria.

Many composition techniques in Western art music also make use of certain algorithmic rules. Especially music from the era of Baroque is written in a very formal way; the composition of many typical musical forms from that time is constrained by strict rules. Developing already during the Renaissance, a key element in Baroque music is the so-called 'counterpoint'. Before that time, polyphonic music was typically composed with one leading voice and several accompanying voices which do not play any melodic role at all. In contrast to this, counterpoint means that every voice makes sense as an independent melody itself – nevertheless, they should also make sense as a whole when playing them together. A typical contrapuntal genre is a 'fugue' which was brought to a high level of perfection by Johann Sebastian Bach. When composing a fugue, many rules have to be applied: at first, a single voice starts playing and introduces the so-called 'subject'. Then, while the first voice continues to play, a second voice repeats the subject transposed by a certain interval (often a fifth). This repetition is called the 'answer' and can sometimes be slightly modified in order to e.g. stay within a certain tonal scale. In this way, all other voices start playing until all of them are playing simultaneously. This first part of a fugue is called 'exposition'. As the fugue continues, the melodic material introduced in

the exposition is often used again either in its original form or in an altered version. There are many different ways of altering melodic material and most of them can be expressed in a procedural way. For example, the length of a melody is often modified by applying a certain constant factor to it, thus lengthening (augmentation) or shortening it (diminution). Typical alterations of a melody´s pitches are for example transposition, retrograde (playing it backwards) or inversion (ascending intervals become descending and vice versa).



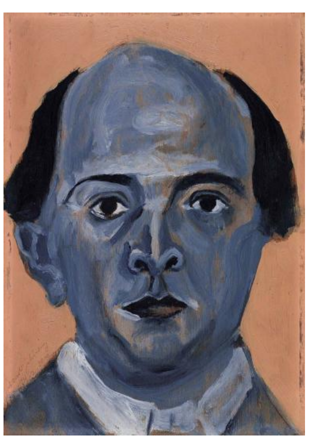


FIGURE 17 JOHANN SEBASTIAN BACH, ARNOLD SCHOENBERG

After the era of Baroque (1600 – 1750), formal rules became less important but, nevertheless, music from the Classical (1750 – 1820) and Romantic period (1820 - 1900) was still composed with respect to a variety of rules and conventions regarding for example tonality or the structure of musical forms (e.g. the well-known 'sonata form'). In the late 19th century, customary tonal conventions were questioned and more and more dissonant music was composed. At the turn of the 20th century, tonality was finally abandoned and the first completely atonal music was composed in the environment of the so-called 'Second Viennese School' around Arnold Schoenberg. Having broken with all tonal rules and conventions, Schoenberg surprisingly introduced a very strict formal method for composing music in 1921: the so-called 'twelve-tone technique' (also often referred to as 'dodecaphony'). The basic idea behind this technique is to use every pitch class of the chromatic scale exactly once until it can be used again, which ensures equally frequent occurrence of each pitch class and avoids tonal centers like in a traditional tonal scale. Taking up typical composition techniques from the Baroque era, Schoenberg also often applied alterations (inversion, retrograde…) to the twelve-tone series in his compositions. The twelve-tone technique was designed as a method for composing atonal music and is hence only concerned with tonal aspects; there are no rules dealing with all the other aspects in a composition (for example rhythm, dynamics or timbre). Throughout the 20th century, various approaches for composing both tonal and atonal music have been developed which are based on or inspired by Schoenberg´s technique. The common

idea behind these approaches is to divide certain musical aspects (e.g. tonality or rhythm) into a series of values (e.g. note pitches or rhythmic divisions) which serve as the most basic building blocks for a composition. These basic values are then subject to all different kinds of generative procedures or manipulations. Composition techniques which are based on this approach of working with one or several series of basic elements are subsumed under the term 'Serialism'. The initial contributions to serial composition techniques beyond the twelve-tone technique were made by Olivier Messiaen in the late 1940´s. He did not only work with a series of note pitches (C, C#…) like Schoenberg but also used for example several series of rhythmic divisions (16th, 8th…), dynamic indications (piano, forte…) or articulations (staccato, legato…). Messiaen´s 'Mode de valeurs et d'intensités' from 1949 is one of the key compositions in Serialism: it is based on several series of independent parameters which are connected among each other such that each note pitch is always played with the same duration and dynamic. 'Mode de valeurs…' is considered as the origin of serial composition techniques and served as inspiration for many others. Pierre Boulez, a student of Messiaen, became one of the central figures of contemporary music in the second half of the 20th century and wrote elaborate compositions based, amongst others, on the serial technique. Boulez is also known for his electronic compositions and as one of the leading conductors of the 20th century.

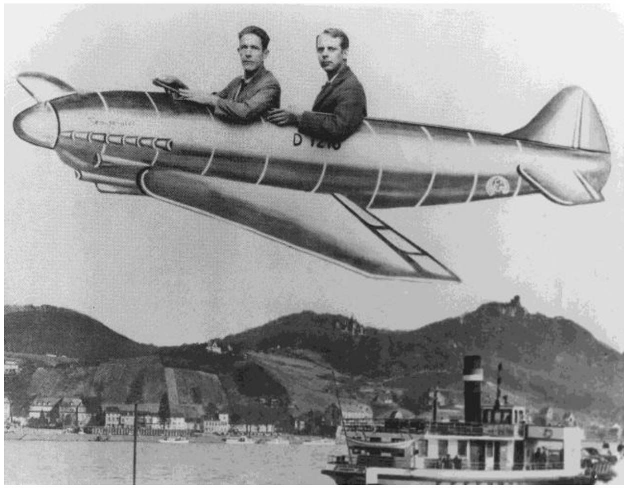
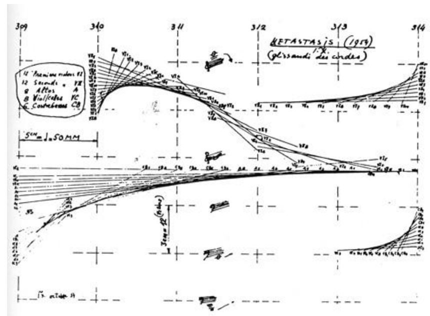


FIGURE 18 JOHN CAGE, KARLHEINZ STOCKHAUSEN

Another key person of music in the 20th century is John Cage, a student of Schoenberg, who is known for example for his aleatoric compositions, philosophical contributions or as a pioneer of so-called 'Fluxus' art. Cage used several algorithmic procedures for assembling some of his compositions (and even speeches). Above all, so-called chance operations are characteristic for Cage´s work. Besides several sources for pseudo-random events like star charts, Cage mainly used the Chinese 'I Ching' divination system for making decisions without his own influence.

Karlheinz Stockhausen was one of the most influential and controversial composers in the 20th century. Being a student of Messiaen, his way of composing was strongly influenced by serial techniques and most of his compositions are constructed in a very formal and precise way. Stockhausen was a pioneer in the composition of electronic music which enabled him to accurately control every parameter of his music. Iannis Xenakis, another student of Messiaen, composed music based on various existing mathematical models with no direct connection to music, for example from the areas of statistics or game theory. One of his most popular works is the piece 'Metastasis' based on a geometrical function called the 'hyperbolic paraboloid'. He also worked as an architect (amongst others under Le Corbusier) and used the same function for designing the Philips pavilion at the World's Fair 1958 in Brussels. In the 1960´s, Xenakis was one of the pioneers in computer-assisted composition techniques. As early as 1977, he designed an interactive computer composition tool called UPIC with a user interface based on a graphics tablet which allowed to compose music in real-time by drawing compositions on it.



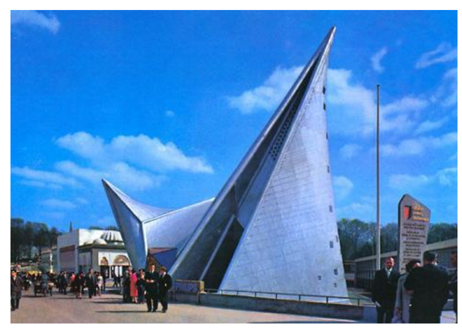


FIGURE 19 METASTASIS SCORE, PHILIPS PAVILLON (IANNIS XENAKIS)

This chapter can of course only give a short und very incomplete overview of the history of formal composition techniques. Above all, we want to convey the idea that the composition of music can always be regarded as algorithmic up to a certain extent, ranging from traditional percussion music over the era of Baroque to the 20th century. Nevertheless, algorithmic composition has never been as significant and self-contained as in the 20th and 21st century. A great book about art music in the 20th century is 'The Rest is Noise' (33) from Alex Ross. Today, algorithmic composition techniques have become a common and popular method for contemporary composers; they are used in a variety of styles, especially in art music and experimental electronic music.

Today, the variety of approaches which have been used for algorithmic composition reads like an overview of artificial intelligence techniques: music has been composed with probability models, generative grammars, artificial neural networks, transition networks (e.g. Petri nets), genetic algorithms, cellular automata, rule-based systems or reasoning techniques.

Furthermore, a variety of mathematical concepts with no direct connection to music have been used, for example self-similarity or chaos theory. The book ‘Algorithmic Composition: Paradigms of Automated Music Generation’ (34) from Gerhard Nierhaus gives a comprehensive and up-to-date overview about this field. In the recent time, techniques subsumed under the term ‘live-coding’ have been of interest: algorithms are written and modified ‘on-the-fly’ while they are running and generating music (e.g. in an improvisational performance). The audience is often able to observe the evolution of the programming code on a projector and some artists also use live-coding techniques to generate additional visual effects. The article (35) covers historical precedents, theoretical perspectives and recent practice of the live coding movement. The work (36) introduces a project focused on the design and development of tools to structure the communication between performer and musical process, e.g. by hardware controllers. In the work (37), a new visual language for live-coding is introduced based on the combination of so-called ‘pattern generators’. Other interesting recent works related to algorithmic composition investigate for example topological representations of musical objects (38), the use of cloud computing techniques for distributed composition systems (39) (40) or concrete applications like automatic sonification of classical Chinese poetry (41).

Electronic musical instruments play an important role in the history of algorithmic composition: originally built to be played like ‘real’ acoustic instruments, the border between sound design (i.e. shaping the timbre of a sound) and composition (i.e. assembling sounds to a piece of music) soon became fuzzy. Léon Theremin´s ‘Theremin’ from 1919 features a contactless user interface based on two proximity-sensing antennae: the distance of the player´s hand to the first antenna continually controls the pitch of an oscillator; the distance to the other antenna controls the amplitude. The Theremin was used for example in various film scores, in popular music (e.g. by the Beach Boys) or in art music (e.g. in compositions from Dmitri Shostakovich). Friedrich Trautwein´s ‘Trautonium’ from 1930 is played with a manual which is somehow similar to a keyboard but allows continuous control over pitch. A later version of the Trautonium, the so-called ‘Mixtur Trautonium’ introduced a novel approach for sound synthesis which makes use of so-called ‘undertones’. The instrument´s manual controls the frequency of one master oscillator and several so-called ‘frequency dividers’ derive additional harmonic oscillations with a lower frequency (undertones or ‘subharmonics’). The divisor of each frequency divider can be adjusted by the player and the outputs of all dividers can be summed up in a variable mixture, making it possible to realize a large variety of different timbres. The Trautonium was used for example for the sound effects in Alfred Hitchcock´s ‘The Birds’ as well as in several compositions e.g. by Paul Hindemith.

FIGURE 20 EQUIPMENT FROM THE WDR STUDIO FÜR ELEKTRONISCHE MUSIK

In 1951, the 'Studio für elektronische Musik' was founded in Köln by the Northwest German Broadcasting institution (today known as the WDR). This studio was the very first of its kind and served as an experimentation platform for composers such as Stockhausen or Boulez as well as technical pioneers such as Werner Meyer-Eppler. The studio was equipped with state-of-the-art devices like noise generators, oscillators, filters and various effect units; some of them were custom developments for the studio. Technical devices which are today common to every electronic musician were first used in a musical context at the 'Studio für elektronische Musik'. At that time, there was no other place with a comparable technical and personal environment. In the 1960's, synthesizers became commercially available from companies like Moog, Buchla or Roland. These early systems were designed based on an open modular architecture with several independent components that can be connected freely among each other, exchanging audio signals (e.g. generated by an oscillator) or control voltages. Technically, there is of course no difference if a synthesizer is controlled by a human performer playing on a keyboard or any other control voltage, e.g. a certain algorithmic procedure or a stochastic process. Especially the system developed by Don Buchla has many modules available which are suited well for algorithmic composition applications, for example random value generators. Buchla´s 'Source of Uncertainty' module provides several different sources for random voltages, from static white noise to a versatile voltage generator based on a probability distribution which offers a high level of control. It is not only possible to manually control the parameters of the distribution: the parameters themselves can also be automatically modulated with external control voltages.

FIGURE 21 BUCHLA SERIES 200E MODULAR SYNTHESIZER, RANDOM VOLTAGE GENERATOR

Modular synthesizers based on dedicated hardware are still popular and valued for their unique sound and direct user interface. However, they do not offer the enormous amount of flexibility as provided by software solutions. Today, musical programming environments are very widespread among composers who work with algorithmic techniques. In 1957, Max Matthews developed the musical programming system MUSIC which provides functionality for sound synthesis on the one hand and the composition of sounds to a piece of music on the other hand. MUSIC can be seen as the ancestor of many other related musical programming environments which are introduced in section 5.4.3 as work related to our approach for interactively generating music. At the present time, the tools of choice for many composers are programming environments with a graphical user interface, making them also accessible to people without programming skills. These visual programming languages like Max/MSP (which is named after Max Matthews) are conceptually very similar to a modular synthesizer: several 'objects' can be connected to a 'patch' which realizes a certain functionality. Objects are the basic building blocks and can perform various tasks, from simple arithmetic operations to complex functionality like sound synthesis algorithms. The basic programming paradigm is based on dataflow: an object can have several inputs and outputs for processing respectively generating streams of data like audio or musical information (e.g. notes or rhythmic events). In addition to the standard library of objects, Max/MSP can be extended with custom objects written for example in C; there is also a large online community providing many objects, patches and tutorials. Max/MSP integrates with several hardware controllers as well as platforms for custom sensor devices, making it possible to develop interactive applications with a specialized user interface, for example custom tools for musical performance or interactive installations.

5 APPROACH: COMPOSING MUSIC WITH SOFT CONSTRAINTS

In this section, we present our approach for interactively composing music in real-time with *soft constraints* (42). We provide a general framework which allows generating music by defining 'preferences' that express how the music should sound. As an example, a preference for a single instrument could be 'fast notes with a high pitch'. Besides preferences for single instruments, it is also possible to define preferences for the coordination of multiple instruments. Coordination preferences typically involve harmonic or rhythmic aspects, for example 'play together in a similar rhythm in rather consonant intervals'. Most of these preferences are rather 'soft': they do not define an exact result and do not need to be strictly obeyed all of the time. In fact, there are often preferences which are even concurrent among each other; for example, a single instrument's rhythmic preferences could be in conflict with global rhythmic preferences. A good technique to deal with such soft and concurrent problems is 'soft constraints'. In previous work, we developed a first prototype for interactively composing music with soft constraints based on Nintendo Wii controllers (13) (43). We participated in conception, formal modeling, implementation and evaluation of this prototype system. We also ported the existing soft constraint solver from Maude to the .NET framework. In this work, we develop a general framework applicable to a variety of different applications. When experimenting with our previous system, it turned out to be hard to generate melodies fitting to a certain musical style (e.g. country music): Manually identifying and defining the preferences for a given style is a large amount of work and therefore not a very scalable approach for integrating many styles. To accommodate this, we now make use of a machine learning approach: musical styles can be defined by playing notes instead of writing code. We developed a general musical transition model which can be trained with existing melodies and 'remembers' the occurrence of notes in certain metric positions as well as transitions between notes. When such model has been trained, it can be used to generate additional preferences for an instrument which express 'how well the music fits the training data', thus making it generate melodies similar to the training melodies. Again, soft constraints showed up to be a very appropriate problem class here and make it possible to accommodate the concurrent preferences of the trained model, the user's interaction and the coordination between multiple instruments.

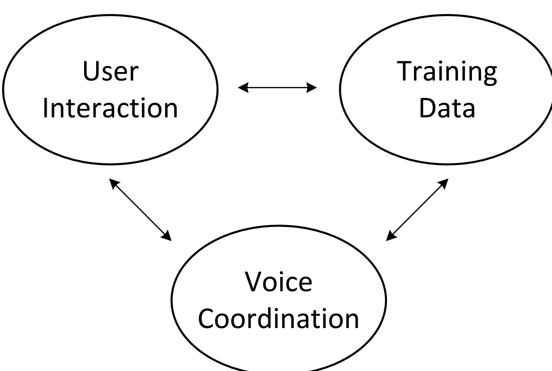


FIGURE 22 EXAMPLE FOR CONCURRENT PREFERENCES

In the next subsections, we introduce a formal model of our approach depending on the theory of monoidal soft constraints from section 2. At first, basic concepts of music theory are defined. In the subsequent section, we present our general approach for composing music with soft constraints. Then, we introduce the musical transition models which are used to train preferences in order to integrate several musical styles in our system. At last, we compare our approach to related ones.

5.1 Music Theory

A key concept in music theory is a sound's *pitch* corresponding to its frequency. On this abstraction level, we are only interested in the fundamental frequency – although almost every sound is composed of multiple frequencies as indicated in section 3. Every pitch belongs to a certain *pitch class*: in so-called western music there are 12 different pitch classes (C,…,B), but the model presented here is not restricted to this and would also allow to work with different systems.

$$PitchClass = \{c, c\#, d, \dots, b\}$$

(pitch classes in western music)

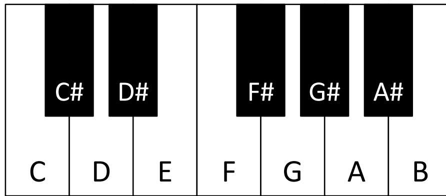


FIGURE 23 PITCH CLASSES ON A KEYBOARD

A pitch class can be played in different *octaves*, which are defined with integer numbers. Theoretically, there are infinitely many octaves - but the human auditory system can only perceive the octaves from 0 to 9 (in most cases, even less). Combining a pitch class with an octave number index defines a concrete pitch (e.g. c_4 or c_7).

$$Octave = \mathbb{Z}$$

(octave numbers)

$$__ : PitchClass \times Octave \rightarrow Pitch$$

(combine a pitch class and an octave to a pitch, e.g. $f\#_4$)

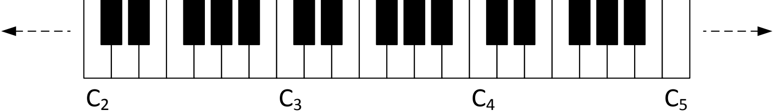


FIGURE 24 PITCH CLASS C IN DIFFERENT OCTAVES

Intervals between pitches are an important concept for dealing with melody and harmony; they represent the distance between pitches. Each interval has a certain grade of harmony, for example a fifth sounds very pleasant whereas a tritone is very disharmonic. Section 3 includes a list of the basic intervals used in western music.

$$Interval = \{unison, minorSecond, majorSecond, \ldots\}$$

(intervals)

$$[_,_] : Pitch \times Pitch \rightarrow Interval$$

(interval between two note pitches, e.g. $[e_3, g_3] = minorThird$)

$$_ < _ : Interval \times Interval \rightarrow \{true, false\}$$

(compare intervals)

A *tonal scale* defines a subset of pitch classes in a certain order. Each scale has several *stages* defining this order, they are denoted with roman numbers (I, II,...). The most common scales in western music consist of seven stages (major, minor…) but there are also common scales with more (e.g. Jazz scales) or less stages (e.g. the pentatonic scale). In many cases, the restriction to a tonal scale is rather fuzzy and often pitch classes are being played which do not belong to the scale (sometimes referred to as 'blue notes'). To deal with this, we make use of the concept of augmented or diminished stages. An augmented stage is one halftone higher than the original stage; e.g. I+ (the plus denotes augmentation) in C-major is C#.

$$Stage = \{I, I+, II, II+, \dots\}$$

(tonal stages in a scale)

A scale is defined with a *mode* (major, minor…) and a *tonic* (the starting pitch class). A mode is defined as a function which associates an interval with each stage. For example, stage III is a major third in major scales and, vice versa, a minor third in minor scales.

$$Mode = \{major, minor, \dots\} = Stage \to Interval$$

(musical modes, e.g. $major(IV+) = tritone$)

Each mode can be played starting from a different pitch class which is called the scale's tonic:

$$_{_} : PitchClass \times Mode \to Scale$$

(tonal scales with tonic and mode, e.g. c_{major} or $f\#_{major}$)

Given a tonal scale, each stage corresponds to a certain pitch class. The conversion between stages and pitch classes is done with the following two functions:

$$_in_ : Stage \times Scale \to PitchClass$$

(interpret a stage in a scale, e.g. $II\ in\ g_{minor} = a$)

$$_in_ : PitchClass \times Scale \rightarrow Stage$$

(interpret a pitch class in a scale, e.g. $a\ in\ g_{minor} = II$)

5.2 Musical Soft Constraints

This section introduces the core framework for composing music with soft constraints in real-time based on several musical interfaces (which we often just call instruments). The constraints express 'how the music should sound' at a given time. We distinguish two kinds of constraints:

- Constraints expressing user preferences for a single instrument (e.g. 'play high notes')
- Constraints coordinating multiple instruments (e.g. 'play harmonic intervals')

Both kinds of constraints can also be generated dynamically, typically based on user interaction. At certain times, each instrument is being asked to state its current preferences with one or several constraints. The constraints from all instruments are extended with global coordination constraints and combined to a single constraint problem. This problem is then being solved and the resulting notes are played. The time intervals at which this happens can either be fixed (e.g. every 16th note) or triggered by user interaction. In general, we define a discrete set of times with natural numbers such that sequential times are modeled with sequential numbers:

$$Time = \mathbb{N}$$

The actual duration between two times can be variable; we are only interested in the correct order. Times can be compared with the common operators ($<, \leq, \dots$) or modified with any other operator on natural numbers. For example, for a given time t, the preceding time is $t - 1$.

We model the task of generating music for several musical interfaces by assigning certain actions to them. In our current implementation, we use three kinds of actions: start a new note (with a given pitch), hold a note and pause. It is also possible to add more parameters to an action, for example a note's velocity or aspects of its timbre – but this of course also increases the search space and could make it impossible to solve the resulting constraint problem in a real-time context. Actions can also consist of a sequence of several notes but this slows down the system's reaction time to user interaction (the shorter the actions, the faster the reaction). However, it might make sense for some user interfaces or non-real-time systems.

Each instrument can have several voices which are used as the constraint problem's variables. Polyphonic instruments (which can play several notes at the same time) need to have an according number of voices.

$$Variable = Voice = \{v_1, v_2, v_3, \dots\}$$

(voices from all instruments)

The problem values are the actions which a musical interface can perform (e.g. play notes):

$$Value = Action = \{pause\} \cup NoteAction$$

(voice actions)

A certain type of actions is note actions, which are always associated with a pitch:

$$NoteAction = NoteOn \cup Hold$$
$$play : Pitch \rightarrow NoteOn\ [ctor]$$
$$hold : Pitch \rightarrow Hold\ [ctor]$$

$$pitch : NoteAction \rightarrow Pitch$$
$$pitch\big(play(p)\big) = p$$
$$pitch\big(hold(p)\big) = p$$

(note actions)

The preferences for the music are defined as soft constraints which rate action assignments for the voices with a grade:

$$MusicalSoftConstraint = (Voice \rightarrow Action) \rightarrow Grade$$

(expressing preferences over voice assignments with soft constraints)

Dynamically changing preferences can be expressed as a set of constraints with one constraint corresponding to each time (denoted with an index):

$$dynamicConstraint_{Time} : (Voice \rightarrow Action) \rightarrow Grade$$

(dynamically changing preference)

Each instrument states its preferences based on the current user interaction. This is realized as a soft constraint which rates action assignments for the instrument's voices. We call these constraints sensor constraints since they are typically based on the interface's sensor readings.

$$sensorConstraint_{Voice,Time} : (Voice \rightarrow Action) \rightarrow Grade$$

(a constraint expressing preferences derived from user interaction)

A sensor constraint controls certain aspects of an instrument's pitch and rhythm. When it comes to pitch, we often take a certain tonal range around a given 'mean pitch' which is derived from user interaction. The pitches in this range can have the same grade but it is also possible to rate the pitches based on a radial function around the mean pitch (e.g. a Gaussian distribution). It is of course also possible to define any other preference for pitch, for example 'pitch class c – no matter in which octave'. Generally, we define a constraint which rates actions (e.g. with a real number) based on their note pitch: $pitchConstraint_{Voice,Time} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$

Controlling the rhythm of an instrument is typically done by dynamically balancing the grades over time for the several types of actions. For example, if an instrument should rather do nothing at a given time, the pause-action will have a high grade compared to the note actions. It is also possible to define certain rhythmic patterns by shifting the grades in favor of starting notes at desired rhythmic positions. If there is no direct and deterministic connection between certain user actions and resulting notes, it proved to be useful to give an instrument an 'energy-account' representing how many notes it still can play. When the instrument should play something, a certain amount of energy is added to its account. The grade of the instrument's note actions is always defined based on the current amount of energy such that high energy yields high grades for note actions and, vice versa, low energy prefers pause actions. Whenever the instrument actually plays a note, its energy gets lower. This way, it is possible to continually control 'how fast the instrument should play'.

$$energy_{Voice,Time} \in [0, maxValue \in \mathbb{R}]$$

$$energyConstraint_{Voice,Time} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$$

$$energyConstraint_{v,t}(val) = \begin{cases} energy_{\,v,t} \;\; if \; val(v) \in NoteAction \\ maxValue - energy_{v,t} \;\; else \end{cases}$$

(a constraint based on an 'energy account')

A sensor constraint which we often use in our applications is a combination of the pitch constraint and the energy constraint defined above. Using these constraints, it is possible to generate music with only two parameters - 'pitch' and 'energy' - which proved to be simple to extract from user interaction on the one hand but also expressive on the other hand. Intuitively, these parameters continually control the note pitch (high/low) and the speed (fast/slow) at which an instrument should play. Interfaces based on these parameters are easy to play because they require only few musical skills (e.g. making exact rhythmic movements) – nevertheless, they provide much control over the music in a very direct way with immediate musical feedback.

$$energyPitchConstraint_{Voice,Time} : (Voice \rightarrow \; Action) \rightarrow \mathbb{R}$$

$$energyPitchConstraint_{v,t}(val) = energyConstraint_{v,t}(val) * pitchConstraint_{v,t}(val)$$

(constraint for user interfaces based on 'energy' and 'pitch')

The coordination preferences between multiple instruments are also expressed with soft constraints. There is no conceptual difference to the instruments' constraints; anyhow, we distinguish them to make their special role clear and also keep our implementation modular to instruments and coordination preferences. We will now give several examples of typical constraints for coordinating instruments. If several voices should 'play together' in the same rhythm, we can define a constraint like this:

$$together : Action \times Action \rightarrow \mathbb{N}$$

$$together(a_1, a_2) = \begin{cases} 1 \; if \; \exists A \in \{NoteOn, Hold, \{pause\}\}. \, a_1 \in A \wedge a_2 \in A \\ 0 \; else \end{cases}$$

$$togetherConstraint_{V \subseteq Voice} : (Voice \rightarrow \; Action) \rightarrow \mathbb{N}$$

$$togetherConstraint_V \; (val) = \sum_{v_a \in V} \sum_{v_b \in V} together(val(v_a, v_b))$$

(voices should make the same type of actions)

An often used constraint maximizes the amount of 'harmony' between several voices (but it is of course also possible to do the opposite and prefer disharmonic music). Intervals between two notes can be rated according to their harmony; if harmonic intervals are preferred, fifths or fourths get high ratings and the tritone interval gets the worst rating:

$$harmony : Pitch \times Pitch \rightarrow Grade$$

(harmony between two note pitches)

This rating can be extended to a whole set of notes; it is then very easy to define a constraint which maximizes the number of harmonic intervals:

$$harmonyConstraint : (Voice \rightarrow \; Action) \rightarrow Grade$$

$$harmonyConstraint(val) = \sum_{v_a \in Voice} \sum_{v_b \in Voice} harmony\big(pitch(val(v_a)), pitch(val(v_b))\big)$$

(harmony in a set of note actions)

Leonhard Euler developed a function called 'gradus suavitatis' with rates the harmony of musical intervals. Given an interval's integer frequency ratio p/q, Euler's function returns low values for simple and harmonic intervals and, vice versa, high values for complex and dissonant intervals. With r_i being the prime factors of the least common multiple of p and q, the gradus suavitatis is defined as:

$$G(p, q) = 1 + \sum (r_i - 1)$$

(Euler's gradus suavitatis)

The unison interval (1:1) has the best possible grade of 1; the tritone (45:32) a bad grade of 14. Extending an interval by one or several octaves changes the grade of an interval (for better or worse); e.g. a fifth (3:2) has a grade of 4 but playing a fifth plus one octave (3:1) has a better grade of 3. This is not desirable in every case since we often have other constraints for the pitch which we do not want to be in conflict with a harmony constraint. The gradus suavitatis is also not 'commutative' to pitch classes, e.g. $[c_3, g_3]$ is a fifth (grade 4) but the interval $[g_3, c_4]$ gets a worse grade of 5 since it is a fourth (4:3).

