Constraint Programming Systems for Modeling Music Theories and Composition