Another harmony rating which proved to be useful in our applications depends on work from the composer Howard Hanson (44). Intervals between pitches (modulo octaves) are being assigned to classes with 'equal' harmony. These classes are then ordered according to their consonance as follows:

P = Perfect fifth, perfect fourth

M = Major third, minor sixth

N = Minor third, major sixth

S = Major second, minor seventh

D = Minor second, major seventh

T = Tritone

When we assign a fixed grade to each class, we get a harmony rating which is very adequate for our needs: the rating does not depend on octave transpositions and is commutative to pitch classes in every case, because commutative intervals are always in the same class.

Another useful constraint using Boolean values as grades restricts notes to a tonal scale. For example, $tonalScale_{c_{major}}\big((v_1 \mapsto play(f\#_4)), \ldots\big)$ is false, since $f\#$ does not belong to c_{major}.

$$tonalScale_{Scale} : (Voice \rightarrow Action) \rightarrow \{\text{true}, \text{false}\}$$

(only pitches in a tonal scale)

Many more coordination preferences can be expressed with soft constraints, for example preferring certain global rhythmic accents or making notes fit to given harmonic progressions.

To sum it up, at each time $t \epsilon Time$, a constraint problem is generated based on the single instruments' constraints and the global constraints. These constraints can be combined in various ways to one single problem. This problem is then being solved and one solution is chosen which defines an action for each voice ($action_{Voice,Time}$).

5.3 Trainable Musical Models

As mentioned in the introduction of this section, it is hard to manually identify and define rules for certain musical styles like country music or hip-hop. Nevertheless, we want to be able to generate music fitting to lots of different styles. To achieve this, we use a machine learning technique which is a far more scalable approach for integrating many styles in an easy and fast way. There exist several related algorithmic composition techniques based on machine learning, but to our knowledge this is the first approach for training melodies which can be interactively 'played' in real-time. Our approach is based on a custom transition model which represents sequences of events aligned upon a structured metric grid. Intuitively, the model represents:

- how often an event occurs at a certain metric position and
- how often other events follow this event at this position

Following typical terms from the closely related area of probability models, the 'events' are called states. The discrete metric positions (representing 'time') are called steps:

$$State$$

$$Step = \{0, \dots, n\}$$

(sets for states and steps)

In each step, each state has a certain weight for a given voice. This weight represents how often the state occurs at the given step:

$$stateWeight_{Voice} : Step \times State \to \mathbb{R}$$

(timed state weights)

The transitions between states at a given step are represented with the following function. The first two arguments define the original step and state – the third argument defines the next state. Transition weights are always defined for subsequent steps; the state in the third argument is implicitly assumed to be on the next state.

$$transitionWeight_{Voice} : Step \times State \times State \rightarrow \mathbb{R}$$

(timed transition weights)

The following figure visualizes a timed transition model with three steps and states. State weights are visualized with black circles: the bigger the circle, the higher the weight. The transition weights are visualized with arrows (a thicker arrow indicates a higher weight). When the model is untrained, all weights are the same. Training the model modifies the weights; the right picture visualizes a trained model with shifted weights.

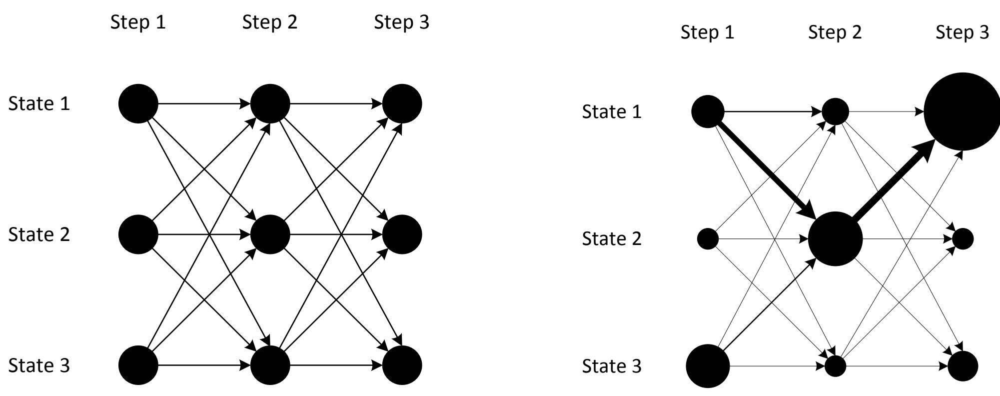


FIGURE 25 TRANSITION MODEL VISUALIZATION (LEFT: EMPTY MODEL, RIGHT: TRAINED MODEL)

When it comes to training melodies within our framework, it seems a good idea at first sight to directly use the existing actions as states. However, there would be a little disadvantage: if note pitches are directly used as states, it is not possible to play a model in another tonal scale. Thus, we do not use note pitches directly but rather use abstract stages in a tonal scale. States are then defined as follows:

$$State = NoteState \cup \{pause\}$$

(musical states)

$$NoteState = NoteOnState \cup HoldState$$

$$noteOnState : Stage \rightarrow NoteOnState\ [ctor]$$
$$holdState : Stage \rightarrow HoldState\ [ctor]$$

(note states)

When generating a constraint from our model, we need to convert an instrument's action to the corresponding state used in the model. This is done with the following function which makes use of a global tonal scale ($currentScale \in Scale$):

$$getState : Action \rightarrow State$$

$$getState(a) = \begin{cases} noteOnState(pitch(a)\ in\ currentScale)\ if\ a \in NoteOn \\ holdState(pitch(a)\ in\ currentScale)\ if\ a \in Hold \\ pause\ else \end{cases}$$

(get the model state corresponding to an action)

Now, we define a constraint which expresses 'how well an action matches the data represented in the model'. Given the last step and the last actually executed state (the state corresponding to the last action chosen by the constraint solver), we can compute a total weight for each state on the subsequent step. The simplest way to do this is by just summing up the transition weight and the step weight itself:

$$totalWeight_{Voice} : Step \times State \times Step \times State \rightarrow \mathbb{R}$$

$$totalWeight_v(lastStep, lastState, step, state) \\ = transitionWeight_v(lastStep, lastState, state) \\ + stateWeight_v(step, state)$$

(total weight)

The constraint itself for a certain voice $v \in Voice$ is constructed based on the last step, the last action and the current step. Given a valuation's action for this voice, the constraint returns the total weight for this action's corresponding state:

$$modelConstraint_{Voice} : (Step \times Action \ \times Step) \to ((Voice \to Action) \to \mathbb{R})$$

$$modelConstraint_v(lastStep, lastAction, step)(val) \\ = totalWeight_v(lastStep, getState(lastAction), step, getState(val(v)))$$

(construct the soft constraint)

This constraint performs well when there is much training data available and when it is distributed over all steps. However, we sometimes also want to generate music with only few training data. For example, if we just use one single melody to train a model, we get steps with only one trained event and many steps without any trained note at all. This would lead to determinism on the one hand and pure randomness on the other hand. To accommodate this, we define an additional structure over the metric grid: Rhythmic structures in music are often composed of several 'similar' basic units (in many cases, these structures are also self-similar). Our basic idea is to not only consider the current step for generating a constraint but rather all steps with respect to a certain hierarchy. When there is no or only few training data at a certain step, similar steps can be taken into account for the constraint. The following illustration shows the typical structure of a 4/4 meter and indicates which metric positions are often considered as similar. The blue divisions indicate similarity for steps 1 and 3 (with decreasing hierarchy from ½ to 8th). Steps separated by an even and large note length (preferably a power of two) are more similar than others; for example 1 is very similar to 3 (this holds of course also for 2 and 4 or any other steps separated by a half note). Steps 1 and 2 are also similar – but not as much as 1 and 3 since this similarity is lower in the hierarchy (4th). Similarity for other steps is defined the same way; e.g. the first 16th after 2 is very similar to the first 16th after 4.

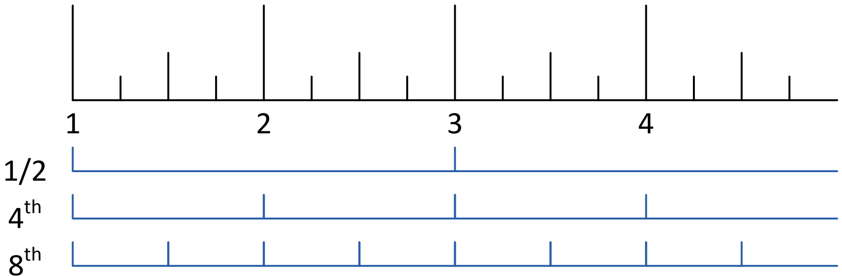


FIGURE 26 EXAMPLE FOR METRIC SIMILARITY IN A 4/4 METER

This is of course just an example of metric similarity – there are many other meters and possible ways for defining similarity (even for the same meter). In general, we define metric similarity as a function from two steps to their distance expressed in real numbers:

$$similarity : Step \times Step \rightarrow \mathbb{R}$$

(similarity of metric positions)

When generating a constraint with respect to metric similarity we now take all existing steps into account. Instead of using a state's weight only at the current step, we sum up the state's weight at all steps scaled by the corresponding similarity. This way, similar steps have a strong influence whereas, vice versa, dissimilar steps change only little. When a constraint should be generated for an untrained step, all states' weights will be equal at this step – in this case, the similar steps will now make the difference and avoid total randomness. However, when generating a constraint for a trained step, the similar steps' weights will only have little influence since they are scaled by lower values. This makes it possible to already play a model with only few training data and nevertheless generate melodies resembling the training data. The constraint is then defined just like above with this variant of $totalWeight$:

$$totalWeight'_{Voice} : Step \times State \ \times Step \times State \rightarrow \mathbb{R}$$

$$\begin{aligned} totalWeight'_{v}(lastStep, lastState, step, state) \\ = transitionWeight_{v}(lastStep, lastState, state) \\ + \sum_{otherStep \epsilon Step} (similarity(step, otherStep) \\ * \left(stateWeight_{v}(otherStep, state)\right)) \end{aligned}$$

(total weight with respect to metric similarity)

5.4 RELATED WORK

In this section, we discuss work related to our approach for algorithmically composing music. The main contribution of our approach is the employment of expressive soft constraints à la Bistarelli et al. (8). At first, we will take a look at the employment of constraint programming in musical applications. Then, we will introduce approaches for imitating musical styles. At last, we present general approaches for developing interactive music applications.

5.4.1 CONSTRAINTS AND MUSIC

Many approaches are known where classical constraint programming is employed in a musical context. Especially in the field of musical harmonization, constraints are widely used: automatic musical harmonization deals with the problem of creating arrangements from given melodies with respect to certain rules. For example, a typical automatic harmonization problem is to generate a four-part arrangement of a fixed melody based on several rules e.g. from the era of Baroque. Most such rules state incompatibilities (i.e. things that are not allowed), so constraints are a very appropriate and natural technique for dealing with this problem. Constraints allow defining *vertical rules* restricting simultaneous notes (e.g. 'do not play the tritone interval') as well as *horizontal rules* restricting successive notes (e.g. 'do not play parallel fifths').

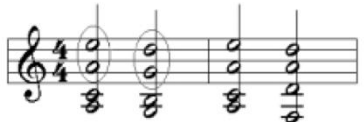


FIGURE 27 VIOLATION OF THE 'PARALLEL FIFTHS RULE' (45)

The actual rules for various musical eras are well-known and have been described in detail by many music theorists, e.g. by J. J. Fux in 1725 who formulated precise rules for counterpoint (46) or by A. Schoenberg who wrote a treatise considering both tonal as well as atonal music (47). The automatic harmonization problem has been investigated in numerous works and to our knowledge constraints have been employed in this field since the late 1970's. Early works in the area of constraint-based automatic harmonization are for example from L. Steels (7), B. Schottstaedt (48), K. Ebcioğlu (49) or C. P. Tsang (50). F. Pachet and P. Roy made a detailed survey on musical harmonization with constraints (45). This work gives a short introduction to the problem of automatic music harmonization and refers to relevant works in this area. In the recent time, the authors of this survey also made several contributions themselves: They developed a problem-solving technique for constraint problems which uses different hierarchy levels to efficiently solve problems (51). In (52), Pachet gives insight into his work at the Sony Computer Science Laboratory Paris and sketches out several scenarios where constraints are employed in multimedia systems. Besides automatic harmonization, constraints are also

explored to be used in other music-related areas: The MusicSpace project (53) gives listeners control over the spatial arrangement of sound sources. Constraints are employed to ensure several invariants, for example keeping a constant level of loudness. Another application of constraints is an approach for searching for songs in a large database and building playlists (54). Constraints are employed for modeling user preferences like ‘no slow tempos’ or ‘at least 30% female-type voice’.

FIGURE 28 AUTOMATIC HARMONIZATION OF THE FRENCH NATIONAL ANTHEM (55)

There is a survey (56) which compares generic constraint programming systems for modeling music theories. M. Laurson’s system PWConstraints (57) allows describing and solving musical problems based on PatchWork (58), a general-purpose visual language with an emphasis on producing and analyzing musical material. Arno (59) is a program for computer-assisted composition which extends the composition environment Common Music (60) by means of constraint programming. The Strasheela system from T. Anders (61) provides a general framework for composing music with constraints based on the Oz programming language. This language is also used in the experimentation platform COMPOzE (62). Strasheela has been extended with real-time capabilities (63) and was employed in a system which automatically generates a counterpoint to a melody played in real-time. Pazellian (64) provides an easy way to control musical performance aspects such as dynamics or pitches which are constrained such that users with no musical training are able to obtain musically sensible results. A method for creating a Markov process that generates sequences satisfying one or several constraints has been patented (65). These constrained Markov processes have been used to generate song lyrics that are consistent with a textual corpus while being constrained by rhythmic and metric aspects (66). Weaker notions of soft constraints have been employed in several systems: these approaches provide classical hard constraints as the basic paradigm and additional soft preferences can be added by some special extension, e.g. a global cost function. (67) introduces an environment for musical constraint solving adapted to contemporary composition in the visual programming language OpenMusic. This approach is based on constraints expressed in

logical form which can additionally be associated with a cost function. So-called ‘value ordering heuristics’ have been used to model soft rules, e.g. in PWConstraints (57) or Strasheela (61): this technique allows defining constraints in a certain order. Depending on this order, certain constraints can be omitted in order to avoid overconstrained situations.

Our approach is the first to employ expressive soft constraints à la Bistarelli et al. for composing music. This elegant theory of soft constraints is based on abstract algebraic structures (semirings rsp. ordered monoids) and is general enough to model many other approaches. Many different types of preferences can be expressed in a uniform way, from simple Boolean constraints to complex concurrent optimization goals. This generality and expressiveness also extends to our framework for composing music based on soft constraints.

5.4.2 Style Imitation

Approaches for imitating musical styles are also related to our work. Typical techniques for dealing with this problem are based on statistical models or musical grammars. The problem of imitating styles is also closely related to the problem of classifying styles since in many cases the same kind of model can be used.

Statistical models have been used for modeling musical styles since the 1950´s, for example by R. Pinkerton who trained higher-order Markov models with a corpus of nursery rhymes and generated new melodies from them (68). To name a few more works out of many, Markov models have for example been employed for analyzing and synthesizing music in the contrapuntal style of the composer G. P. da Palestrina (69) and hidden Markov models were used to classify folk music from different countries based on a corpus of several hundred melodies from Ireland, Germany and Austria (70). A very general approach for representing a stochastic process is a technique called ‘universal prediction’ which does not depend on a specific model; it has been applied for learning musical styles in (71). There are numerous other works where statistical techniques are used for imitating musical styles. In (72), several basic techniques for generating music from statistical models are discussed and relevant works in this area are introduced. A general overview about musical applications of statistics is given in the book ‘Statistics in Musicology’ (73).

‘Grammatical inference’ deals with the problem of finding the syntactic rules of an unknown language: In (74), grammatical inference algorithms are employed to learn stochastic grammars for musical styles. These grammars can be used to stochastically synthesize new melodies as well as to classify existing melodies. In (75), two methods for learning musical styles are described and compared. Both perform analyses of musical sequences and then compute a model from which new variations can be generated. The first approach computes a model based on ‘incremental parsing’, a method derived from compression theory. The second one uses ‘prediction suffix trees’, a learning technique initially developed for statistical modeling of

complex sequences with applications in linguistics and biology. GenJam (76) uses a genetic algorithm to automatically generate Jazz solos. Several populations of 'melodic ideas' are used to play solos over the accompaniment of a standard rhythm section. While playing, a human mentor can give feedback to the system which is used to derive fitness values for the melodic ideas. Using these fitness values, GenJam applies genetic operations to its populations and tries to improve them. In a later version (77), GenJam was modified such that it does not require a human mentor giving feedback anymore. This is accomplished by seeding an initial population of melodic ideas from a database of published Jazz licks and employing an intelligent crossover operator to breed child licks that tend to preserve the musicality of their parents. Evolutionary music composition is discussed in (78): Several evolutionary algorithms for composing music are described and two distinct approaches to the evaluation of generated music are discussed: interactive evaluation based on a human mentor's judgement and autonomous evaluation of generated musical material by the system itself.

Our approach for modeling musical styles has its focus on interactivity. Other approaches employ more complex techniques based e.g. on higher-order models. We use a rather basic musical transition model, but are able to train and play it in real-time. We introduce a novel combination of musical transition models and constraint-based rules: to our knowledge, there is no other approach which uses transition models to generate soft constraints which can then again be combined with other constraints, reflecting e.g. user interaction or general musical rules. This way, training data can be integrated into constraint-based systems in a uniform way within a single framework.

5.4.3 Interactive Music Systems

There are many systems for interactively generating music in real-time. Numerous performance devices and interaction paradigms have been explored, e.g. based on interactive tables, body movements or global positioning. In this section, we will focus on generic approaches; systems related to concrete applications of our framework will be introduced in the corresponding sections.

Musical programming environments are very popular for developing interactive music systems; e.g. for live performances or public installations. Above all, Max/MSP (79) is very widespread. It is based on an object-oriented approach and provides many pre-defined modules for generating and processing audio streams and control data. These modules can be connected among each other with a graphical user interface which makes the system accessible to people without programming skills. However, it is also possible to integrate custom modules written in e.g. C, C++ or Java. Interactive systems can be realized with a large number of compatible hardware controllers or platforms for custom sensor devices. Very similar systems based on the same visual programming approach are Pure Data (80) and SuperCollider (81); both are published under an Open Source license.

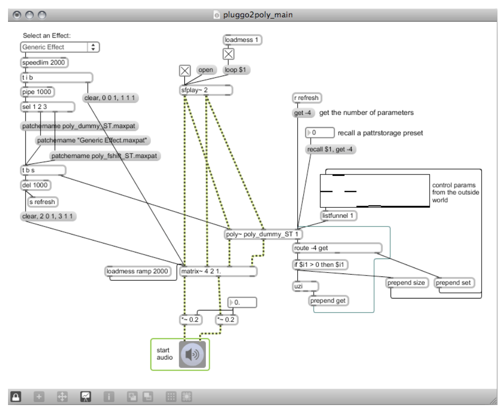


FIGURE 29 MAX/MSP PATCH

A slightly different approach is taken by modular software synthesizers such as Reaktor (82), Nord Modular (83) or Origin (84); the first one is a pure software product, the latter two are based on custom hardware with dedicated digital signal processors (DSP) and additional hardware controls. These systems also provide a graphical user interface running on a desktop computer which allows connecting modules, but – in contrast to low-level programming environments - their focus is more on sound synthesis rather than interaction or algorithmic composition. However, this is of course no inherent restriction; it is also possible to use modular software synthesizers for these kinds of application. Especially Reaktor is designed very open and allows integrating custom modules.

There are also textual musical programming languages which can be used in interactive applications. In 1957, Max Matthews developed MUSIC at Bell Labs: MUSIC was one of the first programming environments for musical applications and can be seen as the ancestor of many others. There exist a large number of derivate environments which were developed for example at Princeton University, the IRCAM center or the MIT. CSound (85) is an indirect descendant of MUSIC which is widely used for composition and performance of music. It is capable of synthesizing and processing sound and can be used in real-time applications; however, CSound is primarily designed as a tool for sound synthesis and composition rather than interactive applications. ChucK (86) is a textual programming environment having its focus on real-time synthesis, composition and performance. It provides a new time-based, concurrent programming model and allows to program ‘on-the-fly’, i.e. a program can be altered or extended while it is being executed. This makes it possible to perform and improvise music by writing code on stage (‘live coding’). Impromptu (87) is another musical environment for programming on-the-fly; it is based on Scheme (a Lisp dialect).

The Continuator (88) from F. Pachet combines style imitation and interactivity. The purpose of this system is to allow musicians to extend their technical ability with stylistically consistent, automatically learnt material. Based on a statistical model, the system is able to learn and generate musical styles as continuations of a musician's input, or as interactive improvisation backup which makes new modes of collaborative playing possible. In (89), several modes of interacting with the Continuator are discussed; experiences with children are presented in (90).

Our framework provides the generality and expressiveness of soft constraints and allows specifying and implementing interactive music systems in a very declarative way. Many common techniques for algorithmic composition can be modeled respectively integrated, for example classical constraints, concurrent optimization goals or musical transition models. The framework is based on an algebraic model which provides insight into all relevant functionality on a high level of abstraction, including musical rules as well as user interaction and musical training data. There exist related interactive systems based on machine learning techniques, but to our knowledge our approach is the first which allows training melodies than can be interactively 'played' in real-time, i.e. it is possible to generate melodies based on training data that can also be flexibly shaped based on user interaction.

6 FRAMEWORK DESIGN AND IMPLEMENTATION

In this section, we introduce the design and implementation of our framework for composing music with soft constraints. It is designed very modular and allows integrating various musical interfaces and coordination preferences. The implementation was developed entirely in the .NET framework with C# which allows running it on target platforms like Windows (PC, embedded, mobile…) or the Xbox 360 gaming console.

The framework is implemented in a straightforward way based on the theory introduced in the last section. It consists of four main components: *Music* models basic concepts of music theory (notes, intervals, scales…) and provides an infrastructure for sending and receiving notes and metric information (internally or using external MIDI connections). *Soft Constraints* allows modeling and solving problems with soft constraints; it is being described in detail in (13). Based on the latter two, *Musical Soft Constraints* makes it possible to compose music with soft constraints. Using this component, *Musical Model* implements our approach for training musical preferences. We will now define functional and non-functional requirements for the framework and then give a detailed insight into the design and implementation of each component along with many practical examples. At last, we evaluate the implementation: We assess its quality based on the non-functional requirements and present the results of performance tests.

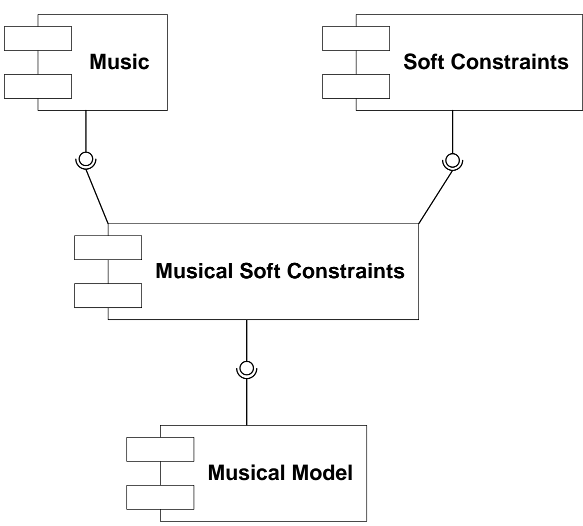


FIGURE 30 FRAMEWORK COMPONENTS

6.1 Requirements

The primary objective of this framework is to implement our approach for composing music with soft constraints. The core *functional requirements* are specified by the formal model from the previous section. To sum it up, the framework has to implement music theory, musical soft constraints and trainable musical models. An external component for modeling and solving problems with soft constraint (13) has to be integrated. A basic requirement is to provide a general infrastructure for communicating notes, controller data and metric information internally and with external MIDI equipment. Furthermore, the framework has to model high-level concepts of music theory like note pitches, pitch classes and scales. Based on this, an infrastructure for defining, combining and solving musical soft constraints has to be provided. This infrastructure should be open and highly extensible on the one hand but, on the other hand, also implement often used functionality like optimization cycles or common constraints. In addition to that, the framework has to implement musical models and basic functionality for training and playing them.

Of equal importance are *non-functional requirements*: As a matter of course, the framework has to be *reliable* and all specified functionality should be implemented accurately and correct. Above all, total system crashes have to be avoided in any case. The *efficiency* of the implementation is also of high importance: It should be possible to generate music for several musical interfaces simultaneously and devices with limited computing power (e.g. mobile phones) should suffice to run applications built on the framework. Besides the overall consumption of resources (processing power, memory etc.), the timing of events has to be accurate and tight. We make high demands on *usability*: The framework should have a clear structure and there should be a straightforward and documented process for developing applications with it. The framework should also be *maintainable*, making it easy to integrate new functionality. *Portability* is also desirable: the framework should not be limited to only a single platform. Instead, it should be possible to develop applications for a large range of platforms, for example gaming consoles, mobile devices or embedded devices.

6.2 Framework Components

We will now introduce the implementation of each component in detail.

6.2.1 Music Theory

This component implements basic concepts of music theory. The classes `Pitch`, `PitchClass`, `Octave` and `Interval` are implemented as described in the theory along with many helper functions.

PitchClass
+Number : int
+Interpret(in octave : Octave) : Pitch
+PitchClass(in number : int)

Pitch
+Number : int
+PitchClass : PitchClass
+Octave : Octave
+TransposeUp(in interval : Interval) : Pitch
+TransposeDown(in interval : Interval) : Pitch
+Pitch(in number : int)

Octave
+Number : int
+Interpret(in pitchClass : PitchClass) : Pitch
+Octave(in number : int)

Interval
+Number : int
+Interval(in number : int)
+Interval(in pitch1 : Pitch, in pitch2 : Pitch)

FIGURE 31 PITCHES, PITCH CLASSES, OCTAVES AND INTERVALS

Each of these classes is defined with an integer number which uniquely identifies it. It is possible to instantiate them with this number but there are also various other ways (for example by getting an octave from a pitch). Many static variables are provided in order to be able to obtain for example common pitch classes or intervals by their name instead of their number (e.g. `PitchClass.C` or `Interval.MINOR_THIRD`). As an example, we show how to combine a pitch class with an octave and get the interval between two pitches:

```
// combine a pitch class and an octave to a concrete pitch
Pitch p1 = PitchClass.C.Interpret(Octave.OCTAVE_3);

// instantiate another pitch (with its MIDI note number)
Pitch p2 = new Pitch(56);

// get the interval between these pitches
Interval i = new Interval(p1, p2);
```

The classes `Stage`, `Mode` and `Scale` allow dealing with tonal scales. Stages are defined with a number and can furthermore be regular, augmented or diminished. Modes define a sequence of intervals associated with stages which makes it possible to convert stages to intervals and the other way round. Scales can be constructed with a starting pitch class (the tonic) and a mode. Their main purpose is to convert stages and pitch classes in both directions. There are also many helper functions, for example for checking if a pitch class belongs to a tonal scale.

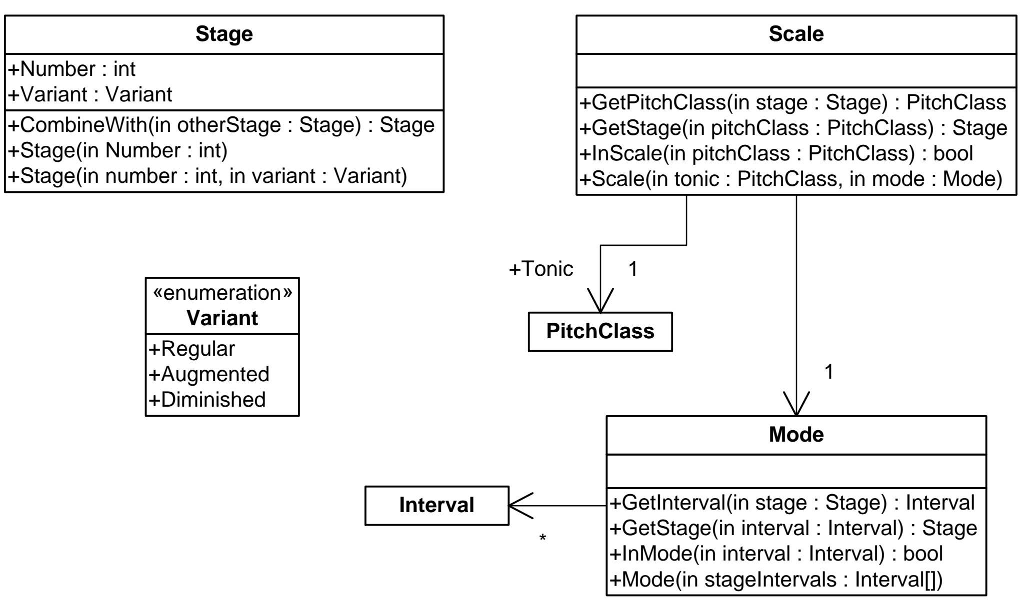


FIGURE 32 TONAL SCALES

Here is a small example for dealing with tonal scales:

```
// instantiate a c major scale
Scale scale = new Scale(PitchClass.C, Mode.MAJOR);

// get the third stage in c major (e)
PitchClass pitchClass = scale.GetPitchClass(Stage.III);

// check if 'e' belongs to c major (yes)
bool b = scale.InScale(pitchClass);

// get the stage of c# in c major (I+)
Stage stage = scale.GetStage(PitchClass.C_SHARP);

// check if this stage is augmented (yes)
b = (stage.Variant == Variant.Augmented);

// check if c# belongs to c major (no)
b = scale.InScale(PitchClass.C_SHARP);
```

When it comes to sending and receiving notes, we decided to use an event-based infrastructure. There are two types of note events: Note-on events start a note with a given pitch and velocity (representing the note's loudness) and note-off events stop a note.

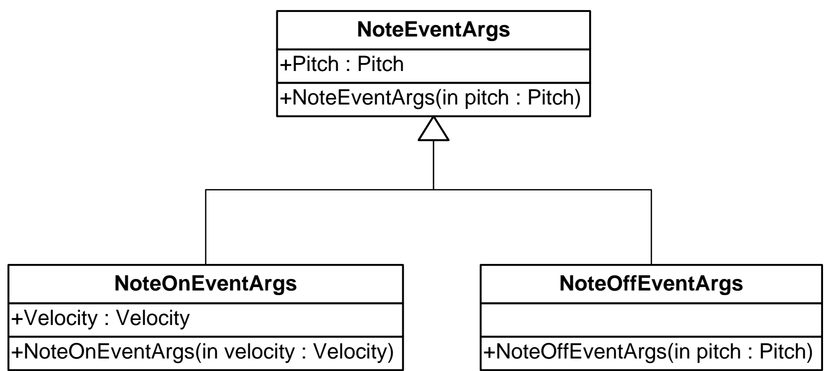


FIGURE 33 NOTE EVENTS

Using .NET's elegant event handling concept, we define interfaces for classes which send and receive notes. Note that it is possible to connect multiple event handlers to one source as well as to connect multiple sources to one event handler.

```
public interface NoteSender
{

    event EventHandler<NoteEventArgs> SendNote;

}

public interface NoteReceiver
{

    void ReceiveNote(object sender, NoteEventArgs e);

}
```

For convenience, there is a class `Instrument` (implementing `NoteSender`) which provides functions for directly starting or stopping individual notes without having to manually instantiate and send note events every time. Furthermore, `Instrument` also keeps track of all currently playing notes and allows to for example stopping all of them at once.

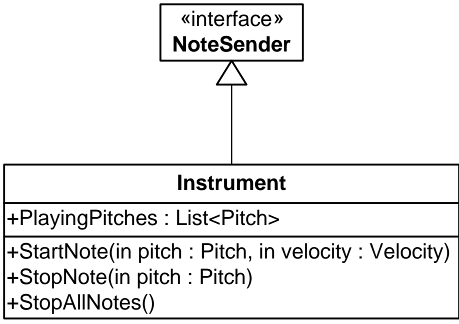


FIGURE 34 INSTRUMENT

Communication with external MIDI devices is realized using the C# MIDI Toolkit (91). This library is encapsulated in our framework using two classes, `MIDIIn` and `MIDIOut`, which can be instantiated with a device number and a channel number. There is also a class `MIDIDeviceHelper` with several helper functions (for example for enumerating all available MIDI devices).

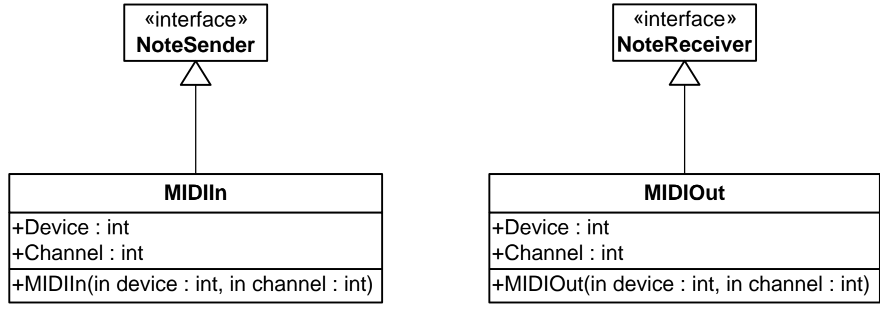


FIGURE 35 MIDI INPUT/OUTPUT

As an example, we show how to instantiate MIDI devices and pass incoming notes to an output:

```
// instantiate a MIDI input (device 0, channel 0)
MIDIIn midiIn = new MIDIIn(0, 0);

// MIDI output
MIDIOut midiOut = new MIDIOut(0, 0);

// connect input to output
midiIn.SendNote += midiOut.ReceiveNote;
```

Similar to sending notes, we also use an event-based system for sending clock information. Clock events provide global metric information making it possibe to for example determine if an event corresponds to a 16th note or if it is an 'off-beat'. A class `Length` represents the length of a note; common note lengths are again provided as static variables (e.g. `Length.QUARTER)`. The smallest possible length corresponds to the time interval between clock events and is called a 'tick'; we currently use a resolution of 96 ticks per bar in all of our applications.

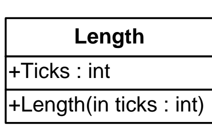


FIGURE 36 NOTE LENGTHS AND CLOCK EVENTS

Clocks are defined with an interface `Clock` and provide an event for sending ticks (the common periodic clock events) as well as events which occur when the clock is being started or stopped. The interface `ClockReceiver` defines the corresponding functions for receiving and processing these events:

```
public interface Clock
{

    event EventHandler<ClockEventArgs> SendTick;

    event EventHandler SendStart;

    event EventHandler SendStop;

}

public interface ClockReveiver
{

    void ReceiveTick(object sender, ClockEventArgs e);

    void ReceiveStart(object sender, EventArgs e);

    void ReceiveStop(object sender, EventArgs e);
}
```

There are two implementations of `Clock`: An `InternalClock` uses an internal timer for which the tempo can be defined in 'quarter notes per minute' (beats per minute, BPM). An `ExternalClock` allows synchronization to external MIDI equipment; this class can be instantiated with a MIDI device number which should be used to receive MIDI synchronization messages. It is also possible to synchronize external equipment to an internal clock; this is realized with a class `ClockOut` implementing `ClockReceiver`. Similar to `ExternalClock`, a MIDI device number defines the target device for synchronization.

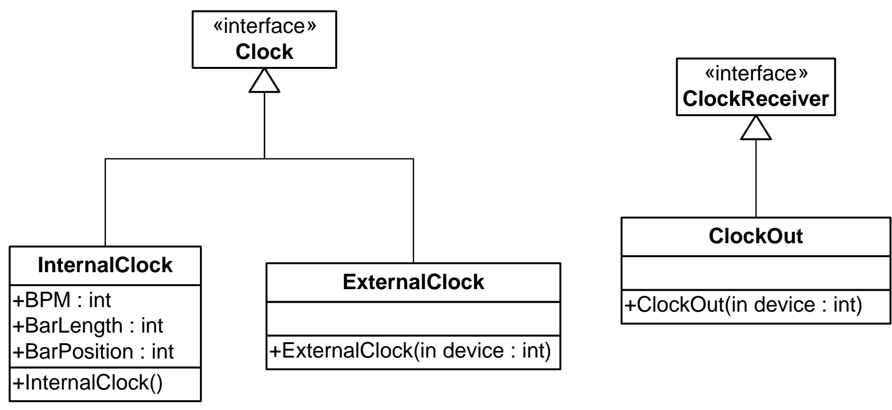


FIGURE 37 CLOCKS

6.2.2 SOFT CONSTRAINTS

This component provides data structures for modeling problems with monoidal soft constraints as well as a solver with several problem-specific performance optimizations. It is based on a formal model with all optimizations proven correct. In this work, we will introduce this component from a user's point of view. We refer to (13) and (28) for a more detailed insight on how it works internally.

Variables and values are defined by implementing the corresponding interfaces and defining equality. Furthermore, each variable has to be associated with a unique number. The classes `NumberedVariable` and `NumberedValue` provide a simple way to obtain variables/values; both have an empty constructor which successively creates distinct instances.

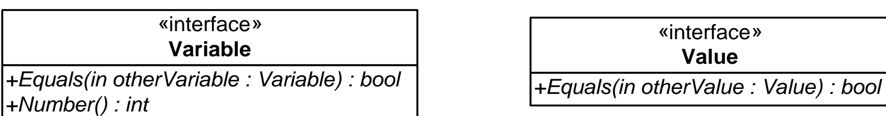


FIGURE 38 VARIABLES AND VALUES

Variables can be grouped to a domain; values to a valuation:

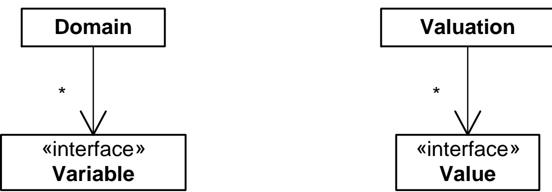


FIGURE 39 DOMAINS AND VALUATIONS

The ordered monoids for rating valuations are defined by implementing the interface `Grade`. Functions have to be provided for comparing and combining grades and checking for equality. Many common monoids are already integrated, for example Boolean values or numbers (integer/float) along with several combination operations (addition/multiplication/minimum…).

«interface»
Grade
+IsGreaterThan(in otherGrade : Grade) : bool
+Equals(in otherGrade : Grade) : bool
+CombineWith(in otherGrade : Grade) : Grade

FIGURE 40 MONOIDS

There are two types of constraints: *Functional soft constraints* are defined by subclassing `FunctionalSoftConstraint` and overriding the abstract function `ComputeGrade`. The constraint's domain is defined in the constructor and the order of values in a valuation passed to `ComputeGrade` is always consistent with the domain's order.

```
public abstract class FunctionalSoftConstraint : SoftConstraint
{
    public abstract Grade ComputeGrade(Valuation valuation);

    public FunctionalSoftConstraint(Domain domain)
        : base(domain)
    {…}

}
```

Explicit soft constraints are defined with a discrete map which assigns grades to valuations. For each variable, there has to be at least one explicit constraint enumerating its domain. A factory class makes it easy to define explicit constraints:

```
public class ExplicitSoftConstraintFactory
{
    public void AddVariable(Variable variable) {…}

    public void AddMapEntry(Value[] values, Grade grade) {…}

    public ExplicitSoftConstraint GenerateConstraint() {…}
    …
}
```

First, all variables must be added with the function `AddVariable`. Then, the map entries can be added with the according function. The number and order of values must match the number and order of variables in the constraint's domain. When all map entries have been added, the constraint can be generated.

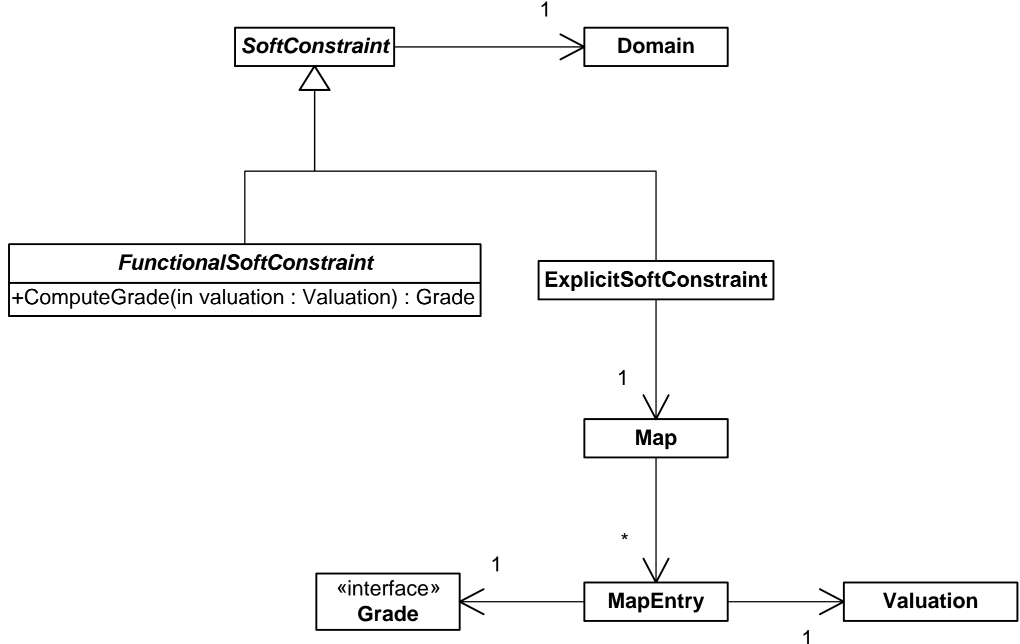


FIGURE 41 SOFT CONSTRAINTS

When no preferences over the constraints itself are needed, a list of soft constraints using the same monoid can be directly solved like this:

```
List<SoftConstraint> constraints = …;

SinglePro blem problem = new SingleProblem(constraints);

List<Solution> solutions = problem.Solve();
```

Single problems can also be combined to more complex problems as described in (12). There are two very basic types of combinations: An `Indifference` combines problems using an order by component; a `Preference` combines problems using a lexicographic order:

```
SingleProblem problem1 = …;

SingleProblem problem2 = …;

SingleProblem problem3 = …;

SoftConstraintCombination c1 = new Indifference(problem1, problem2);

SoftConstraintCombination problem = new Preference(c1, problem2);

List<Solution> solutions = problem.Solve();
```

It is also possible to define other preferences over constraints by defining embedding functions. Embedding functions are instances of this delegate:

```
public delegate Rank Embed(Grade grade, Rank rank);
```

The problem can then be defined with an instance of `SoftConstraintProblem`. Each soft constraint has to be added to this problem together with its embedding function using the function `AddSoftConstraint`. These problems can be solved with an instance of the class `Solver`.

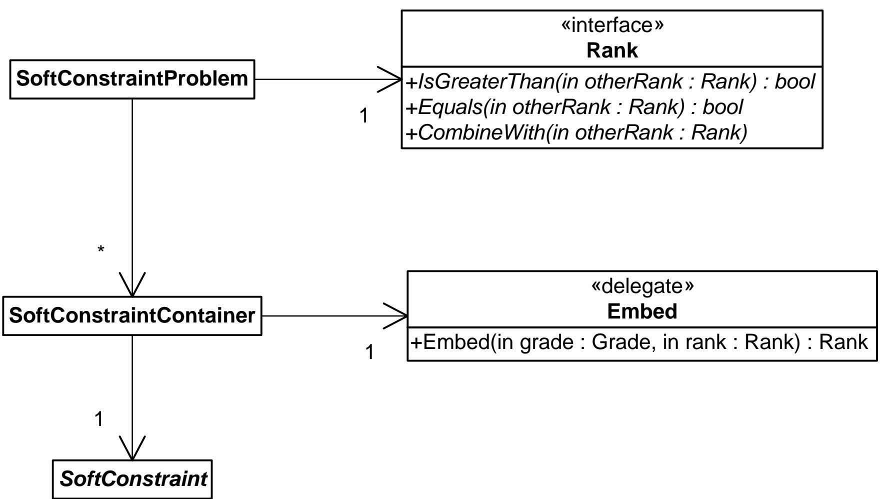


FIGURE 42 SOFT CONSTRAINT PROBLEMS

6.2.3 Musical Soft Constraints

This component makes it possible to compose music with soft constraints; it depends on the components described above: Music and SoftConstraints. The problem of generating music for several musical interfaces is modeled by assigning *Actions* to *Voices*. Each voice can play one note at a time - if an interface should be polyphonic (i.e. it can play multiple notes at a time) it has to have an according number of voices. As described in the theory, we currently use three types of actions:

- Start a note with a given pitch (`NoteOn`)
- Hold a note (`Hold`)
- Do nothing (`Pause`)

We use soft constraints for assigning actions (values) to voices (variables). A `Voice` implements `Variable` and always stores the last action it performed. The abstract class `Action` inherits from the interface `Value`; `Pause` directly inherits from `Action` and `NoteOn` and `Hold` inherit from another abstract class `NoteAction` which holds a note pitch.

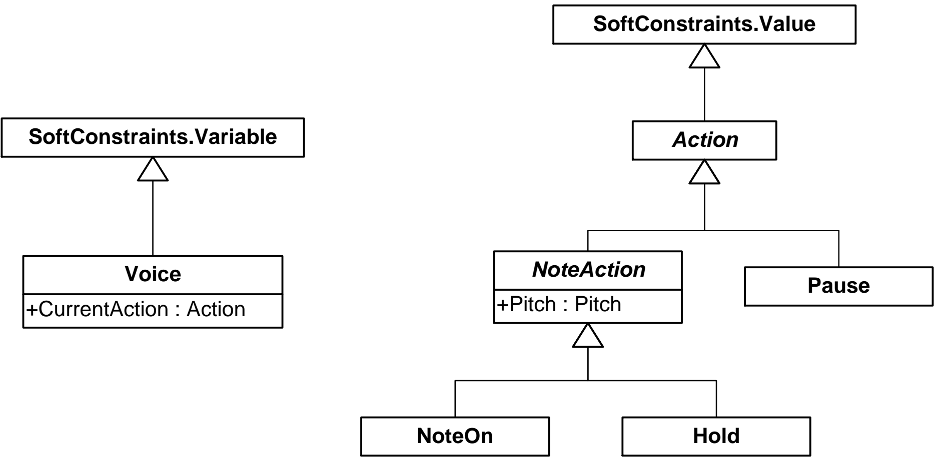


FIGURE 43 VARIABLES AND VALUES

The component uses the following operation cycle: first, each musical interface states its current preferences with one or several soft constraints (typically based on sensor data). Then, these soft constraints are combined among each other and extended with global preferences coordinating the interplay between the interfaces. The complete problem is solved and the resulting solution defines an action for each voice. These actions are passed to the corresponding musical interfaces which execute it. After a certain time, the cycle is repeated.

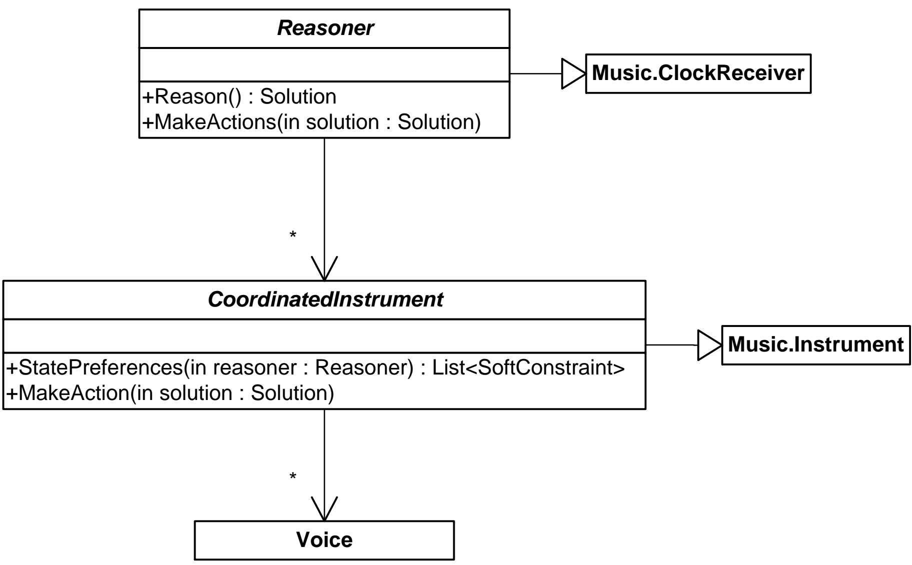


FIGURE 44 INSTRUMENT COORDINATION

The class `Reasoner` is the component's central controller. The operation cycle is triggered when the method `Reason` is called: this can be done at fixed time intervals by a clock or manually, for example initiated by user interaction. Musical interfaces are instances of `CoordinatedInstrument`, which is a subclass of `Instrument`. Each coordinated instrument holds one or several voices. The method `StatePreferences` is called when the instrument should state its preferences with soft constraints. Both `Reasoner` and `CoordinatedInstrument` are abstract classes: when implementing a concrete application, the abstract methods `Reason` and `StatePreferences` have to be implemented. A typical implementation of `Reason` first collects all instruments' preferences by calling `StatePreferences`. Then, global constraints can be added. The final problem is then passed to the constraint solver which returns one or several optimal solutions and one of these has to be chosen (since all solutions are optimal, one can be randomized). This solution just has to be returned and the framework will do the rest by calling `MakeAction` in `Reasoner` which passes it to each instrument by calling `MakeAction` in `CoordinatedInstrument`. This function finally executes the actions by sending note events.

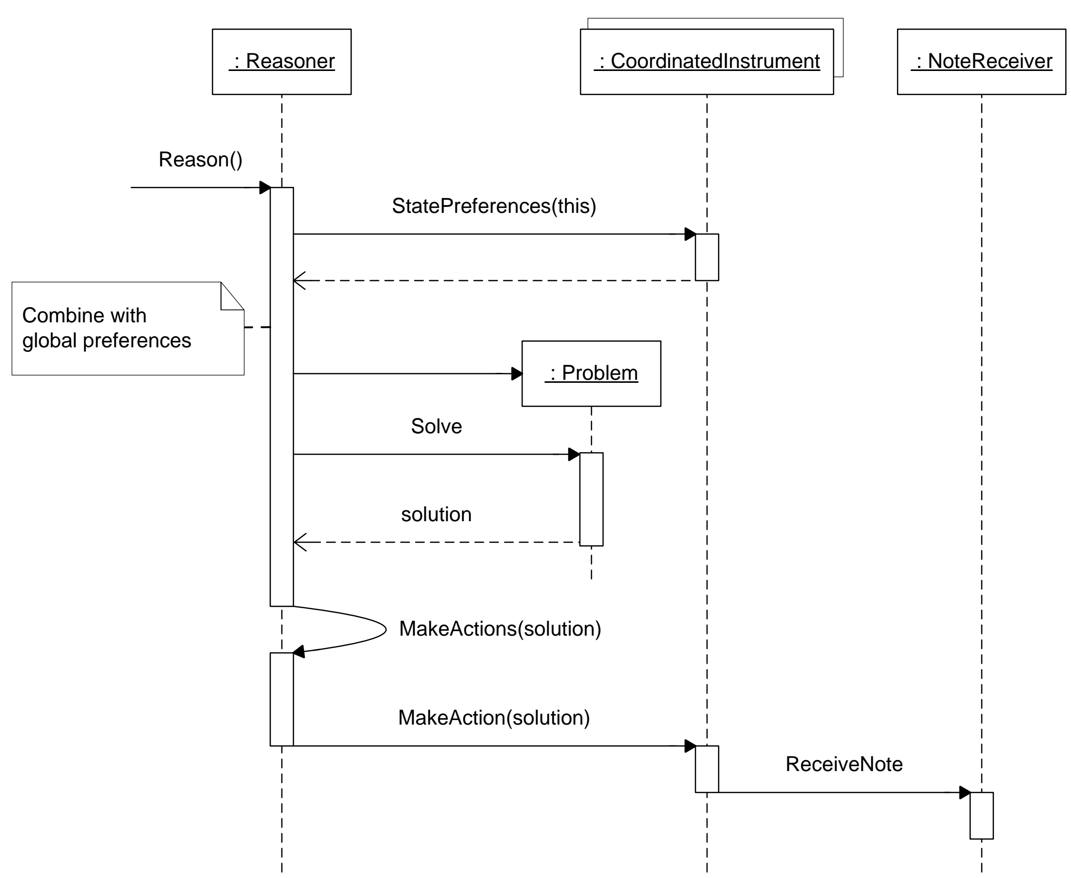


FIGURE 45 OPERATION CYCLE

There is a factory class `MusicConstraintFactory` which makes it easy to define explicit musical soft constraints. First, all voices have to be added by calling `AddVoice`. Then, the actions can be added with a grade. For constraints over multiple voices, the method `AddActions` has to be called with a list of actions and a grade. Constraints over a single voice can also be generated by calling functions which directly add an action with a certain grade (e.g. `AddPause`). When all actions for a constraint have been added, the function `GenerateConstraint` has to be called. This generates the constraint and adds it to the list of constraints `SoftConstraints`. Then, a new constraint over the same voices can be generated. There are also several subclasses of the factory which directly generate constraints over a certain monoid (for example `MusicConstraintFloadAddFactory`).

```
public class MusicConstraintFactory
{

    public List<SoftConstraint> SoftConstraints;

    public void AddVoice(Voice voice);

    public void AddActions(List<Action> actions, Grade grade);

    public void AddAction(Action action, Grade grade);

    public void AddNoteOn(Pitch pitch, Grade grade);
    public void AddHold(Pitch pitch, Grade grade);
    public void AddPause(Grade grade);

    public void GenerateConstraint();
    public void Reset();

}
```

We will now sketch out an example for implementing a very simple music application with several constraints. First, we implement a subclass of `CoordinatedInstrument` and override `StatePreferences` such that it returns five random note-on actions and a pause action with a random grade from the monoid of real numbers with addition:

```
public override List<SoftConstraint> StatePreferences(Reasoner r)
{
    MusicConstraintFactory factory = new MusicConstraintFactory();

    factory.AddVoice(Voices[0]);

    for (int n = 0; n < 5; n++)
    {
        factory.AddNoteOn(
            new Pitch(random.Next(40, 80)),
            new FloatAdd((float)random.NextDouble())
            );
    }

    factory.AddPause(new FloatAdd((float)random.NextDouble()));

    factory.GenerateConstraint();

    return factory.SoftConstraints;
}
```

We implement a simple reasoner and override `Reason`. This method collects all instrument constraints and generates an additional harmony constraint with a random weight (we will take a closer look at this afterwards). All constraints are combined, solved and the first solution is returned:

```
public override Solution Reason()
{
    List<SoftConstraint> constraints = new List<SoftConstraint>();

    foreach (CoordinatedInstrument i in Instruments)
    {
        constraints.AddRange(i.StatePreferences(this));
    }

    HarmonyConstraint harmony = new HarmonyConstraint(
            Instruments,
            (float)random.NextDouble()
            );

    constraints.Add(harmony);

    SingleProblem problem = new SingleProblem(constraints);

    return problem.Solve()[0];
}
```

This is pretty much it – the framework will do the rest. We just need to connect some instruments and a clock to the reasoner and the application is complete. In this example, we

make use of a predefined functional soft constraint, the `HarmonyConstraint`. We will now take a look at the implementation of this constraint. For realizing the harmony constraint, we need a function which rates the harmony between two actions. In general, we define an interface `HarmonyRating` which rates the harmony between two actions with an integer number. The default harmony rating is based on the theory of harmony from Howard Hanson (44) as described in section 5.2.

```
public interface HarmonyRating
{

    int Harmony(Action action1, Action action2);

}
```

In contrast to explicit soft constraints, functional soft constraints are defined with an arbitrary function from valuations to grades. The `HarmonyConstraint` sums up the harmony between all pairs of actions in a valuation and multiplies this sum with a constant factor `weight` which is defined in the constructor. The task of optimizing this value is done by the constraint solver. Besides a few more technical lines of code (class declaration, constructor etc.) the harmony constraint can be defined as short as this:

```
public override Grade ComputeGrade(Valuation valuation)
{
    int totalHarmony = 0;

    for (int v = 0; v < valuation.Values.Length; v++)
    {
        for (int w = (v + 1); w < valuation.Values.Length; w++)
        {
            totalHarmony += harmonyRating.Harmony(
                valuation.Values[v] as Action,
                valuation.Values[w] as Action
                );
        }
    }

    return new FloatAdd(totalHarmony * weight);
}
```

6.2.4 Trainable Musical Models

Musical models allow training melodic preferences in order to make the generated music consistent with a certain musical style. These models are used for automatically computing soft constraints expressing how well the melodies generated by an instrument comply with the training data. We want to be able to use musical models in real-time: it should be possible to both play and train a model in a real-time application. As defined in section 5.3, a musical model has several *States* corresponding to the actions an instrument can perform and several *Steps* representing rhythmic divisions. At each step, each state has a certain weight. Furthermore, for each step and state, there are transition weights to the states on the next step. Both kinds of weights are modeled with floating-point numbers. Playing and training musical models requires frequent access to these weights and hence, the implementation of musical models should allow reading and modifying weights in an efficient way. We decided to implement an abstract musical model which does not depend on a certain type of states. Based on this general model, we developed two concrete models specialized for rhythmic respectively tonal sequences.

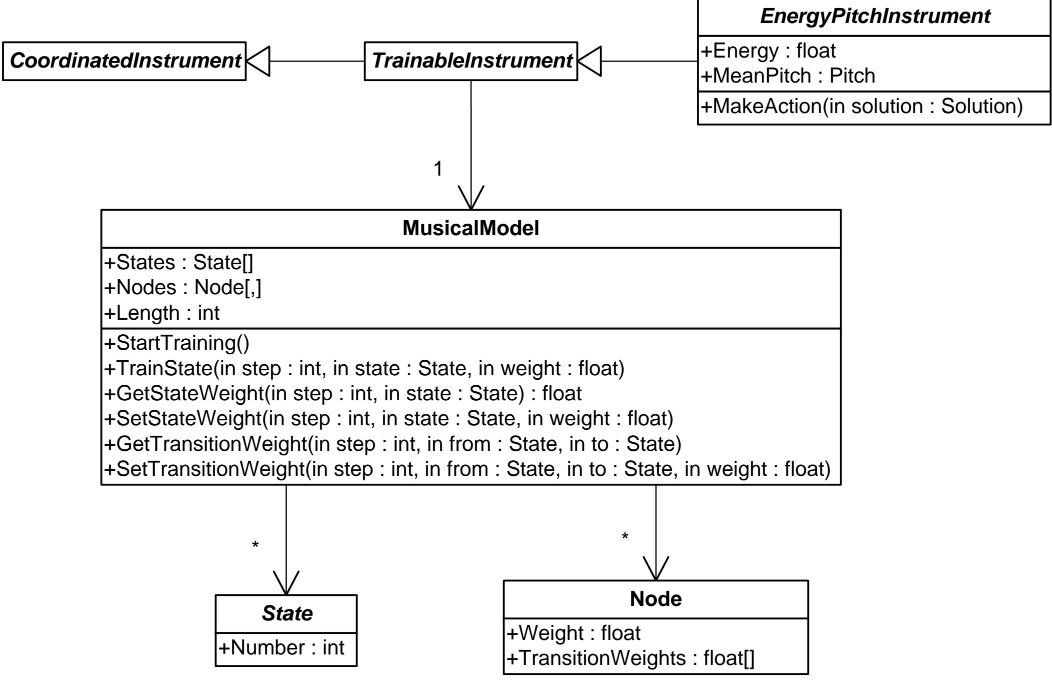


FIGURE 46 MUSICAL MODEL

A musical model holds an array of states: `State` is an interface which requires a unique integer number that has to correspond with the state´s index in the array. A class `Node` is used for storing the states´ weights. A two-dimensional array of nodes holds one node for each step and state: a node´s first index corresponds to the step, the second one to the state´s unique number.

A node stores a single state weight as well as transition weights to the states on the subsequent step. These transition weights are stored in an array of floating-point numbers such that a state´s transition weight can be accessed by the state´s unique number. This way, it is possible to get and set all weights in constant time:

```
public float GetStateWeight(int step, State state)
{
    return nodes[step, state.Number].Weight;
}

public float GetTransitionWeight(int step, State from, State to)
{
    return nodes[step, from.Number].TransitionWeights[to.Number];
}
```

`TrainableInstrument` is an abstract subclass of `CoordinatedInstrument` which already holds a reference to a musical model and can serve as a base class for own implementations of generating soft constraints from a musical model. `EnergyPitchInstrument` is an abstract subclass extending `TrainableInstrument` and allows playing a musical model by deriving values for 'energy' and 'pitch' from user interaction. Based on this class, there are two classes for playing rhythmic respectively tonal sequences with energy and pitch; each depends on a corresponding specialized musical model:

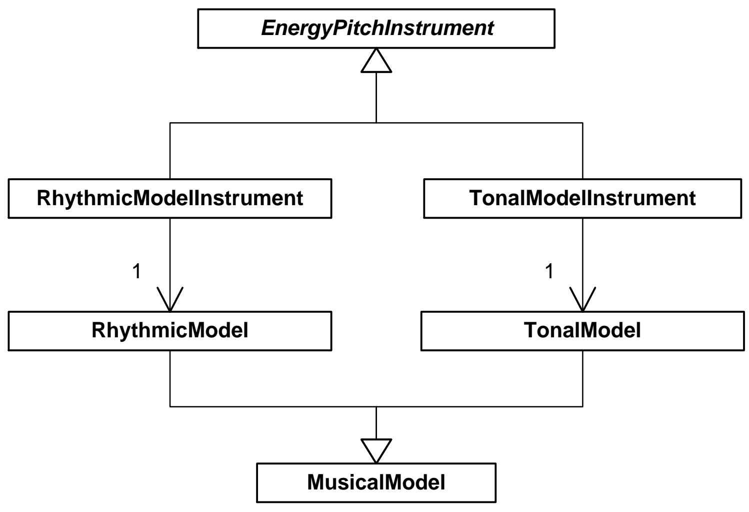


FIGURE 47 TONAL AND RHYTHMIC MODELS

Each model uses a distinct set of states: A rhythmic model contains only two states representing notes (`HitState`) and silence (`PauseRhythmic`). Similarly, a tonal model contains a state for silence (`PauseTonal`) as well as states representing notes which are based on an abstract class `StageState`. This state holds a reference to a stage in a tonal scale and has two subclasses: `NoteOnState` represents a starting note and `HoldState` represents a note which was started before and is still being held.

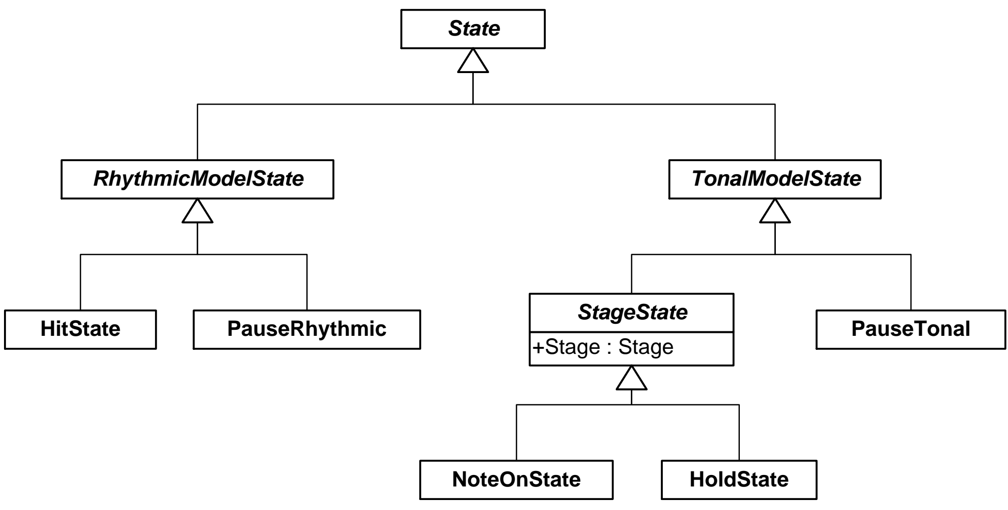


FIGURE 48 TONAL AND RHYTHMIC STATES

As mentioned above, each state has to have a unique number which makes it possible to get and set weights in constant time. Assigning this number to rhythmic states is easy since there are only two states; it becomes a bit more complicated for tonal states: we want to be able to use models with a variable number of stages which can also be augmented. Furthermore, there are two subclasses of `StageState` which need to have a unique number for each stage. Each stage has a unique number which is computed based on its position in the tonal scale (`Number`) and whether it is regular or augmented (`IsRegular`):

```
public int GetID()
{
    if (IsRegular) // regular stage
    {
        return Number * 2;
    }
    else // stage is augmented
    {
        return (Number * 2) + 1;
    }
}
```

Based on a stage´s number, we assign unique numbers to all tonal states: `PauseTonal` has a static number of 0; the two subclasses of `StageState` get alternating numbers: a `NoteOnState` with a given `Stage` has a number of `(Stage.GetID() * 2) + 1`; the corresponding `HoldState` gets a number which is one higher: `(Stage.GetID() * 2) + 2`. This way, the number of a state can be computed in constant time without any additional data.

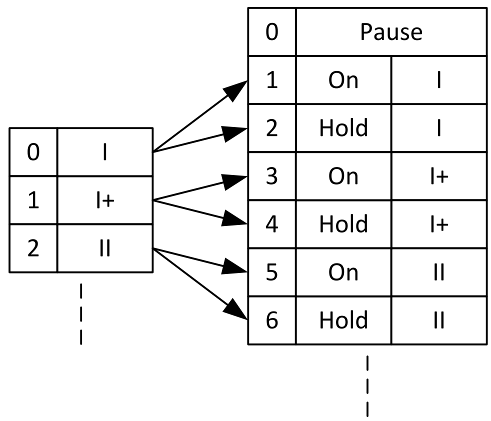


FIGURE 49 UNIQUE STATE NUMBERS

We will now take a closer look at how models can be trained with note pitches and then show how soft constraints can be constructed from a model. Both kinds of weights can be trained with a single function `TrainState`. Calling this function modifies the state weight as well as the transition weight from the last step to the current step using a member variable `lastState` which stores the last trained state:

```
public void TrainState(int step, State state, float weight)
{
    // modify state weight
    nodes[step, state.Number].Weight += weight;

    // compute the last step number
    int lastStep = step - 1;
    if (lastStep < 0) lastStep = Length - 1;

    // modify transition weight
    nodes[lastStep, lastState.Number].
    TransitionWeights[state.Number] += weight;

    // remember last training state
    lastState = state;
}
```

Given a step s, a tonal scale scale and a weight w, a tonal model m can be trained with a certain pitch p like this:

```
m.TrainState(s, new NoteOnState(scale.GetStage(p)), w);

m.TrainState(s, new HoldState(scale.GetStage(p)), w);
```

When a NoteOn message is received, a NoteOnState has to be trained on the current step. Until the NoteOff message arrives, a HoldState has to be trained at every passing step. When no notes are received, pauses have to be trained like this:

```
m.TrainState(s, new PauseState(), w);
```

Rhythmic models are trained the same way:

```
m.TrainState(s, new HitState(), w);

m.TrainState(s, new PauseRhythmic(), w);
```

Playing a musical model can be done in several ways. The framework directly implements the approach based on 'energy' and 'pitch', but it is of course also possible to implement custom approaches. Independently from a concrete approach, we will now present several code examples showing the most basic steps for generating soft constraints from a musical model. In these examples, we make use of the following variables: We keep track of the current rhythmic position with two integer numbers, currentStep and lastStep. The last actually executed state lastState is required for computing transition weights. When generating soft constraints from a tonal model, a StageState has to be reconstructed from the the last action´s pitch (Voices[0].Pitch) and the current tonal scale (reasoner.CurrentScale). The actual task of generating constraints is done with a factory for constraints over a monoid based on floating-point numbers (factory). Generating a soft constraint from a musical model is based on three types of actions respectively states for pausing, starting and holding. The grade of an action is derived from the corresponding state´s weights. Depending on the concrete implementation, a StageState can also have multiple actions corresponding to it, because it can be played in different octaves and tonal scales.

The Pause action directly corresponds to a state (`pauseState`); its grade can be computed by summing up the pause´s current state weight and its transition weight:

```
float weight = 0;

// add the state weight
weight += TonalModel.GetStateWeight(currentStep, pauseState);

// add the transition weight
weight += TonalModel.GetTransitionWeight(lastStep, lastState,
    pauseState);

// add pause action to the constraint factory
factory.AddPause(weight);
```

The grade of a rhythmic action (`HitState`) can be computed the same way. When it comes to tonal models, pitch classes and stages in a tonal scale can be converted among each other based on the current tonal scale. A pitch class can correspond to a single note pitch in a certain octave as well as to multiple pitches in different octaves. `NoteOn` actions can then (for example) be added to a constraint like this:

```
foreach (NoteOnState state in TonalModel.NoteOnStates)
{
    float weight = 0;

    // add the state weight
    weight += TonalModel.GetStateWeight(currentStep, state);

    // add the transition weight
    weight += TonalModel.GetTransitionWeight(lastStep, lastState,
        state);

    // get the stage´s pitch class w.r.t. the current tonal scale
    PitchClass pitchClass = reasoner.CurrentScale.GetPitchClass(
        state.Stage);

    // convert the pitch class to a concrete note pitch
    Pitch pitch = DoSomethingWith(pitchClass)

    // add note on action to the constraint factory
    factory.AddNoteOn(pitch, weight);
}
```

When an instrument is currently playing a note, it is possible to hold this note in the next step. The stage corresponding to this note has to be reconstructed w.r.t. the current tonal scale. Then, a weight can be computed for a `HoldState` based on this stage:

```
if (Voices[0].IsPlaying)
{
    // reconstruct playing stage w.r.t. the current tonal scale
    Pitch pitch = Voices[0].Pitch;
    Stage stage = reasoner.CurrentScale.GetStage(pitch);

    // instantiate a hold state with this stage
    HoldState holdState = new HoldState(stage);

    float weight = 0;

    // add the state weight
    weight += TonalModel.GetStateWeight(currentStep, holdState);

    // add the transition weight
    weight += TonalModel.GetTransitionWeight(lastStep, lastState,
        holdState);

    // add hold action to the constraint factory
    factory.AddHold(pitch, weight);
}
```

The framework includes implementations of `EnergyPitchInstrument` which allow playing tonal and rhythmic models with only two parameters. For efficiency reasons, these implementations do not generate separate constraints for the model and the user interaction. Instead, both constraints are already merged into a single constraint which reduces the constraint problem´s complexity.

6.3 EVALUATION

Based on the non-functional requirements, we will now asses the quality of our framework and then present the results of performance tests.

6.3.1 QUALITY ANALYSIS

The first prototypes for implementing musical soft constraints have been developed in 2008; the first version of the current framework was developed in early 2009. Until now (2014), we used the framework in a variety of applications and continually improved it. The framework is built based on a formal model which depends on well-known theories for music and soft constraints and the constraint solver is based on a verified prototype in Maude (28). In the current implementation, all search optimizations are verified. These are all factors which contribute to the system´s quality and stability. Our personal impression is in accordance with this: the framework works as intended and has a high *reliability*.

The *efficiency* of the framework suffices in all of our applications. When the number of voices does not exceed about four (which is reasonable for most applications), even devices with low computing power suffice to run the framework. When a higher number of voices are required, the performance can be greatly improved by using problems where search optimizations are applicable. The memory usage of the core framework is rather low (typically less than a megabyte) which is unlikely to be a limiting factor on devices running the .NET framework. In addition to the core framework, musical models also require a constant amount of memory depending on the model´s complexity and length. In most cases, memory usage will not be the limiting factor. The timing of events is good in most cases. Garbage collection can often be a problem in real-time applications but the garbage collector of the .NET framework is implemented very efficiently and did not audible affect the timing. When the CPU has usage peaks from other processes, the timing can get irregular for a short amount of time. Detailed results of performance tests can be found in the next section.

The *usability* of the framework is hard to assess, because so far no person worked with it which was not involved in its development and thus has knowledge of the internal structure. Nevertheless, we think that there is a clear structure and a straightforward and documented process for developing applications. The applications based on the framework did only require few programming code related to generating music: defining musical soft constraints can be done very fast but requires knowledge of the underlying theory. When there is no knowledge about soft constraints or music theory, the initial learning curve will have a rather low slope in most cases. Nevertheless, applications based on musical models and predefined constraints can be developed without having to deal with soft constraints. Musical models can be trained

without having any programming skills at all; developing interactive applications based on musical models can be done by only deriving two parameters from user interaction.

In order to achieve *maintainability*, the framework is implemented in a very modular way with high cohesion within a single component and low coupling among multiple components. The framework has been developed in an agile process with incremental integration of functionality and much effort was spent for keeping a clear structure by constant refactoring. All relevant parts of the framework are documented textually and with class diagrams. All programming code is written with respect to a consistent coding style. The code is documented well; complex functionality is described in natural language with one comment for every line of code.

The framework´s *portability* is not satisfying at the moment. Since it is developed with Microsoft´s .NET framework, it will only run on Windows platforms (Desktop, Surface, embedded, mobile, Xbox360 etc.). There are several approaches for running .NET applications on other platforms (e.g. the Mono project), but – from our personal experiences - they do not have a good quality when it comes to efficiency and ease of development. In order to achieve a good portability, the framework has to be converted to a commonly accepted programming language like C++. This can partially be done with automatic code converters but also requires a large amount of manual work.

6.3.2 Performance Tests

All tests were performed on a laptop with a 1.67 GHz CPU and 2 GB RAM.

We consider problems where no search optimizations are applicable, i.e. the whole search space has to be examined in order to get an optimal solution. In any case, such problems will be at least as complex to solve as any other problem with the same dimensions. We generated random problems with a variable number of voices and actions per voice and measured the time for finding the best solution. As long as this time is below the time between two steps, it is possible to achieve a constant latency (which is hardly audible thanks to its constancy). Otherwise, the problem is too complex and has to be reduced or optimized. In our applications, we use a resolution of 16th notes which leads to upper bounds for search times from about 250ms (60 bpm) to 83ms (180 bpm).

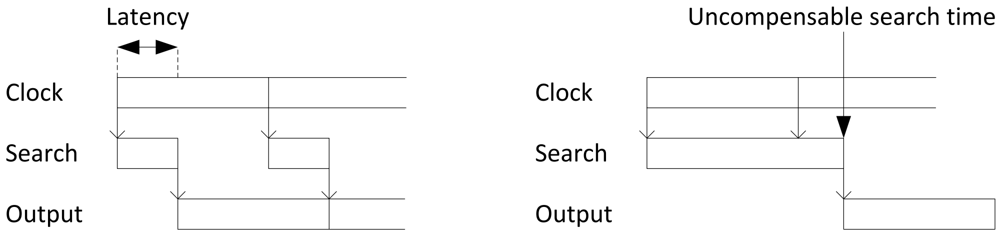


FIGURE 50 LEFT: APPLICABLE SEARCH TIME, RIGHT: SEARCH TIME IS TOO HIGH

The following diagram visualizes the computation times of random problems with 3 – 6 voices, each having 1 – 10 actions. Furthermore, a `HarmonyConstraint` is defined over all voices. Since no optimizations are used, all problems having the same dimensions take about the same time to solve. The horizontal line marks the time between two 16th notes at 120 bpm (125ms):

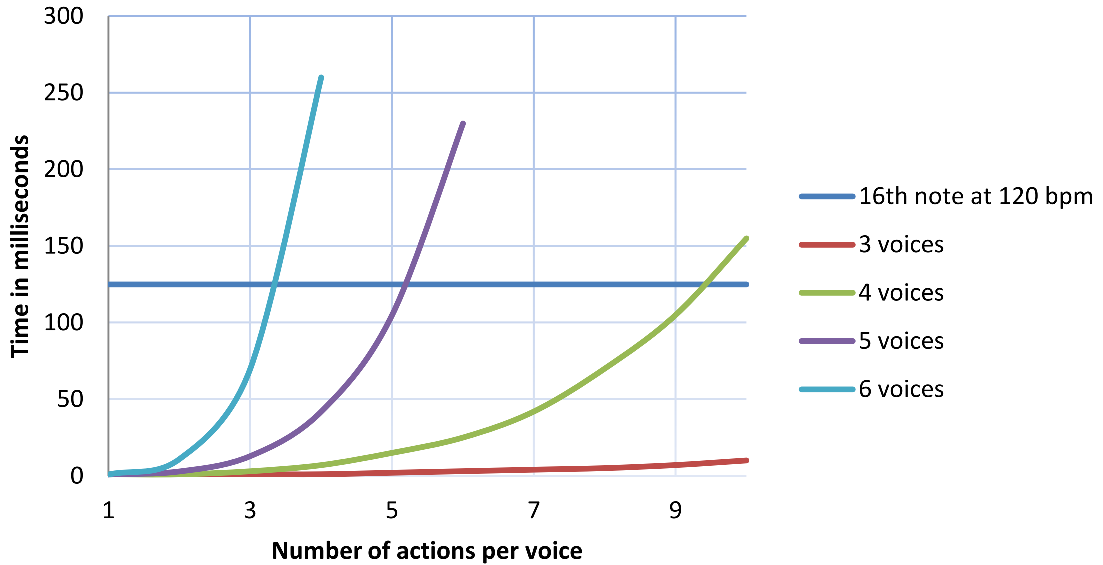


FIGURE 51 TIME TO SOLVE A MUSICAL SOFT CONSTRAINT PROBLEM

When experimenting with the framework, we had the personal impression that the generated music begins to sound unorganized when the number of voices exceeds about four. We also made the observation that it is sufficient in all of our applications to consider only the about five best actions for each voice. This type of problems (four voices, five actions per voice and a harmony constraint) takes about 15ms to compute (without any optimization), which is clearly below the time between two steps at common tempos. The cost of a functional constraint can vary strongly from case to case. The harmony constraint is rather expensive to compute because

it needs to consider all pairs of variables. The next figure compares the computation times of problems over four voices with and without the harmony constraint:

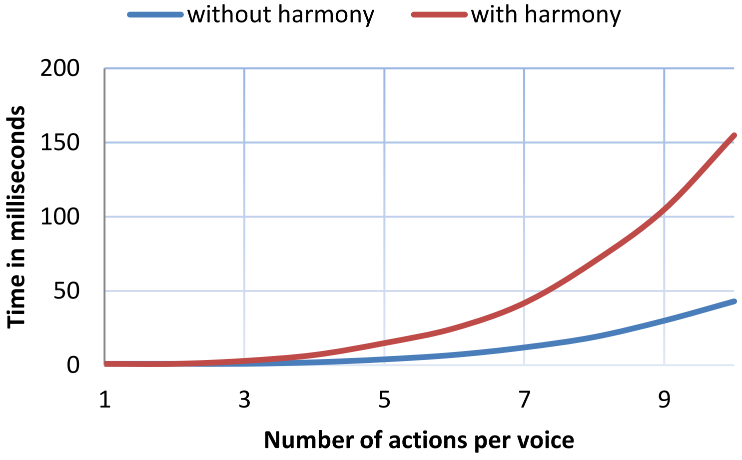


FIGURE 52 COST OF THE HARMONY CONSTRAINT (FOUR VOICES)

For some types of problems, search optimizations can be employed which lead to drastic reductions of search times under certain conditions. The solver used in this framework supports several search optimizations (see (13)). The following diagram shows the distribution of computation times for general soft constraint problems with 10 constraints. Without optimization, these problems take several minutes to compute. When using an optimized branch and bound algorithm which is applicable to a certain type of problem, efficiency is greatly improved: The average computation time is 5.2s with most problems taking clearly below one second. There was one heavy outlier taking nearly one and a half minute and several outliers in the other direction which took nearly no computation time at all.

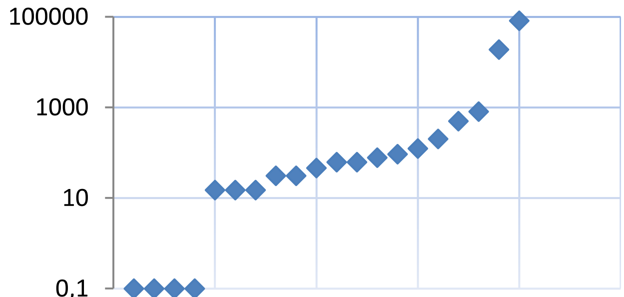


FIGURE 53 SOME OPTIMIZED COMPUTATION TIMES IN MILLISECONDS (LOGARITHMIC SCALE)

The memory usage of the framework is unlikely to be a limiting factor since it is rather low compared to the processor usage. The following figure shows the memory usage of an application based on problems with 5 voices and 6 actions per voice. While the program is running, data is generated; for example, the constraint solver stores backtracking continuations and the communication of notes and metric information is done with events. At certain time intervals, the garbage collector releases obsolete data:

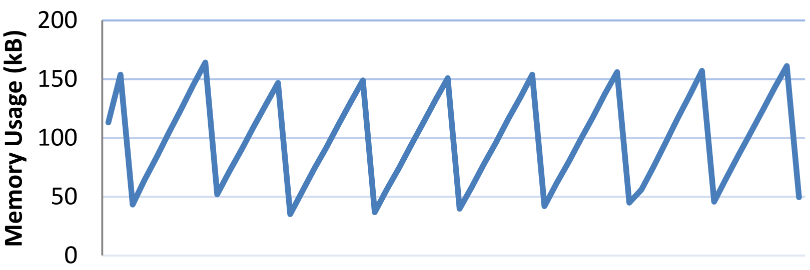


FIGURE 54 MEMORY USAGE IN KILOBYTE OVER 10 SECONDS

The memory usage of the core framework can be kept rather small since only few data needs to be stored for a longer time. Even for larger problems where the CPU comes to its limits, the maximum memory usage is less than 200 Kilobyte. Besides the data used by the basic infrastructure, musical models also have to be kept in the memory because fast access to them is required. At least all currently active musical models have to be available; new models can also be loaded from permanent storage in the background while the system is playing. The size of a musical model depends on its type and number of steps: a tonal model with seven stages in a tonal scale (15 states) needs about 0.54 kB memory per step. Using a resolution of 16th notes, a model with one bar requires for example about 9 kB; a model with 16 bars about 138 kB. Rhythmic models have only two states and require only 0.01 kB per step.

7 Applications

This section presents the applications built upon our framework so far: ‘The Planets’ is an interactive music system controlled by a table-based tangible interface where music can be interactively composed by arranging planet constellations. Then, we present a pattern-based step sequencer called ‘Fluxus’ which allows training and interactively playing musical models. In the subsequent section, we present a general approach for transforming spatial movements into music and two concrete applications of it: the first one is based on two-dimensional movements on a touch screen; the second one uses markerless motion-tracking to generate music from three-dimensional body movements. At last, we will present research on how interactive music systems can be employed in the area of pervasive advertising.

7.1 ‘The Planets’ for Microsoft Surface

‘The Planets’ (92) was developed in cooperation with Max Schranner from the Academy of Fine Arts in Munich and combines our approach for algorithmic composition with new human-computer interaction paradigms and realistic painting techniques. In this work, we contributed to the concept, implemented the application and participated in creating the tangibles. It was exhibited at the ‘night of science (Senses09)’ in Munich and was a finalist in the ‘Ferchau Art of Engineering (2010)’ contest. The main inspiration for it was the composition ‘The Planets’ from Gustav Holst who portrayed each planet in our solar system with music. Our application allows to interactively compose music in real-time by arranging planet constellations on an interactive table. The music generation is controlled by painted miniatures of the planets and the sun which are detected by the table and supplemented with an additional graphical visualization, creating a unique audio-visual experience.

FIGURE 55 THE PLANETS AT SENSES09 IN MUNICH

7.1.1 Concept

The Planet's user interface is entirely based on miniatures of the sun and the planets in our solar system which can be arranged on an interactive table and generate music depending on their current constellation. A planet's absolute position on the table does not play a role – only its relative position towards the sun is of importance. There are five smaller planets (Mercury, Venus, Mars, Neptune and Uranus) each representing an instrument with a different sound. Moving an instrument planet towards the sun makes it play faster (more and shorter notes); a planet which is far away from the sun plays only few notes.

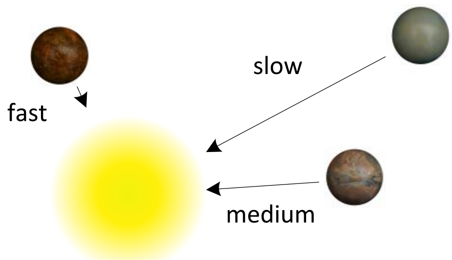


FIGURE 56 SPEED

The instrument's pitch is controlled by the planet's relative angle to the sun: rotating it clockwise around the sun increases its pitch, rotating it counter-clockwise decreases it such that a full rotation corresponds to one octave. The pitch is not controlled deterministically; there are always a larger number of possible pitches corresponding to a certain angle. Whenever a planet plays a note, a supplementing visualization is displayed on the table: at the position where the note was started, a sphere appears which becomes bigger and fades out over time. The visualization does not follow the planet but rather stays where the planet was when the note was started. This way, a planet being moved on the table leaves a trace of spheres behind it representing the notes it recently played.

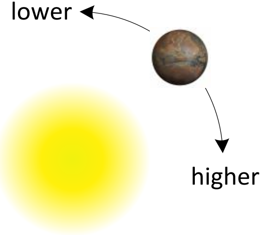


FIGURE 57 PITCH

Jupiter and Saturn do not play notes – instead, they control global parameters affecting the interplay between the instrument planets; they are also bigger than the other planets in order to make their special role clear. Jupiter controls the global 'harmony' between all instrument planets: the closer it is to the sun, the more harmonic intervals between the instrument planets are played (like fifths, fourths or thirds). This is also being reflected in its visualization on the table: when there is high harmony (Jupiter is near the sun), its visualization is green – moving it away fades its color to brown ('medium harmony') and, finally, to red ('no harmony at all'). Rotating Jupiter around the sun changes the global tonal scale. This is done in steps of fifths in order to create natural harmonic modulations around the 'circle of fifths'. For example, when the current tonal scale is c-major (also corresponding to a-minor since there is no fixed tonic pitch class), a clockwise rotation modulates to g-major (and then to d-major, a-major and so on). Playing just notes from a given tonal scale is only enabled when Jupiter is within a certain range around the sun – if it is out of this range, there is no restriction to a scale anymore and any note can be played.

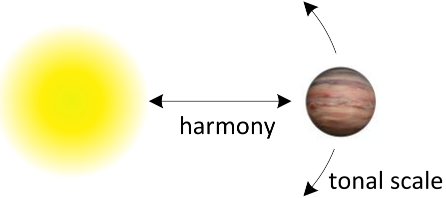


FIGURE 58 JUPITER ('HARMONY')

Saturn controls global rhythmic parameters. When Saturn is near the sun, the global rhythmic accuracy is high and every metric time interval has a constant length (e.g. any 16th is as long as any other 16th). Moving it away from the sun leads to a more loose and imprecise rhythm with random tempo variations. The global tempo can be controlled by moving Saturn clockwise (faster) or counter-clockwise (slower) around the sun. Saturn's visualization on the table reflects both parameters it controls: Every 8th note, it emits a circle which becomes bigger and fades out. This visualizes both the tempo and the rhythmic accuracy: the faster the tempo, the smaller the distance between the circles; a high rhythmic accuracy leads to circles with equal distance.

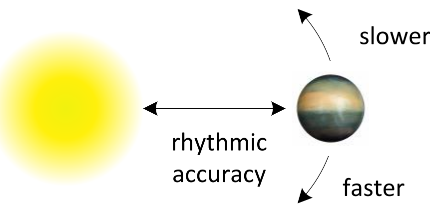
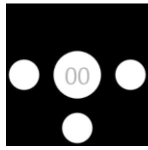
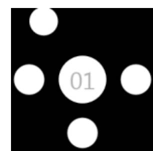


FIGURE 59 SATURN ('TIME')

7.1.2 Realization

We implemented our application for the Microsoft Surface table which recognizes fingers and objects put upon its display. This makes it possible to use new interaction paradigms based on multiple fingers (multitouch) or dedicated physical control objects (tangibles) which are detected either by their form or by a visual tag. The Surface SDK is fully integrated in .NET and provides a core library for all basic functions (like registering for user events) as well as a very high-level library based on WPF (Windows presentation foundation). Many table-specific functions (like tag visualizations or dedicated layout managers) are included and can also be used in Microsoft's WPF editor Expression Blend.





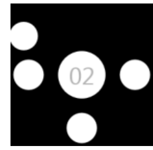


FIGURE 60 MICROSOFT SURFACE TAGS (ORIGINAL SIZE)

Before implementing the system on the table, we have built a prototype with a drag-and-drop mouse interface in order to get a proof-of-concept for the music generation and early feedback from other people. Then, we decided to develop a user interface for the Surface based on tagged physical objects (the planets and the sun) and an additional graphical visualization on the table. Much effort was spent for designing the visual and haptical appearance of the application: before the final design was established, we developed and discussed many alternative versions based on design sketches and prototypes of the tangibles.

7.1.2.1 Tangibles

The planet miniatures are realized as half-spheres made of aluminum. We decided to use this material because it is very robust and has a good haptic quality on the one hand (in contrast to e.g. wood) but still is not too heavy on the other hand (in contrast to e.g. steel). We wanted the tangibles to look like the real planets in our solar system; they are painted using techniques from the area of 'trompe l'oeil'-painting. 'Trompe l'oeil' can be translated as 'trick the eye' and tries to make a painting look like a real thing. It can often be found on the facades of buildings, adding e.g. fake windows or pretending the use of expensive material (like marble). Our tangibles are painted precisely based on satellite images of the real planets. Although being relatively small with diameters of only 5 and 6 cm, many realistic details are captured on them. Since the tangibles are meant to be touched and played with, they are covered with an additional protective layer on top of the painting.

For realizing the sun, we wanted to make use of a transparent material in order to illuminate it using the table's display. Real glass has a very good haptic quality – but it is not robust enough and would also be too heavy for a half-sphere with a diameter of 8.5 cm. We finally decided to use acryl glass which is sandblasted on the spherical side, thus creating a diffuse texture. The sun's flat bottom side is not sandblasted in order to let as much light as possible pass through it.

FIGURE 61 JUPITER TANGIBLE

7.1.2.2 VISUALIZATION

The visualization on the table is designed to look appealing but also to help in understanding how the system works and give additional visual feedback to acoustic events. We implemented it in .NET using WPF (Windows Presentation Foundation) and the Microsoft Surface SDK which provides several table-specific WPF controls.

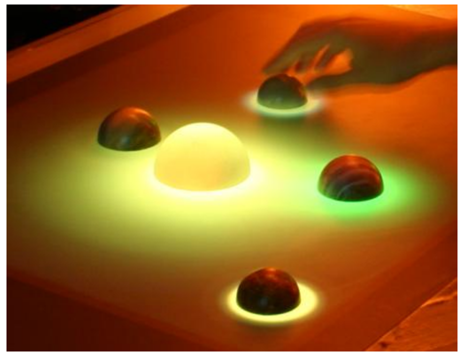


FIGURE 62 TABLE DISPLAY

The main design elements are spheres. The sun is the only tangible object with has a static sphere being constantly displayed below it, filled with a radial gradient fading out to transparency. All other objects (the planets) only emit spheres which stay at their original position. An instrument planet does not have a constant visualization: only when it plays a note, it leaves a sphere at the position where the note was started which becomes bigger and fades

out over time. The sphere does not follow the planet but rather stays in its initial position. This way, a planet being moved on the table leaves a trace of spheres behind it, visualizing the notes it recently played. Jupiter, controlling the global harmony between the planets, emits a filled sphere every 8th note. The sphere's color visualizes the current harmony: when there is high harmony, its color is green. With decreasing harmony, the color continually fades to brown representing 'medium harmony' and then to red, representing 'no harmony at all'. Similarly, Saturn (controlling rhythm) also emits a sphere every 8th note. These spheres are not filled and should resemble Saturn's rings. The distance between these rings visualizes the global tempo (small vs. large distance) and the rhythmic accuracy (equal vs. irregular distance).

7.1.2.3 Music Generation

Based on our framework, the music generation was easy to realize. The concept is very minimal and makes only use of the framework's most basic features. Thus, it serves as a good example for understanding how music can be composed with soft constraints. We will now introduce a formal model of the music generation.

We have a set of stellar objects and a subset of instrument planets (the voices). At each reasoning time, each stellar object has a certain position on the table:

$$StellarObject = \{sun, mercury, venus, mars, neptune, uranus, jupiter, saturn\}$$

$$Voice = \{mercury, venus, mars, neptune, uranus\}$$

$$pos_{Time} : StellarObject \rightarrow Point2D$$

$$x, y : Point2D \rightarrow \mathbb{R}$$

(stellar objects)

The generation of constraints is based on the Euclidean distance between objects:

$$dist : Point2D \times Point2D \rightarrow \mathbb{R}$$

$$dist(p_1, p_2) = \sqrt{(x(p_1) - x(p_2))^2 + (y(p_1) - y(p_2))^2}$$

(Euclidean distance)

The maximum distance between any two objects on the table is defined with this constant:

$$maxDist \in \mathbb{R}$$

(greatest possible distance between objects)

We also need to keep track of an object's rotations around the sun. Computing only the angle between the object and the sun is not sufficient: we also need to detect multiple rotations. For example, rotating an instrument planet around the sun once changes its pitch by one octave – rotating it again changes it by another octave and so on. Tracking rotations around the sun is not complicated, but very technical. To simplify things here, we define a function which computes values from 0 to 1 for each object. This interval already represents multiple rotations:

$$rotation_{Time} : StellarObject \rightarrow [0,1]$$

(rotation around the sun)

Based on an instrument planet's relative position to the sun, a constraint is generated reflecting its preferences for pitch and 'rate of notes'. We use the $energyPitchConstraint$ as defined in section 5.2:

$$planetConstraint_{v,t} = energyPitchConstraint_{v,t}$$

A planet's energy at a given reasoning time is based on its energy at the preceding reasoning time, the last action, the distance to the sun and a constant experimental factor f_1:

$$consume_{Voice,Time} \in \mathbb{R}$$

$$consume_{v,t} = \begin{cases} 1 \; if \; action_{v,t} \in NoteOn \\ 0.5 \; if \; action_{v,t} \in Hold \\ 0 \; else \end{cases}$$

$$energy_{v,t} = energy_{v,t-1} - consume_{v,t-1} + f_1 * \left(maxDist - dist(pos_t(v), pos_t(sun))\right)$$

The pitch constraint is based on the planet's rotation around the sun. The rotation value is mapped to a certain pitch (the 'mean pitch'):

$$getPitch : [0,1] \rightarrow Pitch$$

(convert rotations to pitch values)

Besides this mean pitch, all pitches within a fifth around it can also be played. When Jupiter's distance to the sun is smaller than a constant value $jupiterMaxDist$, the restriction to a tonal scale derived from Jupiter's rotation around the sun is enabled.

$$getScale : [0,1] \rightarrow Scale$$

$$inScale_{Time} : Pitch \rightarrow \{true, false\}$$

$$inScale_t(p) = \begin{cases} true\ if\ distance(pos_t(jupiter), pos_t(sun)) > jupiterMaxDist \\ p\ in\ getScale(rotation_t(jupiter))\ else \end{cases}$$

$$possiblePitches_{Voice,Time} \subseteq Pitch$$

$$possiblePitches_{v,t} = \{\, p \in Pitch : \left[\, p, getPitch\big(rotation_t(v)\big) \,\right] < fifth \wedge inScale_t(p)\}$$

The pitch constraint is then defined as follows:

$$pitchConstraint_{Voice,Time} : (Voice \rightarrow\ Action) \rightarrow \mathbb{R}$$

$$pitchConstraint_{v,t}(val) = \begin{cases} 1\ if\ val(v) = pause \\ 1\ if\ pitch\big(val(v)\big) \in possiblePitches_{v,t} \\ 0\ else \end{cases}$$

We use the $harmonyConstraint$ defined in section 5.2 for optimizing the harmony between the instrument planets. This constraint is weighted with an experimental factor f_2 and Jupiter's inverse distance to the sun:

$$planetHarmony_{Time} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$$

$$planetHarmony_t(val) = f_2 * (maxDist - dist(pos_t(jupiter), pos_t(sun))) * harmonyConstraint(val)$$

(harmony between the planets)

The final constraint problem is then defined by summing up the harmony constraint and all planet constraints using the monoid of real numbers with addition $(\mathbb{R}, +, \leq, 0)$:

$$thePlanets_{Time} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$$

$$thePlanets_t(val) = planetHarmony_t(val) + \sum_{v \epsilon Voice} planetConstraint_{v,t}(val)$$

(the final constraint problem)

The implementation is realized in a straightforward way based on this formal model. The central classes are `PlanetInstrument` subclassing `CoordinatedInstrument` and `Planets` subclassing `Reasoner`. The communication between the user interface and the music generation is bidirectional: the positions of the tangibles are passed to the music generation and playing notes are passed to the user interface. This is realized with the interface `StellarObject` and its subclasses `Planet` and `Sun`. These interfaces are implemented by the user interface which keeps a stellar object's position and state up to date. This data is processed by the music generation. Playing notes are communicated to the corresponding planet by calling the method `Blink`. This way, the bidirectional communication between user interface and music generation is realized in a simple way with shared communication objects and the music generation has no dependency to the user interface.

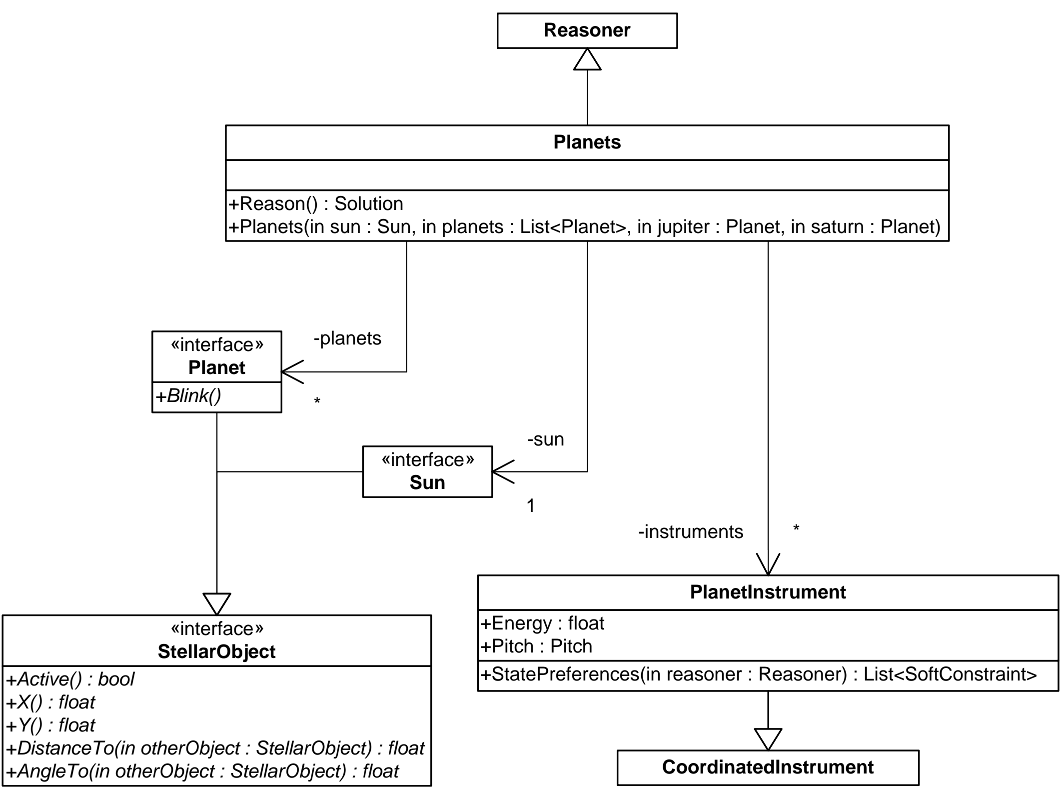


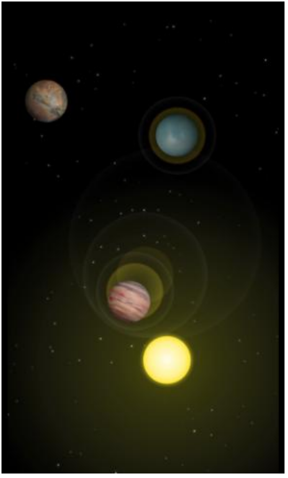
FIGURE 63 THE PLANETS CLASS DIAGRAM

7.1.3 THE PLANETS FOR WINDOWS PHONE 7

We recently implemented The Planets for Windows Phone 7. In this application, the planets can now be moved on the phone's display with multitouch interaction (i.e. several planets can be moved at the same time with multiple fingers). When porting the application, the following design constraints showed up: First, the available space on the display is very small – especially compared to the surface table. Second, we want the planets to move in a natural way with no objects being at the same place at any time (which is an inherent restriction for tangibles). Third, there is no standard software synthesizer available for generating audio from note information.

We could re-use the existing music generation component without any change. However, we decided to implement a new user interface based on Microsoft's .NET game engine XNA (93). We made much use of so-called particle effects for creating visualizations of e.g. playing notes or the constantly changing stars in the background (based on the Mecury Particle Engine (94)). In order to save space on the display, we decided to omit the two control planets Jupiter and Saturn. Saturn's functionality for controlling the tempo has been completely omitted. Jupiter's functionality for controlling tonal scales has also been removed, but we definitely wanted to

keep the possibility to control the harmony of the music. We decided to use the phone's accelerometer for controlling this parameter: the harder one shakes the phone, the more dissonant notes are played. This is also accompanied by a matching visualization (see Figure 64).



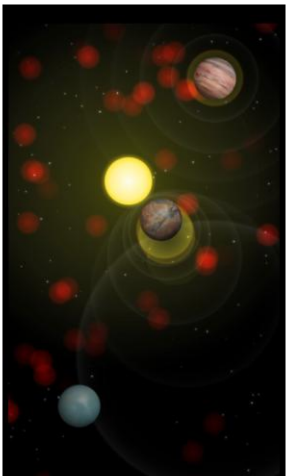


FIGURE 64 LEFT: REGULAR SCREEN, MIDDLE: DISSONANCE, RIGHT: SOUND SELECTION

In order to create a natural behavior of the planets and to keep them separated, we decided to use the Farseer physics engine (95). All planets and the sun are controlled by a realistic physical simulation: it is possible to push them around on the display and they also bump into each other. For creating audio from notes, we have built a simple sound generator based on pre-recorded audio files (a so-called 'sampler'). These short audio files ('samples') cover individual notes from a certain instrument which are transposed to a desired note pitch and written to a buffered audio stream. For each planet, three different sounds can be selected: Bell, Piano and Guitar.

7.1.4 Related Work

There is much related work where musical applications are controlled by tangible user interfaces: for example, the famous reacTable (96) provides a completely new paradigm for interacting with a modular synthesizer, Audiopad (97) is based on arranging samples and Xenakis (98) allows composing music with probability models (to name just a few). Above all, the work of Toshio Iwai was a great inspiration: Elektroplankton (99) for the Nintendo DS handheld gaming console offers a collection of several musical mini-games and allows creating music in a very playful way. Another system designed by T. Iwai is Yamaha's Tenori-On (100) which provides a matrix of 16x16 lighted buttons that control several intuitive music generation applications. The employment of planets (which have no direct connection to music) was inspired by the orchestral suite 'The Planets' from Gustav Holst: every planet in our solar system is portrayed with a musical piece which reflects its special astrological character. The earth is not included in Holst's composition so we also decided to omit it.

'The Planets' provides direct control over the shape of melodies and is based on high-level concepts of music theory like tonal scales or the harmony between notes. To our knowledge, there is no other system where the amount of harmony between several simultaneously playing voices can be controlled in a continuous way.

7.2 Fluxus Pattern Sequencer

In contrast to the other applications presented in this work, the *Fluxus* pattern sequencer is an application geared towards musicians. It was designed to provide a convenient interface to the musical framework's basic features like training musical models. Besides this, it also provides additional functionality like recording of static sequences. The two main areas of application are:

- Creating musical 'styles' for applications geared towards non-musicians
- Live performance of music with a focus on improvisation

Interactive applications for non-musicians based on the Fluxus sequencer were exhibited at the 'Komma' trade fair for communication and marketing in Munich and at the concert series 'Zukunft(s)musik' in Augsburg (both in 2011).

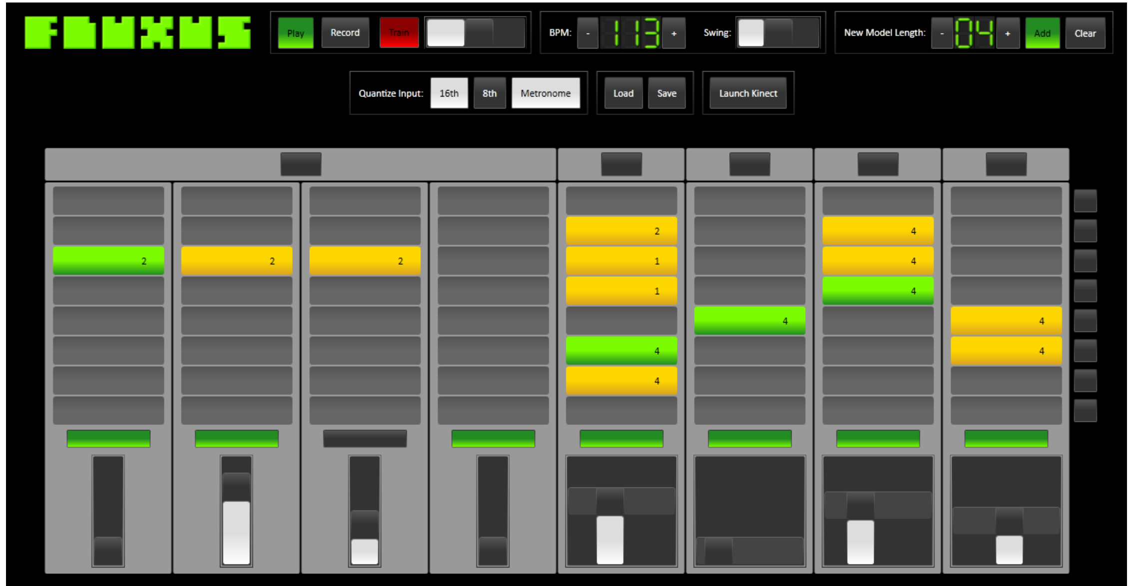


FIGURE 65 FLUXUS PATTERN SEQUENCER

The Fluxus sequencer is based on so-called *patterns*: A pattern is a rather short snippet of music which has a length of only few bars (for example 4 bars). A *pattern sequencer* allows creating, organizing and playing back these patterns for one or several instruments. Fluxus provides the novel possibility to also train musical models that can be interactively 'played', i.e. varied in both rhythm and pitch. It soon turned out that a key requirement for this system would be a seamless integration with all the conventional things a typical pattern-based MIDI sequencer provides. Playing only trained models sounds much too unorganized in most cases: it should also be

possible to record static sequences and switch between them and interactive models at any time. Much effort was spent for designing the system's user interface since we wanted to build a system with short and direct control that can also be used in an improvisational performance. The system can be controlled with a mouse or a touch screen. Furthermore, it is also possible to control most functions with generic hardware MIDI controllers. The pattern sequencer allows creating musical styles that can be used in other applications. A whole set of patterns, including static background music as well as training data for several players, can be exported to a single file and imported by another application, for example a casual game based on motion tracking.

In the next subsection, we will introduce the pattern sequencer from a user's point of view. Then, we will take a closer look at the system architecture and its implementation. At last, we will give a short overview on the history of music sequencers and introduce related approaches.

7.2.1 Concept

The Pattern Sequencer has four rhythmic tracks and four tonal tracks. Each track can hold several patterns, each consisting of a *loop* holding static sequence data and a *model* holding dynamic training data. Loops and models can be recorded respectively trained by playing notes on an external keyboard (or any other MIDI device). Trained models can be used to interactively generate new melodies from them: the playback speed (rate of notes) can be controlled with a parameter *energy*; the desired note height can be controlled with a second parameter *pitch*. The music is then generated according to these parameters such that it also sounds similar to the training melodies.

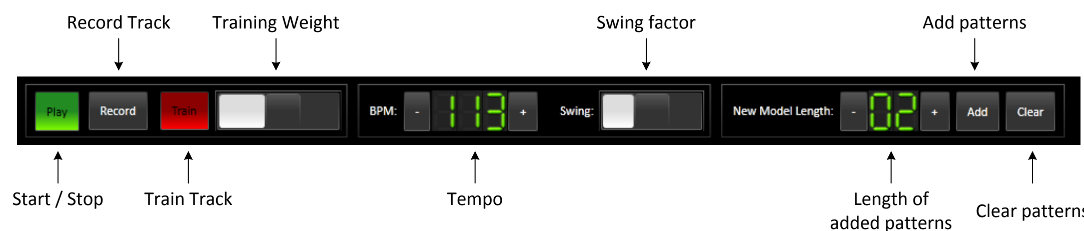


FIGURE 66 TOOLBAR

At the top, the toolbar can be found. It provides global controls for starting and stopping the sequencer, recording loops and training models. The tempo can be set in BPM (beats per minute) and a swing factor can be set (deferring every odd 16th note by a certain amount). At the right, there are controls for adding and removing patterns. Below, there are additional

controls for saving or loading the sequencer's state as well as several helpers (e.g. a metronome and optional buttons for launching external controllers).

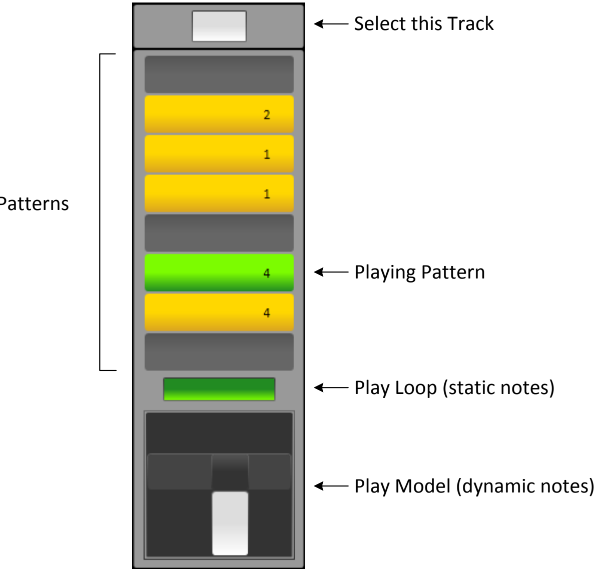


FIGURE 67 TRACK VIEW

Below the toolbar, four rhythmic and four tonal tracks can be found. The button at the top selects a track: all MIDI input (e.g. from a connected keyboard) is routed to the selected track and can be used for recording and training patterns. The rhythmic tracks are always selected altogether in order to be able to record or train all of them at once; tonal tracks are selected individually. Each track can hold up to eight patterns of variable length. Adding new patterns is done by selecting *Add* and setting the desired length in the toolbar. Then, clicking an empty pattern slot will insert a new pattern. Vice versa, existing patterns can be removed by selecting *Clear* and clicking on the patterns to remove. The currently playing pattern is marked in green; clicking on another pattern starts it at the next bar change. On the right, there are buttons which allow starting a complete row of patterns for all tracks at the same time. Below the patterns, there is a button which allows turning playback of the static loop on or off. At the bottom, playback of the model can be controlled: a rhythmic model's energy is controlled with a regular fader. Tonal models are controlled with a two-dimensional pad: the vertical position controls the energy, the horizontal position the pitch. This way, both parameters can be controlled fast und intuitively with a single finger. Recording loops is done in the obvious way by pressing the *Record* button. Similarly, training a model is done by pressing the *Train* button. This is always done with a certain weight (controlled by a fader): Using a high weight makes training notes have a strong influence on existing training data. Vice versa, low weights do only slightly change an existing model. Note that there are no separate controls for switching between a

loop, a model and live input, e.g. from a MIDI keyboard. Instead, we use a hierarchical 'Barge-In'-system for intuitively switching between them: Playing a model by increasing its energy mutes the loop. Both model and loop are being muted when a track is played using live input.

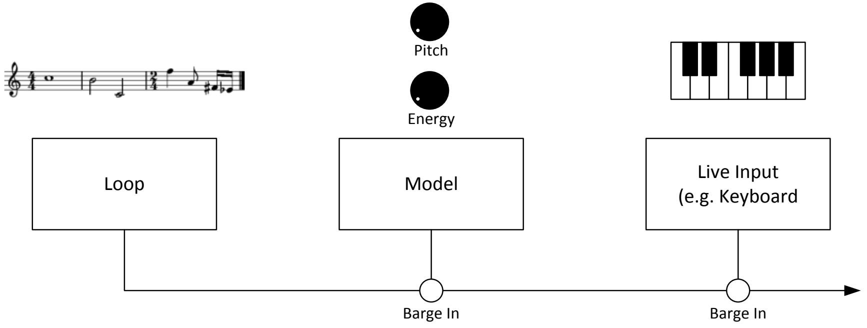


FIGURE 68 BARGE-IN HIERARCHY

7.2.2 Realization

Like the musical framework itself, the Fluxus sequencer is also written in .NET with C#. It was developed in an iterative process with continuous evaluation, improvement and refactoring.

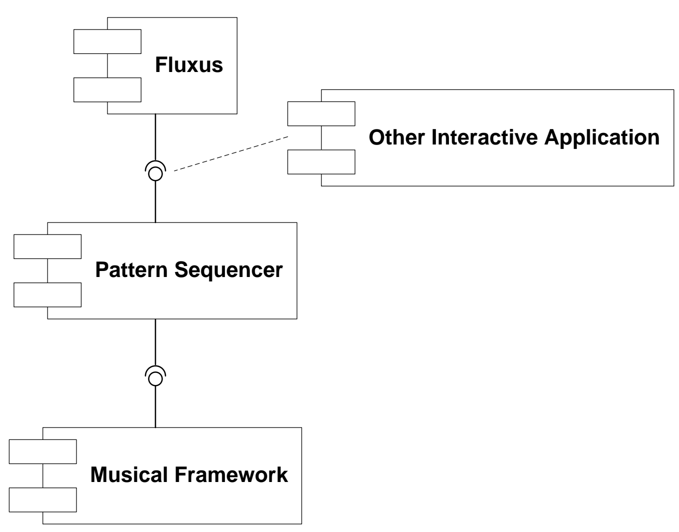


FIGURE 69 PATTERN SEQUENCER ARCHITECTURE

Based directly on the musical framework, a component *Pattern Sequencer* implements the application logic, i.e. the whole musical functionality: patterns, tracks, loops and models, train and record, playback, save and load data and so on. The component *Fluxus* provides a user interface to the sequencer. The pattern sequencer component can also be used by another application, e.g. geared towards non-musicians. This way, musical models and loops can be created by musicians using Fluxus and then used by non-musicians in another application.

7.2.2.1 MUSIC GENERATION

In this section we will introduce the general architecture of the pattern sequencer component and take a closer look at how trainable musical models are employed.

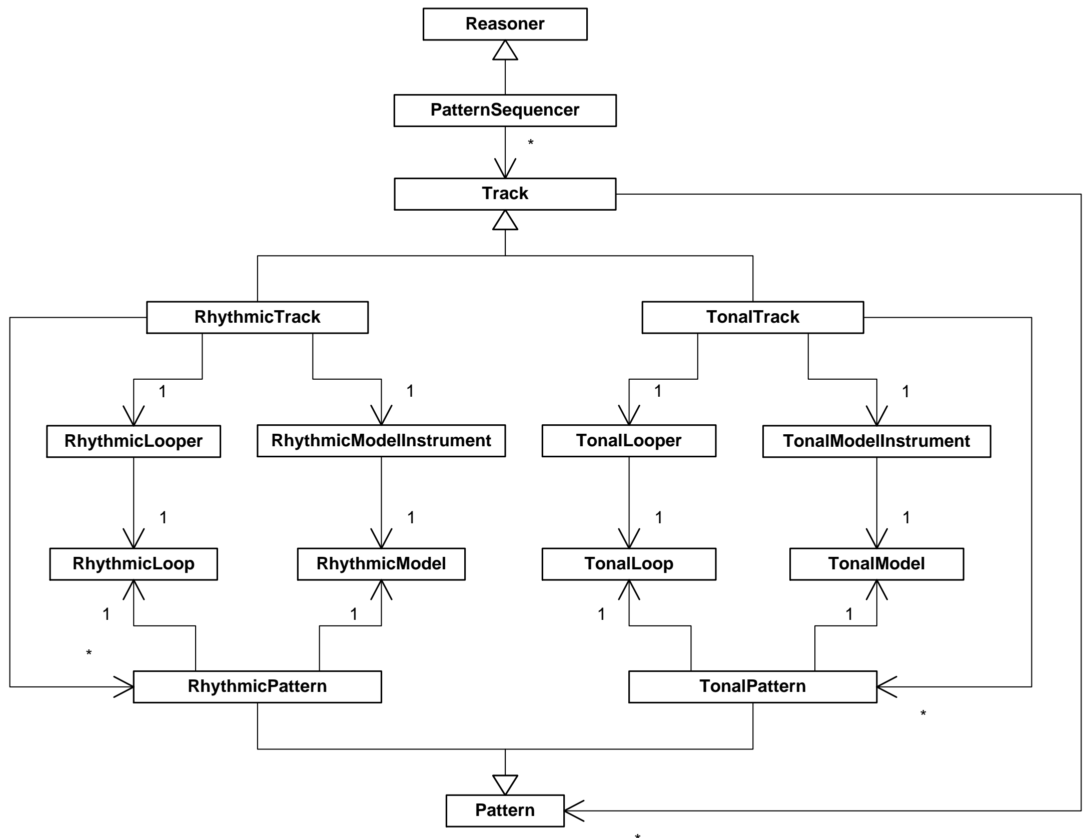


FIGURE 70 PATTERN SEQUENCER ARCHITECTURE

`PatternSequencer` is a subclass of `Reasoner`. This class holds references to `Tracks`, which can be either rhythmic or tonal. Each track has a looper for static sequence data and a trainable instrument; these can also be rhythmic or tonal: `TonalModelInstrument` and `RhythmicModelInstrument` are presented in detail in section 6.2.4. A `Track` holds

several `Patterns` which consist of a loop as well as a model having the same length. The `Track` classes handle all functionality related to playback, recording and training. The barge-in system is also implemented in tracks.

The generation of music from trainable models is based on the $energyPitchConstraint$ as introduced in section 5.2. At each step, a certain amount of energy is added to a track, controlled by the Fluxus user interface's faders or any other interaction paradigm in another application (e.g. the speed of hand movement detected by a motion tracking system):

$$addedEnergy_{Voice,Time} \in \mathbb{R}$$

Whenever a note is started or being held, a certain amount of energy is subtracted ('consumed') from the track like this:

$$consumedEnergy_{Voice,Time} \in \mathbb{R}$$

$$consumedEnergy_{v,t} = \begin{cases} 1 \; if \; action_{v,t} \in NoteOn \\ 0.5 \; if \; action_{v,t} \in Hold \\ 0 \; else \end{cases}$$

The track's current energy is computed from the energy at the last step, the amount of consumed energy and the amount of added energy:

$$energy_{v,t} = energy_{v,t-1} - consumedEnergy_{v,t-1} + addedEnergy_{v,t}$$

The pitch constraint for tonal tracks is generated from a single number which can be set with the two-dimensional pad or by a different interaction paradigm in another application, e.g. the current height of a user's hands:

$$pitch_{Voice,Sample} \in \mathbb{R}$$

This number is converted to a note pitch e.g. such that all possible pitches can be reached or, if desired, such that the pitches remain in a desired range:

$$getPitch : \mathbb{R} \rightarrow Pitch$$

A set of all possible pitches is computed by taking all pitches within a certain interval around the root pitch (we use a fifth):

$$possiblePitches_{Voice,Time} \subseteq Pitch$$

$$possiblePitches_{v,t} = \{\, p \in Pitch : \left[\, p, getPitch\left(totalPitch_{v,t}\right)\right] < fifth\}$$

This set is used to define the pitch constraint:

$$pitchConstraint_{Voice} : (Voice \rightarrow\ Action) \rightarrow \mathbb{R}$$

$$pitchConstraint_{v,t}(val) = \begin{cases} 1\ if\ val(v) = pause \\ 1\ if\ pitch\left(val(v)\right) \in possiblePitches_{v,t} \\ 0\ else \end{cases}$$

The constraint for a track is defined as a combination of the $energyPitchConstraint$ and the $modelConstraint$ as defined in section 5.3:

$$trackConstraint_{Voice,Time} : (Voice \rightarrow\ Action) \rightarrow \mathbb{R}$$

$$trackConstraint_{v,t}(val) = energyPitchConstraint_{v,t}(val) \\ * modelConstraint_{v}\left((t-1)\ mod\ |Step|, action_{v,t-1}\,, t\ mod\ |Step|\right)(val)$$

The final constraint problem is a combination of all track constraints and an optional harmony constraint:

$$patternSequencer_{Time} : (Voice \rightarrow\ Action) \rightarrow \mathbb{R}$$

$$patternSequencer_{t}(val) = harmonyConstraint(val) + \sum_{v \in Voice} trackConstraint_{v,t}(val)$$

7.2.2.2 User Interface

The User Interface was developed with WPF based on the Model-View-ViewModel (MVVM) pattern, which allows a very clear separation between visual elements and presentation logic. Like most user interface design patterns, MVVM also consists of three layers: the application logic and data (here: the pattern sequencer component) is called the *Model*. The *ViewModel* encapsulates the Model and (re-)organizes it such that it represents the application's conceptual model from the point of view of a user. The *View* defines the visual appearance of the application. It consists of visual elements (e.g. buttons) and user interface logic which has nothing to do with the application itself (e.g. mouse-over effects or animations). In WPF, the View is usually written in the declarative language *XAML* (which supports graphical editors) and additional programming code (C#). In contrast to other common user interface patterns (e.g. Model-View-Controller or Model-View-Presenter), the layer between View and Model has no dependency to the View. Instead, controls in the View are only loosely coupled to the ViewModel via the Command pattern and data bindings. This results in a very clear conceptual model of the user interface and the presentation logic which is easy to maintain because it is completely separated from graphical programming code. The following UML diagram shows how the Fluxus user interface is implemented based on the MVVM pattern. For simplicity, the distinction between rhythmic and tonal has been omitted:

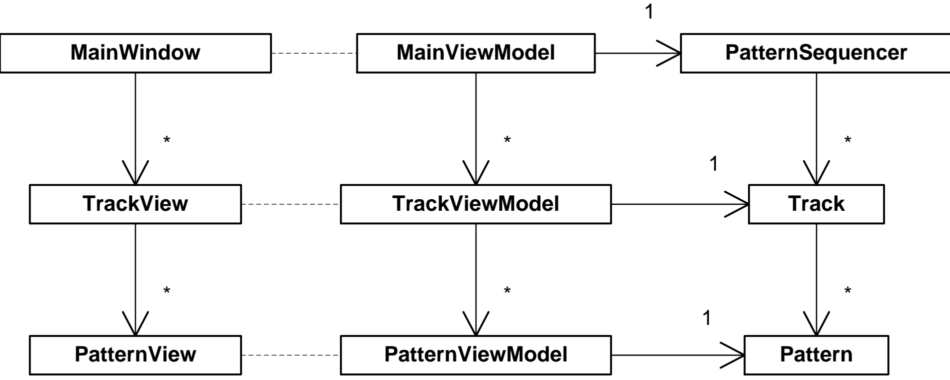


FIGURE 71 FLUXUS USER INTERFACE (MVVM PATTERN)

Each ViewModel class encapsulates the corresponding Model class and exposes only properties and commands which are relevant for the user interface. As an example, the `MainWindow` class is loosely coupled to the `MainViewModel`. The `MainViewModel` exposes a property `Play` which gets or sets the playback state:

```
public bool Play
{
    get { … }
    set { … }
}
```

The actual control for the playback state is defined in the `MainWindow` (using the XAML markup language) and loosely references the property `Play` in the corresponding ViewModel:

```
<ToggleButton IsChecked="{Binding Play}" … >Play</ToggleButton>
```

The user interface can be controlled by a mouse as well as a touch screen. Furthermore, it is also possible to use external MIDI hardware controllers in order to achieve a more direct and tactile user experience.

Currently, two controllers are directly supported: Novation launchpad and Korg Nano control. The selection of patterns can be controlled with the Novation launchpad which also supports visual feedback from the application. With a matrix of 8 x 8 multi-color LEDs, patterns can be selected for each track. The color of the LEDs are the same as in the application: 'off' visualizes an empty pattern slot, 'yellow' an existing pattern and 'green' a currently playing pattern. The Nano control provides transport buttons which are used to control the playback state (play, record, train, set tempo). Furthermore, it provides eight groups each having a fader, a dial and two buttons. Each group corresponds to a track: the fader controls 'energy', the dial 'pitch' (for tonal tracks) and the two buttons 'track selection' and 'loop playback'.

The MVVM pattern makes it easy to integrate external controllers and maintain a consistent state between the controller and the regular user interface. This is achieved by connecting a single ViewModel to the regular control in the application as well as to the external control. This way, changes in the ViewModel are automatically kept consistent for both interfaces.

7.2.3 Related Work

Musical automata were among the first programmable systems ever (2), using e.g. pin rollers or punch cards to store note information. These mechanical machines can be seen as the first sequencers and were capable of controlling for example flutes, bells or percussion instruments. In the 19th century, Thomas Alva Edison was one of the first to record and reproduce sounds. However, recording sound is not a matter of concern here: we are rather interested in ways of recording and editing abstract note information.

The first electronic sequencers came up in the 1960's along with commercially available synthesizer systems (e.g. from Moog, Buchla or Roland). These early sequencers were so-called step sequencers which have a series of columns (typically 8 or 16), each representing a certain metric position in a bar (a step). Each row provides a control element for every step: switches for binary information (e.g. note on/off) and dials for continuous information (e.g. note pitch). These sequencers are often components of modular synthesizer systems and as such not

restricted to control only notes. Instead, they can rhythmically change any available parameter in the system, for example aspects of a sound's timbre. Early popular electronic music was strongly influenced by step sequencers: the sound of bands like Kraftwerk or Tangerine Dream is above all characterized by frequently repeating sequences. Today, these simple step sequencers are still very popular and there are lots of new systems available (despite or even because of their restrictions and simplicity). The first systems which could store several patterns in a digital memory were simple drum machines. In the early 1970's, drum machines came up which had a set of predefined rhythms that could only be selected but not edited. A few years later, it became possible to store and recall user-defined rhythms. Above all, the Roland TR-series of rhythm machines was (and still is) very popular. A set of 16 buttons represents the metric positions in a bar: pushing a button makes the selected instrument play a note at this position. There are also similar concepts for monophonic tonal instruments like the Roland TB-303 synthesizer.

In the early 1980's, when personal computers became available, the first software sequencers for standard hardware came up (e.g. by Karl Steinberg who later developed Cubase which is still one of the most popular software sequencers in its current version). Arbitrary sound generators can now be connected over the MIDI protocol (Musical Instrument Digital Interface) which became a widespread standard. These sequencers are based on a very different approach: instead of patterns, they use a linear timeline along which a whole song can be arranged. The raw division into rather big steps (e.g. 16th notes) of early systems was improved with a far more precise resolution. With increasing computing power, software sequencers also became very powerful: today, digital audio workstation software contains the functionality of a whole recording studio – it is possible to record multiple audio tracks at the same time, apply effects or play complex software-synthesizers in real-time. Besides Steinberg Cubase, there are many other sequencers available (for example Apple Logic, Cakewalk Sonar or Cockos Reaper to name just a few). Another recent sequencer is Ableton Live which returned to the pattern-based concept of early systems and has its focus on arranging and mixing patterns in a live performance. In our application, the organization of patterns in a matrix of tracks and patterns is inspired by Ableton Live.

The Fluxus pattern sequencer is based on a novel and unique concept: it integrates static sequences and dynamic models which can be recorded respectively trained while the system is running. This makes it possible to interactively generate variations of melodic material, for example in an improvisational performance. Models trained with the Fluxus sequencer can also be used in other interactive applications, e.g. targeted at non-musicians.

7.3 Transforming Spatial Movements to Music

In this section, we describe a general approach for transforming spatial movements into music. Based on this general approach, we implemented two concrete applications: The first one is controlled with a touch display, the second one with body movements. The application based on touch interaction was exhibited at the 'Komma' trade fair for communication and marketing in Munich and at the concert series 'Zukunft(s)musik' in Augsburg (both 2011).

FIGURE 72 CONTROL MUSIC WITH BODY MOVEMENTS

7.3.1 Approach: Generating Music based on Spatial Movements

In general, we have a set of moving 'objects' which control the music. Adopting terminology from the area of computer vision, we call these objects *features*. Typical features are a user's body parts, for example a finger moving on a display or a whole hand moving in three-dimensional space. A single interactive voice can be controlled by a single feature as well as by multiple features at the same time. Furthermore, a single person can also control multiple voices simultaneously by multiple (typically distinct) sets of features. In general, we have a set of features for each voice corresponding to the spatial objects which are used to control it:

$$Feature_{Voice} \subseteq Feature$$

These features are tracked with a certain sampling rate. Just like the set of reasoning times $Time$, the tracking samples are also modeled with natural numbers:

$$Sample = \mathbb{N}$$

There is no quantitative correspondence between these sets, but we nevertheless want to compare elements of $Time$ with elements of $Sample$. We hence define the comparison operators ($<, \leq, \ldots$) such that they compare with respect to the elements' actual time of occurrence – regardless of their discrete sequential numbers. The tracking of features is typically done with a much higher sampling rate than the reasoning, i.e. in most cases there will be several tracking samples between two reasoning times. For a given reasoning time, the following function returns all samples since the last reasoning time:

$$samples : Time \rightarrow \mathrm{P}(Sample)$$

$$samples(t) = \{s \epsilon Sample : s < t \wedge s \geq t - 1\}$$

(samples between t and $t - 1$)

For each sample, we can determine the position of each feature as a vector. Without loss of generality, we use a three-dimensional vector space here:

$$pos_{Sample} : Feature \rightarrow Point3D$$

$$x, y, z : Point3D \rightarrow \mathbb{R}$$

(position of a feature at a given sample)

We compute distances between points with the Euclidean distance:

$$dist : Point3D \times Point3D \rightarrow \mathbb{R}$$

$$dist(p_1, p_2) = \sqrt{(x(p_1) - x(p_2))^2 + (y(p_1) - y(p_2))^2 + (z(p_1) - z(p_2))^2}$$

(Euclidean distance)

For each voice, we compute the amount of movement between two sequential samples. This is done by summing up all distances between the old and new positions for each of the voice's features:

$$movement_{Voice,Sample} \in \mathbb{R}$$

$$movement_{v,s} = \sum_{f \in Feature_v} dist(pos_s(f), pos_{s-1}(f))$$

(movement of all features between two samples)

Given a voice and a reasoning time, we sum up the movement for all samples since the last reasoning time:

$$totalMovement_{Voice,Time} \in \mathbb{R}$$

$$totalMovement_{v,t} = \sum_{s \in samples(t)} movement_{v,s}$$

(movement of all samples between two reasoning times)

We generate music based on the $energyPitchConstraint$ as introduced in section 5.2. Given a voice and a reasoning time, the energy for this time is computed based on the preceding energy, the amount of energy consumed by the preceding action and the sum of all movements since the last reasoning time (scaled by an experimental factor):

$$consume_{Voice,Time} \in \mathbb{R}$$

$$consume_{v,t} = \begin{cases} 1 \; if \; action_{v,t} \in NoteOn \\ 0.5 \; if \; action_{v,t} \in Hold \\ 0 \; else \end{cases}$$

$$energy_{v,t} = energy_{v,t-1} - consume_{v,t-1} + f_1 * totalMovement_{v,t}$$

The voice's pitch is controlled similarly by averaging over all samples and features:

$$pitch_{Voice,Sample} \in \mathbb{R}$$

$$pitch_{v,s} = \frac{1}{|Feature_v|} \sum_{f \epsilon Feature_v} y\big(pos_s(f)\big)$$

$$totalPitch_{Voice,Time} \in \mathbb{R}$$

$$totalPitch_{v,t} = \frac{1}{|samples(t)|} \sum_{s \,\epsilon\, samples(t)} pitch_{v,s}$$

The resulting number is converted to an actual pitch such that it gets in the desired range:

$$getPitch : \mathbb{R} \rightarrow Pitch$$

We define a constraint that restricts pitch to be within e.g. a fifth around this mean pitch:

$$possiblePitches_{Voice,Time} \subseteq Pitch$$

$$possiblePitches_{v,t} = \{\, p \,\epsilon\, Pitch : \big[\, p, getPitch\big(totalPitch_{v,t}\big)\big] < fifth\}$$

$$pitchConstraint_{Voice} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$$

$$pitchConstraint_{v,t}(val) = \begin{cases} 1 \;\; if\ val(v) = pause \\ 1\ if\ pitch\big(val(v)\big)\ \epsilon\ possiblePitches_{v,t} \\ 0\ else \end{cases}$$

The complete constraint for one voice is a combination of the $energyPitchConstraint$ and the $modelConstraint$ as defined in section 5.3:

$$motionConstraint_{Voice,Time} : (Voice \rightarrow Action) \rightarrow \mathbb{R}$$

$$motionConstraint_{v,t}(val) = energyPitchConstraint_{v,t}(val) * modelConstraint_v\big((t-1)\ mod\ |Step|, action_{v,t-1}, t\ mod\ |Step|\big)(val)$$

The final constraint problem is a combination of all motion constraints with an optional harmony constraint:

$$motionToMusic_{Time} : (Voice \rightarrow\ Action) \rightarrow \mathbb{R}$$

$$motionToMusic_t(val) = harmonyConstraint(val) + \sum_{v \epsilon Voice} motionConstraint_{v,t}(val)$$

7.3.2 Applications: Touch Display and Motion Tracking

We implemented our approach for transforming spatial movements to music based on two different interaction paradigms: First, we implemented a user interface based on a touchscreen where the user can play music by moving his finger: the faster he moves, the more notes are being played; the notes' pitches depend on the finger's vertical position. Then, we implemented a user interface based on a motion tracking system where the music is controlled by body movements (we use Microsoft's markerless tracking system Kinect). The music can be controlled with a set of body features: The faster these features are moving, the more notes are being played; their average vertical position controls pitch.

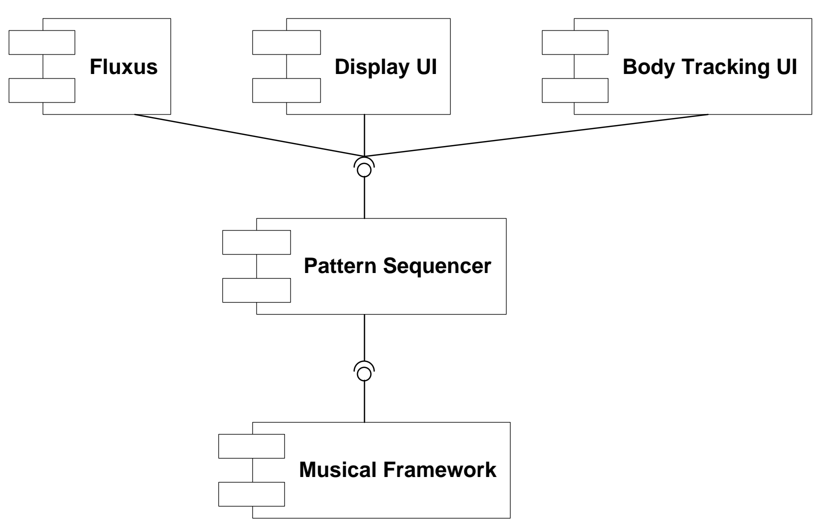


FIGURE 73 SYSTEM ARCHITECTURE

Both applications use the Pattern Sequencer component as described in the previous section and generate music based on models which can be trained with the Fluxus application. Only few lines of code were required for the music generation: both touchscreen and Kinect provide a callback function which updates (among other things) the current position of all features. We just sum up the distances for all relevant features' positions to the last update and use this to set the 'energy'. The average vertical position of all features is used to set 'pitch'. For both applications, the code dealing with music generation is less than 10 lines.

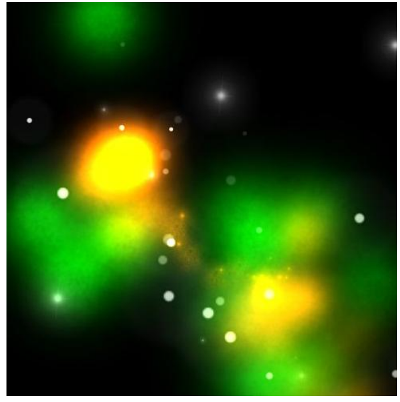


FIGURE 74 PARTICLE EFFECTS

The most code was required for realizing the graphical part of the user interface. We considered it very important to provide a visualization fitting to the music which helps understanding how the system behaves and creates an appealing audio-visual experience. Our visualization is mainly based on so-called 'particle effects' which are often used in video games to create 'blurry' effects (for example fire, smoke or blood). We use these effects (created by the 'Mercury Particle Engine' (94)) to visualize the user's movements as well as notes resulting from interaction. For example, the user's movements can be followed by a rather subtle and slowly moving trail of particles whereas playing notes can be visualized with a fast moving and rather drastic effect (similar to an explosion). Besides an appealing look which is easy to achieve, particle effects also partially compensate the latency of a touch display and – even more – body tracking because of their blurry nature.

7.3.3 Related Work

Several related approaches and systems have been described where music is generated based on spatial movements of body parts or the general physical body state. In (101), a system is described which allows users to use their full body for controlling the generation of an audio-visual feedback in real-time. The system extracts motion features from the user's body movements and maps them to acoustic parameters for rendering a piece of music as well as additional visual feedback projected on a screen in front of the user. The Impromptu Conductor (102) is a system for mapping hand movements to music based on a supervised learning method called 'pattern-recognition'. The Cyber Composer (103) generates music according to hand motions and gestures. Musical expressions like the pitch, rhythm and volume of a melody can be controlled and generated in real-time by wearing a pair of motion-sensing gloves. In (104), a system for musical performance is patented based on user input and stored original music data representing a music piece. A user's physical actions and physiological state are acquired and used to alter the stored tones. Similarly, in the patent (105), sensors are used to assess a user's physical condition which alters a stored piece of music. This gives constant acoustic feedback to the user and helps him to achieve a certain desired behavior in a training or therapy context. Another tool targeting the same application context is described in (106): pressure-sensitive controls allow even people with severe disabilities to control the generation of music. The system introduced in (107) uses a performance device (e.g. based on hand-proximity) to interactively control several aspects of a composition algorithm. When no input is provided, the system proceeds automatically to compose music and produce sound. A general-purpose position-based controller for electronic musical instruments is described in (108). The position signal may be used for generating music or for applying effects to the output of another instrument.

Our approach for generating music from spatial movements is mainly based on the location and velocity of body features. Other approaches provide control on a higher level of abstraction based e.g. on gesture recognition – it would be interesting to investigate how this could be integrated in our framework. We introduce a general and modular way for transforming spatial movements to music within our framework for composing music with soft constraints: the user interaction paradigm, i.e. the mapping of raw sensor readings to musical preferences is specified in a declarative way and does not depend on other preferences. General musical preferences can be specified with additional constraints optimizing e.g. the similarity to a musical model or the harmony between several voices. This way, applications which generate music based on spatial movements can be composed in a very modular way with loose coupling between functionality dealing with user interaction and functionality dealing with other musical rules.

7.4 Interactive Advertising Jingles

This chapter is based on joint research with Gilbert Beyer from LMU Munich on how interactive music can be used within the area of pervasive advertising (109) (110) (111) (112). In these works, we participated in developing basic requirements and design constraints for interactive music systems for advertising purposes and contributed to the conception of the described prototype systems. We realized these systems based on our framework for composing music with soft constraints and trainable transition models.

Music has been used for advertising purposes for a long time, e.g. in the form of advertising jingles or background music in shopping malls. Nowadays, sounds are also used within interactive media in the internet and in digital signage. Yet, there have only been limited attempts to include sound itself to the interactive experience. In this chapter, we present an approach that enables users to interactively play advertising music. This approach meets two important requirements: First, the user should have control over the music. Second, the music should still be recognizable as a given brand melody. To our knowledge, there is currently no related work describing the combination of music generation and interactive advertisements. There exist many articles on specific sound branding issues in classical and digital media, but they do not cover the field of user-controllable brand music. No work so far focused on how to control the brand music itself within the interactive experience, while the same is often done with visual elements of the brand identity. For a general survey on the topic of sound branding we refer to (113) and (114). At first, we assess requirements and design issues for several potential application areas for interactive advertising music. Then, we present our general approach for realizing interactive advertising jingles that can be interactively played on the one hand but still remain recognizable on the other hand. We present a prototype based on public displays and close with results from a user study.

7.4.1 Requirements and Application Areas

Along with the emergence of interactivity in common media, advertising has also become interactive, for example within interactive internet banners, advertisements in video games or public installations. Today, there exist a variety of platforms which are potentially suitable for advertising involving user-controllable music. In general, we see two different approaches for using interactive music in advertising: On the one hand, it can supplement other content, e.g. as part of a company´s web page or an interactive application running on a public display. In this case, interactive music has the function to enhance the overall experience. On the other hand, an interactive music application can also be the primary part of an advertisement, for example in the form of a free application for mobile phones which is fun to play with and hence will be used voluntarily by people. We see a variety of application areas where interactive music can be employed for advertising:

Online Advertising: In online advertising, the enrichment of ads by sound and interactivity is already common. In general, all kinds of rich media advertisements can be used to integrate interactive music, e.g. banners, skyscrapers or floating ads. Microsites can also be an appropriate platform for interactive music since they are self-contained areas within a larger web presence which are typically used for advertising particular products, sweepstakes or promotional events. There are also modes of application where interactive music might be inappropriate: in general, online advertisements should only include sound which is triggered by user interaction, i.e. they should be silent unless being explicitly started. Advertisements which can be closed by the user (e.g. peel backs or interstitial ads) might also be unsuitable because they might be clicked away before the user understands that he can interact with the music.

Mobile Advertising: Interactive music applications can be used well on mobile devices which typically provide many possibilities for designing user interaction: from common controls like buttons or touch displays to specialized functionality like acceleration sensors or global positioning. Musical applications are very popular on such devices and there exist a large number of applications for all common platforms. On the one hand, there are professional applications designed for musicians and on the other hand - much more interesting for advertising - simple applications targeted at non-musicians. Musical applications on mobile phones have already been used for advertising purposes: for example, Audi gives away an app for the iPhone which is a combination of car racing and a rhythmic game. With another app from Procter and Gamble, one can play drums on Pringles chips cans.

In-Game Advertising: Advertisements in video games are an expanding market. Common formats of in-game advertisements are virtual billboards in sports games and product placements (e.g. car brands in racing games (115)). While these classical forms of in-game advertisements might provide possibilities for interactive music, a more obvious field of application are video game soundtracks which contribute considerably to the gaming experience: background music is often dynamically adapted to the player's current situation in order to create a certain mood, for example if he gets involved in a fight. This can be used to integrate brand sounds and associate them with a certain mood. Besides background music playing for a rather long time, there are also often very short sounds which are directly connected to certain events (e.g. jumping or collecting items). These game events could also be used to generate interactive advertising music based on a player's actions in order to associate them with a given brand.

Out-of-Home Advertising: Besides applications for devices which are typically owned by the user, interactive music could also be used in the area of out-of-home advertising (e.g. on public displays). Especially in this area, the interaction paradigm has to be designed such that it requires no or only very few training. Furthermore, the system's location has to be chosen such that sound pollution is avoided: it should not be installed at a place where other people could be disturbed while being e.g. at work or shopping. We developed a prototype for a system where

users can interactively play music with hand movements in front of a public display and conducted a user study on how untrained users interact with it (see section 7.4.3).

This list of application areas can only cover a part of advertising opportunities for interactive music systems and may be completed with many others. There might be types of advertising or sites where interactive music or music at all do not make sense or are inappropriate. Many people have fun with musical applications, but they can also be annoying in some situations. It is much harder to ignore sounds than visual stimuli and hence, one has to take care of maintaining a moderate level of loudness and avoiding sound pollution, especially in out-of-home applications. Interactive music applications should not be misplaced and should only be employed where entertaining content is appropriate.

7.4.2 Approach: Interactive Advertising Jingles

In this section, we show how it is possible to generate music which is controlled by user interaction on the hand but which is still recognizable as a certain brand melody on the other hand. Our general approach for interactively generating music makes use of declarative preferences that express 'how the music should sound'. These preferences are expressed as soft constraints, a technique which is suited well for tackling concurrent problems. With our approach, it is possible to automatically derive preferences from existing melodies: this way, well-known melodies can be used in interactive applications and their characteristic properties can be preserved up to a certain extent. With this kind of preferences, it is possible to flexibly alter given melodies based on user interaction: the melodic material can be subject to dynamic changes while still remaining recognizable.

We make use of three basic kinds of preferences: First, we use preferences that are derived from user interaction, e.g. a touch display or a motion tracking system. These preferences reflect how the user wants the music to sound, for example ´I want to play fast notes with a high pitch´. The actual transformation of raw sensor readings to preferences depends strongly on the chosen user interface and interaction paradigm. The second type of preferences expresses general melodic rules: With this kind of preferences, it is possible to make the music consistent with a certain musical style. Furthermore, it is also possible to make the resulting melodies comply with a brand´s distinct acoustic identity, e.g. a certain advertising jingle. In most cases, the preferences derived from user interaction will be concurrent to an advertising jingle, i.e. the user interaction does not fit to the jingle with respect to both tonality and rhythmics. Since a certain amount of control over the music is assigned to the user, it is inherently not possible to exactly play a given melody note by note. Nevertheless, it is possible to generate melodies which are similar to it by using note pitches as well as tonal and rhythmic patterns appearing in the brand's distinct melody. This way, melodies can be generated considering both interactivity and brand recognition. At last, we use preferences that coordinate several instruments playing

simultaneously, for example a single player with static background music or multiple players among each other. This coordination can be made by preferring harmonic intervals between different instruments. Furthermore, it is also possible to coordinate multiple instruments such that they play similar rhythmic patterns.

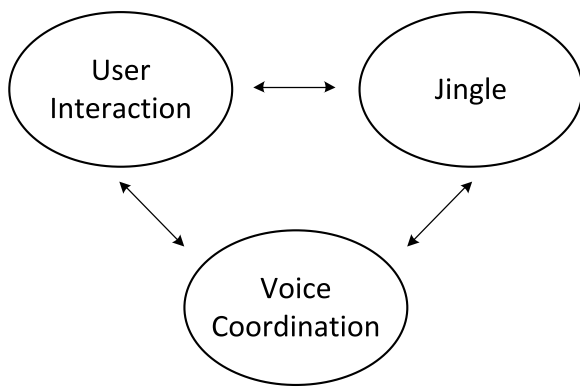


FIGURE 75 CONCURRENT PREFERENCES FOR INTERACTIVE ADVERTISING JINGLES

For developing applications which allow interactively playing with advertising jingles, the Fluxus sequencer can be used to train musical models as described in section 7.2. Based on these musical models, constraints can be generated which express 'how close a certain action is to a given advertising jingle'. This dynamic constraint is dependent on the current time and models the tonal as well as the rhythmic structure of the jingle:

$$jingleConstraint_{Voice,Time} : (Voice \rightarrow \ Action) \rightarrow \mathbb{R}$$

(constraint expressing similarity to an advertising jingle)

For each player, a $trackConstraint$ is defined as a combination of the jingle constraint and one or several other constraints reflecting user interaction, for example the $energyPitchConstraint$ (see section 5.2.):

$$trackConstraint_{v,t} = energyPitchConstraint_{v,t} * jingleConstraint_{v,t}$$

(constraint for a single player)

The final constraint problem is then a combination of all players' constraints and, for example, an additional harmony constraint:

$$interactiveAdvertisingJingles_t(val) = harmonyConstraint(val) + \sum_{v \epsilon Voice} trackConstraint_{v,t}(val)$$

(final constraint problem for interactive advertising jingles)

In order to achieve a good result, a lot of fine tuning is required: Just training the jingle's melody will not suffice in most cases because there will not be enough possibilities for varying the jingle based on user interaction. Consider for example the case that one wants to play a note when there is a pause in the jingle: one can use the concept of metric similarity as described in section 5.3, but the best musical results are obtained when there is explicit training data for all metric positions. Thus, the musical model should not only include the jingle itself but also several explicit variations of it. We achieved good results by training the jingle itself with a high training weight and several variations of it with a lower training weight. This way, the jingle itself will be the most dominant source for melodic material and the variations of it will be used if there is no other training data available. The challenge in training a jingle is to achieve a good balance between a high level of recognition on the one hand and a high level of control on the other hand. This depends of course also strongly on the given melody: a long advertising jingle will per se provide more tonal and rhythmic material than a short sound logo.

7.4.3 Study: Interactive Music on Public Displays

To investigate how people can interact with brand music in a public setting, we developed a prototype system in the area of digital signage. We wanted to find out how novice users with no previous training period or musical expertise would use such system and which gestures they would use. Our prototype is based on a large display where users are able to interactively play music with hand movements based on our approach for transforming spatial movements to music as described in section 7.3. At the time of the study, we have not integrated markerless tracking into our system yet. Instead, we used the marker-based vision framework Touchless SDK (116) for tracking hand movements with markers. We used a wall of luminous plasma displays (four 42" seamless displays arranged to a 16:9 screen of 1.85 meters width and 1.05 meters height with a resolution of 1706 to 960 pixels) and a high resolution camera attached above the screens.

We developed several variations of the system based on different interaction paradigms and conducted a user study (117). First of all, we wanted to figure out if users understand that they have control over the music, and second, in which way people would interact with the display. In general, interactive applications on public displays can be controlled for example with finger touch, hand gestures or body movements in a simple and unobtrusive way. Based on informal observations of colleagues, we already knew that hand movements are understood and accepted quite quickly if there is additional visual feedback. We developed several paradigms for interacting with the system based on hand movements and compared three paradigms that had shown promise in pretests. All paradigms are based on two continuous parameters: 'pitch' controls the note height (high or low); 'energy' the rate of played notes (slow or fast). There is always an additional visualization of the hand movements as well as the resulting notes (as described in section 7.3.2). The first paradigm uses only one hand at a time: The pitch of the music is controlled by the hand's vertical position (up or down); the rate of played notes is controlled by the velocity of the movements. The second paradigm allows the user to control the music with both hands: for computing the note pitch and the rate of played notes, the mean value of both hands' vertical position and velocity is taken. The third interaction paradigm extends the second one by allowing the user to control both parameters with separate hands, i.e. one hand controls the note pitch and the other hand controls the rate. To assess these variations, we conducted the following user study at our lab over the course of three days: we prepared a room which contained the system and installed 4 cameras around it in order to be able to observe the users' behavior from different angles. We recorded all cameras to a synchronized and time-stamped video file. In total, 21 people participated in the study (12 males, 9 females). The average age was 30 years; participants were students, employees, technicians, web designers, marketing managers, assistants and housewives. We started with an initial briefing where we explained the setting of the study: participants were told that there was a room containing a public display, but we did not tell if or how they could interact with the display. To be able to track hand movements with our marker-based system, we asked them to put on colored gloves (after the study we surveyed if participants felt constricted by the gloves in any way). After the initial briefing we guided the participants to the room where our sample music content with one of the three described interaction paradigms was running. We surveyed the people with the cameras and did not interrupt them. When they came out, we conducted a semi-structured interview with them. For all 21 subjects synchronized and time-stamped videos were recorded and predefined user behavior regarding hand movements and gestures was transcribed.

Even without any previous instructions, most users were aware that they have control over the music. Only 2 out of 21 people did not recognize the connection between their hand movements and the music they heard. No user stopped interacting while standing in front of the system for longer periods. The average user made hand gestures for over 90% of the time which gives us confidence that people understood the basic interaction paradigm. Based on the videos, we analyzed how long it took until people interacted in the way we intended, i.e. when they

started to primarily make hand gestures which are relevant for the music generation. The variant based on only one hand took 132 seconds on average, the variant based on the mean value of both hands took 118 seconds and the third variant (separate hands for both parameters) took 92 seconds. Most users seemed to interact in a rather intuitive way but interviews revealed that not everybody did consciously identify the variable parameters ('pitch' and 'energy', i.e. 'rate of played notes') and how they can be controlled: 12 out of 21 people stated that they used up-and-down movements to control the music and only 10 out of 21 people could tell how note pitches can be controlled; only 2 users understood how they can vary the rate of notes. Nevertheless, the results of this study make us confident that public displays are an appropriate platform for advertising based on interactive music and that hand gestures can be used for interacting with music without any previous training. We are confident that additional visual clues or context-sensitive help instructions can greatly help in understanding how the system works: we observed that users were able to understand the system very fast if we gave them only few initial instructions.

7.4.4 Related Work

In the recent time, both interactive advertisements and interactive music systems have become increasingly popular. There exist advertisements including interactive music, but to our knowledge there are no approaches which enable customers not only to play with, but also manipulate and shape well-known brand melodies by means of interactive control mechanisms.

Advertisements make use of sound in various form and function. Today, sound is employed in traditional media like television or radio as well as in new media like the Internet. Regardless of the medium, it is possible to distinguish between several types of advertising sounds: A *sound logo* is a short, distinctive melody or tone sequence with a length from one to three seconds. It can be seen as the acoustic equivalent to a visual logo and, in the ideal case, establishes a symbiosis with it. It is mostly played along with the visual logo at the beginning or ending of a commercial (114). Well-known examples for sound logos are the Intel 'Bong' or the famous sequence of five tones from Deutsche Telekom. *Advertising Jingles* are short songs that are often played along with lyrics to convey an advertising slogan. They distinguish themselves from sound logos by not only transporting associations but also functioning as a mnemonic for the slogan. Thus, they often follow known and memorable folk songs in melody, rhythm and tone and integrate other brand elements like the brand name (114). A well-known example is the Haribo jingle which has been translated to many different languages. *Background Music* is mostly purely instrumental. Its purpose is to create a certain atmosphere, thus functioning just as a supplement to other stimuli such as narrated text or images. An example is accordion music, eliciting convenient associations to an advertisement for French wine (114) (118). In the area of product design, so-called *sound objects* are connected to activities like closing a car door (119). In the area of interactive media, acoustic signals connected to certain events like a mouse

click on a graphical element are also referred to as sound objects. An example for the employment of sound objects in advertising is a banner ad from Apple where one can choose different colors for the iPhone and a sound appears when the user moves the mouse cursor over one of the items. In literature, many more types and subtypes of sound branding are described, such as *Brand Songs*, *Soundscapes*, *Sound Icons*, *Brand Voices* or *Corporate Songs* (114). Most of the sound branding elements are used in different application areas and have been transferred to various platforms, e.g. you can find a sound logo, jingle or sound object today on television, the internet, mobile devices or even when unpacking a real or virtual product. As diverse are proposals for possible future applications of sound branding, including areas such as multi-sensory communication, the use of interactive sounds in packaging or at the point of sale (113).

In the last years, a trend to interactive advertisements can be identified, for example casual games in the web (for example within a banner ad) or interactive wall or floor displays in the out-of-home domain. Besides regular games where one has to achieve a certain goal, there are also interactive plays (120), i.e. invitations to less structured activities that imply creative or participatory elements. Such advertisements often allow manipulating visual objects that are constituent parts of the brand identity, like a brand logo that can be moved along the display surface by hands or feet. Acoustic events mostly play only a secondary role or do not appear at all in these interactive advertisements: often, they are delimited to sound objects supplementing the visual interaction or statically playing background music.

The functions of sound in advertising are manifold: The acoustic sensory channel is hard to ignore (compared e.g. to the visual channel) and thus, sounds are often used to gain or hold attention (121) (122). Sound is also used to influence the mood of consumers, to structure the time of an ad or to persuade consumers by using rhetorical elements like rhythm, repetition, narration, identification or location (123). Sound can increase the reception and memorization of information and can enhance the overall user experience (114).

Advertising sounds are a powerful tool, but they are also subject to specific requirements. For example, according to John Groves, the characteristics of an effective sound logo include memorability, distinctiveness, flexibility, conciseness and brand fit (113) (114): *Memorability* is the most important quality of a good sound logo. It strongly depends on the sound designer´s ability to create a 'mini hit' or catchy tune. Memorability includes the recognition and recall of the sound logo, where the latter is more difficult to achieve. Memorable elements are often used to quickly evoke associations. A good sound logo has to be *distinctive*; otherwise it may not be recognized or confused with a competitor. For this reason, an unmistakable sound characteristic has to be found. This is usually done by analyzing the market sector and how competitors deal with music and sound. For a sound logo, two kinds of *flexibility* are advantageous: musical and technical flexibility. Musical flexibility means that a melody can be combined with different contexts and emotional situations of different advertisement, e.g. by

recording it with different instrumentations or styles. Technical flexibility means that the sound can be perceived cross-platform (e.g. via phone, computer speaker or television) without impairments. Good sound logos have to be *concise*, i.e. short and with tone combinations usually only lasting seconds since they are often combined with visual logos which also appear only for a short time. A typical sound logo has a length of about two seconds. *Brand fit* means that a sound logo should reflect the brand's values and communicate its attributes on the acoustic level. It can be very adverse if the sound identity contradicts or obviously overstates the brand´s attributes.

7.4.5 Conclusion

We investigated how interactive music systems can be used in the area of pervasive advertising, described requirements of interactive advertising music and outlined possible application areas and limitations. Based on our framework for composing music with soft constraints, we introduced a general approach for developing applications where advertising music is controlled by user interaction such that it complies with the requirements of both interactivity and brand recognition. The results of our first study make us confident that it is possible to intuitively control music with hand movements in front of a public display without any previous training period.

We presented interactive advertising jingles controlled by a touch display at the 'Komma 2011' trade fair for communication and marketing in Munich and were in contact with several companies which showed interest in employing the system in a real advertising campaign. In the meantime, we also developed a system based on markerless motion tracking and would like to see how people interact with it in a real public situation. Besides this, we are also interested in developing and assessing new user interfaces and interaction paradigms.

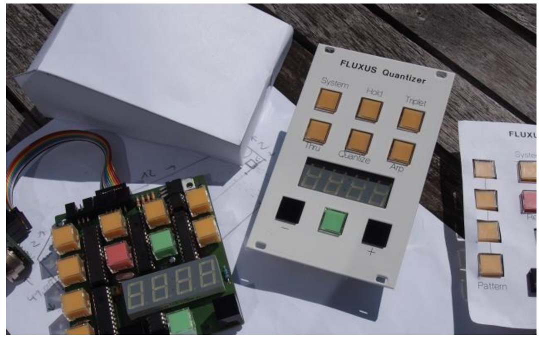
Sound is a powerful tool which has to be employed with care: many people have fun with musical applications, but they can also be annoying in some situations. When employed appropriately, we think that interactive advertising music can be a quite attractive way of communicating a certain brand image.

8 Conclusion and further work

In this work, we extended known approaches for composing music based on classical constraints with soft constraints à la Bistarelli et al. that provide a new level of expressiveness and allow specifying additional types of musical rules. We introduced an algebraic framework for interactively composing music in real-time with soft constraints and implemented it along with several applications based on a tangible interface, touch displays and a motion tracking system. Furthermore, we investigated how interactive music can be employed in the area of pervasive advertising. In our systems, we successfully used soft rules that would not be expressible in a classical constraint framework, e.g. optimization of harmonic intervals or maximizing similarity to a musical transition model. Our framework allows specifying interactive music systems by declaratively defining preferences and is general enough to model common techniques like classical constraint-based rules, optimization goals or machine learning approaches. Different aspects of the music generation can be constrained in order to e.g. coordinate multiple voices or generate music fitting to a certain style. Besides static preferences, it is also possible to define dynamically changing preferences. To achieve the latter, we extended the theory of monoidal soft constraints with an approach for defining preferences which change over time.

We introduced our framework and demonstrated applications of it in the scientific area at conferences, talks and workshops. We also made public exhibitions at the 'night of science' in Munich (Nacht der Wissenschaften / Senses09), 'Ferchau Art of Engineering 2010', the 'Komma' trade fair for communication and marketing in Munich (2011) or the concert series 'Zukunft(s)musik' in Augsburg (2011). The overall feedback was very positive: many people got attracted by our interactive music applications, wanted to know how they work and enjoyed playing with them. To our taste, the interactively generated music sounds quite good for being algorithmically composed. Especially in applications which make use of previously trained melodic rules, the resulting melodies' rhythmic and tonal sequences sound quite 'natural'. There is of course a significant difference to music which was composed by a human: the melodies are not as elaborate as in a carefully composed piece and an overall structure and development is missing. This could be improved with additional rules or by varying several styles. With our approach, we 'correct' the actions of a user with respect to both tonality and rhythm: a rough declaration of the desired tonal height is transformed to a concrete note pitch such that it also complies with musical rules. When it comes to correcting rhythm, there is an inherent problem: when a user's action is detected shortly before a regular rhythmic position (e.g. a 16th note), it can be corrected by delaying it a bit (which is barely inaudible). However, when the user's action is detected shortly after a regular position, it has to be delayed for a rather long time to correct it – which is clearly audible and leads to strange rhythms and undesired syncopations (i.e. accenting of unusual rhythmic positions). We consider this as one of the main musical deficiencies of our system – especially in combination with high-latency user interfaces. In order to improve this, we want to experiment with rules which constrain certain syncopations. While being very interesting in the beginning, we also made the observation that many people do not

have long term motivation to play. This is fine for musical applications in the area of casual gaming, but we also envision applications which make users want to play longer musical performances. One possible reason for this could be the missing challenge in playing: the user has much control over the music, but our system always ensures a rather good result. It would be interesting to investigate how our system can be made more 'difficult to play' and if this has an effect on long term motivation.



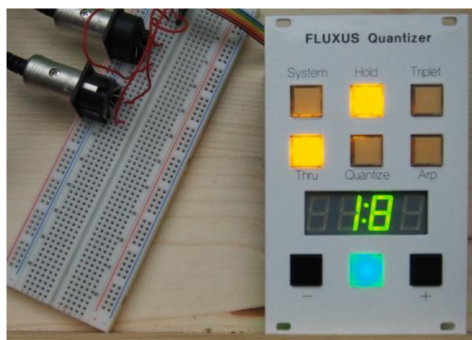


FIGURE 76 FLUXUS MODULAR PROTOTYPES

We want to continue our work in two directions: interactive algorithmic tools for (electronic) musicians and musical instruments for people with strong physical disabilities. At the moment, we are working on a modular version of the 'Fluxus' sequencer based on the Arduino platform for embedded devices. This modular system will be a simpler version of the desktop application: it will for example not be possible to coordinate multiple players among each other due to the distributed modular architecture. We also do not use a generic constraint solver in this system because of limited computing power and memory. Instead, we directly program all rules and optimization goals. In the past, we made initial investigations how interactive music systems could be used for music therapy (13) (43). We think there are several application areas where computer-assisted musical instruments could be adequate, for example for people with strong physical disabilities that can use an accessible instrument controlled e.g. by an eye tracker. We want to iteratively design and evaluate such musical instrument together with a music therapist and a group of disabled people that are willing to learn an instrument. At the moment, we are in contact with a music therapist who wants to evaluate the usage of electronic systems in general and would like to participate in the development process.

Besides our scientific contribution of extending known constraint-based approaches for generating music with soft and concurrent rules, we also have personal proof of concept that soft constraints can be used in interactive real-time systems in general with high reliability and maintainability. This could be interesting for any application area where concurrent processes have to be optimized in real-time while running.

BIBLIOGRAPHY

1. **Aage Gerhardt Drachmann.** The mechanical technology of Greek and Roman antiquity, Page 197. *Munksgaard, Copenhagen.* 1963.

2. **Teun Koetsier.** On the prehistory of programmable machines: musical automata, looms, calculators. *Mechanism and Machine Theory, Volume 36, Issue 5, Pages 589-603.* 2001.

3. **Henry George Farmer.** The Organ From Arabic Sources (Hydraulic Organs). *The Organ of the Ancients, From Eastern Sources (Hebrew, Syriac and Arabic), Pages 79-118, W. Reeves, London.* 1931.

4. **John Cohen.** Human robots in myth and science. *A. S. Barnes and Co., New York.* 1966.

5. **Roger Smith.** Game Impact Theory: The Five Forces That Are Driving the Adoption of Game Technologies within Multiple Established Industries. *Interservice/Industry Training, Simulation and Education Conference.* 2009.

6. **David Plans, Davide Morelli.** Experience-Driven Procedural Music Generation for Games. *IEEE Transactions on Computational Intelligence and AI, Volume 4, Issue 3, Pages 192-198.* 2012.

7. **Luc Steels.** Reasoning Modeled as a Society of Communicating Experts. *MIT AI Lab Technical Report.* 1979.

8. **Stefano Bistarelli, Ugo Montanari, Francesca Rossi.** Semiring-based constraint satisfaction and optimization. *Journal of the ACM, Volume 44, Issue 2, Pages 201–236.* 1997.

9. **Rina Dechter.** Constraint Processing. *Morgan Kaufman, Burlington.* 2003.

10. **Roman Barták.** Modelling Soft Constraints: A Survey. *Neural Network World, Volume 12, Pages 421-431.* 2002.

11. **Stefano Bistarelli.** Semirings for Soft Constraint Solving and Programming. *Springer, New York.* 2004.

12. **Matthias Hölzl, Max Meier, Martin Wirsing.** Which Soft Constraints do you Prefer? *Electronic Notes in Theoretical Computer Science, Volume 238, Pages 189 - 205.* 2009.

13. **Max Meier.** Life is Soft - Modeling and Solving Problems with Soft Constraints. *Master Thesis LMU München.* 2008.

14. **Leslie Lamport.** Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM, Volume 21, Issue 7, Pages 558-565.* 1978.

15. **Stefano Bistarelli, Rosella Gennari, Francesca Rossi.** Constraint Propagation for Soft Constraints: Generalization and Termination Conditions. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Volume 1894, Pages 83-97.* 2000.

16. **Thomas Schiex.** Arc consistency for soft constraints. *Artificial Intelligence, Volume 154, Pages 411-424.* 2004.

17. **Stefano Bistarelli, Philippe Codognet, Kin-Chuen Hui, Jimmy Ho Man Lee.** Solving Finite Domain Constraint Hierarchies by Local Consistency and Tree Search. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Volume 2833, Pages 138-152.* 2003.

18. **Louise Leenen.** Solving semiring constraint satisfaction problems. *Dissertation University of Wollongong.* 2010.

19. **Louise Leenen, Thomas Meyer, Aditya Ghose.** Relaxations of semiring constraint satisfaction problems. *Information Processing Letters, Volume 103, Pages 177 - 182.* 2007.

20. **Louise Leenen, Aditya Ghose.** Branch and Bound Algorithms to Solve Semiring Constraint Satisfaction Problems. *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence, Pages 991 - 997.* 2008.

21. **Fabio Gadducci, Matthias Hölzl, Giacoma Valentina Monreale, Martin Wirsing.** Soft Constraints for Lexicographic Orders. *Proceedings of the 12th Mexican International Conference on Artificial Intelligence, Part I, Pages 68-79.* 2013.

22. **Stefano Bistarelli, Spencer K.L. Fung, J.H.M. Lee, Ho-fung Leung.** A Local Search Framework for Semiring-Based Constraint Satisfaction Problems. *In proceedings of the 5th international workshop on Soft Constraints.* 2003.

23. **Philippe Codogneta, Daniel Diaz.** Compiling constraints in clp(FD). *The Journal of Logic Programming, Volume 27, Issue 3, Pages 185-226.* 1996.

24. **Yan Georget, Philippe Codognet.** Compiling Semiring-based Constraints with clp(FD,S). *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Volume 152, Pages 205-219.* 1998.

25. **Hana Rudová.** Soft CLP(FD). *16th international Florida Artificial Intelligence Research Society conference, Pages 202 - 207.* 2003.

26. **Alberto Delgado, Carlos Alberto Olarte, Jorge Andrés Pérez, Camilo Rueda.** Implementing Semiring-Based Constraints using Mozart. *Lecture Notes in Computer Science, Volume 3389, Pages 224-236.* 2005.

27. **Stefano Bistarelli, Thom Frühwirth, Michael Marte, Francesca Rossi.** Soft Constraint Propagation and Solving in Constraint Handling Rules. *Computational Intelligence, Volume 20, Issue 2, Pages 287-307.* 2004.

28. **Martin Wirsing, Grit Denker, Carolyn Talcott, Andy Poggio, Linda Briesemeister.** A rewriting logic framework for soft constraints. *Electronic Notes in Theoretical Computer Science, Volume 176, Issue 4, Pages 181-197.* 2007.

29. **Thomas D. Rossing.** Springer Handbook of Acoustics. *Springer, New York.* 2007.

30. **Werner Mohrlok, Herwig Mohrlok.** Method of and control system for automatically correcting a pitch of a musical instrument. *United States Patent 5442129.* 1995.

31. **Ross W. Duffin.** How Equal Temperament Ruined Harmony (and Why You Should Care). *W. W. Norton & Company, New York.* 2006.

32. **Godfried T. Toussaint.** The Euclidean algorithm generates traditional musical rhythms. *BRIDGES: Mathematical Connections in Art, Music, and Science, Pages 47 - 56.* 2005.

33. **Alex Ross.** The Rest is Noise. *Farrar, Straus and Giroux, New York.* 2007.

34. **Gerhard Nierhaus.** Algorithmic Composition: Paradigms of Automated Music Generation. *Springer, New York.* 2009.

35. **Nick Collins.** Live Coding of Consequence. *Leonardo Music Journal, Volume 44, Number 3, Pages 207-211.* 2011.

36. **Samuel Aaron, Alan F. Blackwell, Richard Hoadley, Tim Regan.** A principled approach to developing new languages for live coding. *10th conference on New Interfaces For Musical Expression, Pages 381 - 386.* 2011.

37. **Roger B. Dannenberg.** Live Coding Using a Visual Pattern Composition Language. *12th Biennial Symposium on Arts and Technology.* 2011.

38. **Louis Bigo, Jean-Louis Giavitto, Antoine Spicher.** Building Topological Spaces for Musical Objects. *Mathematics and Computation in Music, Lecture Notes in Computer Science, Volume 6726, Pages 13-28.* 2011.

39. **Jesús L. Alvaro, Beatriz Barros.** Computer Music Cloud. *Exploring Music Contents, Lecture Notes in Computer Science, Volume 6684, Pages 163-175.* 2011.

40. **Jesús L. Alvaro , Beatriz Barros.** A new cloud computing architecture for music composition. *Journal of Network and Computer Applications, Volume 36, Issue 1, Pages 429-443.* 2013.

41. **Chih-Fang Huang, Hsiang-Pin Lu, Jenny Ren.** Algorithmic approach to sonification of classical Chinese poetry. *Multimedia Tools and Applications, Volume 61, Issue 2, Pages 489-518.* 2011.

42. **Max Meier (RechNet GmbH).** Musiksystem. *German Patent DE102009017204A1.* 2009.

43. **Matthias Hölzl, Grit Denker, Max Meier, Martin Wirsing.** Constraint-Muse: A Soft-Constraint Based System for Music Therapy. *3rd conference on Algebra and Coalgebra in Computer Science, Pages 423 - 432.* 2009.

44. **Howard Hanson.** Harmonic Materials of Modern Music - Resources of the Tempered Scale. *Appleton-Century-Crofts, New York.* 1960.

45. **François Pachet, Pierre Roy.** Musical Harmonization with Constraints: A Survey. *Constraints, Volume 6, Issue 1, Pages 7-19.* 2001.

46. **Johann Joseph Fux.** The Study of Counterpoint from Johann Joseph Fux's Gradus ad Parnassum. *Translated and edited by Alfred Mann, W.W. Norton & Company, New York.* 1965.

47. **Arnold Schoenberg.** Harmonielehre. *Universal Edition, Wien.* 1922.

48. **Bill Schottstaedt.** Automatic species counterpoint. *Technical Report STAN-M-19, Stanford University CCRMA.* 1984.

49. **Kemal Ebcioğlu.** An Expert-System for Harmonizing Four-Part Chorales. *Computer Music Journal, Volume 12, Number 3, Pages 43-59.* 1988.

50. **C. P. Tsang, M. Aitken.** Harmonizing music as a discipline of constraint logic programming. *Proceedings of the International Computer Music Conference, Pages 61 - 64.* 1991.

51. **François Pachet, Pierre Roy (Sony France).** Problem solving using an improved constraint satisfaction problem formulation. *European patent EP0961262 A1.* 1999.

52. **François Pachet.** Constraints and Multimedia. *Practical Applications of Constraint Logic Programming, Pages 3-13.* 1999.

53. **François Pachet, Olivier Delerue, Peter Hanappe.** MusicSpace goes Audio. *Sound in Space Symposium.* 2000.

54. **François Pachet, Pierre Roy.** Automatic Generation of Music Programs. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, Volume 1713, Pages 331-345.* 1999.

55. **Pierre Roy.** Satisfaction de contraintes et programmation par objets. *Dissertation University of Paris 6.* 1998.

56. **Torsten Anders, Eduardo R. Miranda.** Constraint programming systems for modeling music theories and composition. *ACM Computing Surveys, Volume 43, Issue 4, Article 30.* 2011.

57. **Mikael Laurson.** PATCHWORK: A Visual Programming Language and some Musical Applications. *Dissertation Sibelius Academy Helsinki.* 1996.

58. **Gerard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, Olivier Delerue.** Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal, Volume 23, Number 3, Pages 59-72.* 1999.

59. **Torsten Anders.** Arno: Constraints Programming in Common Music. *Proceedings of the International Computer Music Conference, Pages 324 - 327.* 2000.

60. **Heinrich Taube.** Common Music: A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal, Volume 15, Number 2, Pages 21-32.* 1991.

61. **Torsten Anders.** Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System. *Dissertation Queen's University Belfast.* 2007.

62. **Martin Henz, Stefan Lauer, Detlev Zimmermann.** COMPOzE - Intention-based Music Composition through Constraint Programming. *Proceedings of the 8th international conference on Tools with Artificial Intelligence, Pages 118 - 121.* 1996.

63. **Torsten Anders, Eduardo R. Miranda.** Constraint-Based Composition in RealTime. *Proceedings of the International Computer Music Conference.* 2008.

64. **Donald P. Pazel, Steven Abrams, Robert M. Fuhrer, Daniel V. Oppenheim, James Wright.** A Distributed Interactive Music Application using Harmonic Constraint. *Proceedings of the International Computer Music Conference.* 2000.

65. **François Pachet, Pierre Roy, Gabriele Barbieri.** Method for creating a Markov process that generates sequences. *Unites States Patent 20120246209.* 2012.

66. **Gabriele Barbieri, François Pachet, Pierre Roy, Mirko Degli Esposti.** Markov Constraints for Generating Lyrics with Style. *Frontiers in Artificial Intelligence and Applications, Volume 242, Pages 115 - 120.* 2012.

67. **Charlotte Truchet, Gerard Assayag, Philippe Codognet.** Visual and Adaptive Constraint Programming in Music. *Proceedings of the International Computer Music Conference.* 2001.

68. **Richard Pinkerton.** Information theory and melody. *Scientific American, Volume 194, Number 2, Pages 77-86.* 1956.

69. **Mary Farbood, Bernd Schoner.** Analysis and Synthesis of Palestrina-Style Counterpoint. *Proceedings of the International Computer Music Conference, Pages 471 - 474.* 2001.

70. **Wei Chai, Barry Vercoe.** Folk Music Classification Using Hidden Markov Models. *Proceedings of the International Conference on Artificial Intelligence.* 2001.

71. **Gérard Assayag, Shlomo Dubnov, Olivier Delerue.** Guessing the Composer's Mind: Applying Universal Prediction to Musical Style. *Proceedings of the International Computer Music Conference, Pages 496 - 499.* 1999.

72. **Darrell Conklin.** Music Generation from Statistical Models. *Proceedings of the AISB symposium on artificial intelligence and creativity in the arts and sciences, Pages 30 - 35.* 2003.

73. **Jan Beran.** Statistics in Musicology. *Chapman & Hall/CRC, Boca Raton.* 2004.

74. **Pedro P. Cruz-Alcázar, Enrique Vidal-Ruiz.** A study of Grammatical Inference Algorithms. *ICML workshop on automata induction, grammatical inference and language acquisition.* 1997.

75. **Olivier Lartillot, Shlomo Dubnov, Gill Assayag, Gérard Bejerano.** Automatic Modeling of Musical Style. *Proceedings of the International Computer Music Conference.* 2001.

76. **John A. Biles.** GenJam: A Genetic Algorithm for Generating Jazz Solos. *Proceedings of the International Computer Music Conference, Pages 131 - 137.* 1994.

77. **John A. Biles .** Autonomous GenJam: Eliminating the Fitness Bottleneck by Eliminating Fitness. *Workshop on Non-routine Design with Evolutionary Systems, Genetic and Evolutionary Computation Conference.* 2001.

78. **Martin Dostál.** Evolutionary Music Composition. *Handbook of Optimization, Intelligent Systems Reference Library, Volume 38, Pages 935 - 964.* 2013.

79. **Cycling '74.** Max/MSP. *Software published under proprietary license, http://cycling74.com (last accessed: 2014-08-26).*

80. **Miller Puckette (original author).** Pure Data. *Open Source Software released under modified BSD license, http://puredata.info (last accessed: 2014-08-26).*

81. **James McCartney (original author).** SuperCollider. *Open Source Software released under GPL license, http://supercollider.sourceforge.net (last accessed: 2014-08-26).*

82. **Native Instruments.** Reaktor. *Proprietary Software, http://www.native-instruments.com (last accessed: 2014-08-26).*

83. **Clavia.** Nord Modular. *Modular analog modeling synthesizer, http://www.nordkeyboards.com (last accessed: 2014-08-26).*

84. **Arturia.** Origin. *Modular analog modeling synthesizer, http://www.arturia.com (last accessed: 2014-08-26).*

85. **Barry Vercoe (original author).** CSound. *Open Source Software published under the LGPL license, http://www.csounds.com (last accessed: 2014-08-26).*

86. **Ge Wang.** The ChucK Audio Programming Language. *Dissertation Princeton University.* 2008.

87. **Andrew Sorensen.** Impromptu. *Open Source Software, http://impromptu.moso.com.au (last accessed: 2014-08-26).*

88. **François Pachet.** The Continuator: Musical Interaction With Style. *Journal of New Music Research, Volume 32, Issue 3, Pages 333 - 341.* 2003.

89. **François Pachet.** Playing with virtual musicians: the Continuator in practice. *IEEE MultiMedia, Volume 9, Number 3, Pages 77-82.* 2002.

90. **Anna Rita Addessi, Laura Ferrari, François Pachet.** Without touch, without seeing. Children playing with the Continuator, a virtual musician. *Proceedings of the International Seminar of Early Childhood Music Education.* 2006.

91. **Leslie Sanford.** C# MIDI Toolkit. *Software library under the MIT license.*

92. **Max Meier, Max Schranner.** The Planets. *Proceedings of the 10th conference on New Interfaces For Musical Expression, Pages 501 - 504.* 2010.

93. Microsoft XNA Game Engine. *Software released as Freeware, http://create.msdn.com (last accessed: 2014-08-26).*

94. Mercury Particle Engine. *Open Source Software published under the Microsoft Public License, http://mpe.codeplex.com (last accessed: 2014-08-26).*

95. Farseer Physics Engine. *Open Source Software published under the Microsoft Permissive License, http://farseerphysics.codeplex.com (last accessed: 2014-08-26).*

96. **Sergi Jordà, Günter Geiger, Marcos Alonso, Martin Kaltenbrunner.** The reacTable: Exploring the synergy between live music performance and tabletop tangible interfaces. *Proceedings of the 1st conference on Tangible and Embedded Interaction, Pages 139 - 146.* 2007.

97. **James Patten, Ben Recht, Hiroshi Ishii.** Audiopad: a tag-based interface for musical performance. *Proceedings of the 2nd conference on New Interfaces for Musical Expression, Pages 11 - 16.* 2002.

98. **Markus Bischof, Bettina Conradi, Peter Lachenmaier, Kai Linde, Max Meier, Philipp Pötzl, Elisabeth André.** XENAKIS - Combining Tangible Interaction with Probability-Based Musical Composition. *Proceedings of the 2nd conference on Tangible and Embedded Interaction, Pages 121 - 124.* 2008.

99. **Toshio Iwai, Indies Zero, Nintendo.** Electroplankton . *Music Game for Nintendo DS.* 2005.

100. **Yu Nishibori, Toshio Iwai.** Tenori-On. *Proceedings of the 6th conference on New Interfaces for Musical Expression, Pages 172 - 175.* 2006.

101. **Ginevra Castellano, Roberto Bresin, Antonio Camurri, Gualtiero Volpe.** Expressive control of music and visual media by full-body movement. *Proceedings of the 7th conference on New Interfaces for Musical Expression, Pages 390 - 391.* 2007.

102. **Rung-huei Liang, Ming Ouhyoung.** Impromptu conductor: a virtual reality system for music generation based on supervised learning. *Displays Volume 15, Issue 3, Pages 141–147, Elsevier.* 1994.

103. **Horace Ho-Shing Ip, Ken Chee Keung Law, Belton Kwong.** Cyber Composer: Hand Gesture-Driven Intelligent Music Composition and Generation. *Proceedings of the 11th Multimedia Modelling Conference, Pages 46 - 52.* 2005.

104. **Yoshiki Nishitani, Kenji Ishida, Eiko Kobayashi (Yamaha Corporation).** System of processing music performance for personalized management of and evaluation of sampled data. *United States Patent 7297857 B2.* 2007.

105. **Peter Wiest.** Verfahren zur Akustisierung körpereigener physikalischer Werte und Vorrichtung zur Durchführung des Verfahrens. *German Patent 19522958 C2.* 1999.

106. **Farah Jubran.** Sound generating device for use by people with disabilities. *United States Patent 2007/0241918 A1.* 2007.

107. **Joel Chadabe.** Interactive music composition and performance system. *United States Patent 4526078.* 1985.

108. **James A. Wheaton, Erling Wold, Andrew J. Sutter (Yamaha Corporation).** Position-based controller for electronic musical instrument. *United States Patent 5541358.* 1996.

109. **Gilbert Beyer, Max Meier.** Interactive Advertising Jingles: Using Music Generation for Sound Branding. *Proceedings of the 3rd Workshop on Pervasive Advertising and Shopping.* 2010.

110. **Max Meier, Gilbert Beyer.** Interacting with Sound. *Pervasive Advertising, Pages 325-342, Springer, New York.* 2011.

111. **Gilbert Beyer, Max Meier.** Music Interfaces for Novice Users: Composing Music on a Public Display with Hand Gestures. *Proceedings of the 11th conference on New Interfaces For Musical Expression, Pages 507 - 510.* 2011.

112. **Gilbert Beyer, Max Meier.** Performing Music in Public Settings using a Constraint-based System. *Proceedings of the Workshop on Performative Interaction in Public Space at CHI.* 2011.

113. **John Groves.** A Short History of Sound Branding. *Audio Branding – Entwicklung, Anwendung, Wirkung akustischer Identitäten in Werbung, Medien und Gesellschaft, Nomos Verlagsgesellschaft, Baden-Baden.* 2009.

114. **Paul Steiner.** Sound Branding – Grundlagen der akustischen Markenführung. *Gabler, Wiesbaden.* 2009.

115. **Micheuf R. Nelson.** Recall of Brand Placements in Computer/Video Games. *Journal of Advertising Research, Volume 42, Issue 2, Pages 80-92.* 2002.

116. Touchless SDK. *Open Source Software released under the Microsoft Public License, http://touchless.codeplex.com (last accessed: 2014-08-26).*

117. **Steffen Ritter.** Evaluation des Prototyps für interaktive Werbejingles hinsichtlich der User-Interaktion. *Bachelor Thesis LMU München.* 2010.

118. **Frank A. Diederichs, Christian Stonat.** Musik und Werbung. Marketing mit Emotionen. *Handbuch der Musikwirtschaft, Pages 409-422, Josef Keller Verlag, Starnberg.* 2003.

119. **Karsten Kilian.** Von der Markenidentität zum Markenklang als Markenelement. *Audio Branding, Pages 54-69, Reinhard Fischer, Munich.* 2009.

120. **Natascha Adamowsky.** Homo Ludens – whale enterprise: Zur Verbindung von Spiel, Technik und den Künsten. *Homo faber ludens: Geschichten zu Wechselbeziehungen von Technik und Spiel, Pages 57-82, Peter Lang Verlagsgruppe, Bern.* 2003.

121. **James J. Kellaris, Anthony D. Cox, Dena Cox.** The Effect of Background Music on Ad Processing: A Contingency Explanation. *Journal of Marketing, Volume 57, Number 4, Pages 114-125.* 1993.

122. **Werner Kroeber-Riel, Franz-Rudolf Esch.** Strategie und Technik der Werbung: Verhaltenswissenschaftliche Ansätze. *Kohlhammer, Stuttgart.* 2004.

123. **Linda M. Scott.** Understanding Jingles and Needledrop: A Rhetorical Approach to Music in Advertising. *Journal of Consumer Research, Volume 17, Number 2, Pages 223-236.* 1990.

PICTURE CREDITS

Section 2 includes pictures under Creative Commons licenses from André Karwath, Rainer Zenz, Nathalie Donne, Joaquim Alves Gaspar and Albin Schmalfuß. Pictures of the wine glass, beer glass and water glass with kind permission of WMF AG.

Figure 11: Created with Adobe Audition

Figure 12: From the book ‘Tidens naturlære’ (1903) by Poul la Cour

Figure 17: Portrait of Johann Sebastian Bach by Elias Gottlob Haussmann (1748); Self-portrait by Arnold Schoenberg (1910, Arnold Schoenberg center Vienna)

Figure 18: With kind permission of the John Cage Trust

Figure 19: Original score by Iannis Xenakis; Picture of the Philips pavilion from ‘Space Calculated in Seconds’ by Marc Treib and Richard Felciano (Princeton University Press, 1996)

Figure 20: ‘Musik der Zeit – 60er‘ - Westdeutscher Rundfunk Köln (WDR), http://www.wdr.de (last accessed: 2014-08-26)

Figure 21: Buchla & Associates website, http://www.buchla.com (last accessed: 2014-08-26)

Figure 29: Cycling74 website, http://cycling74.com (last accessed: 2014-08-26)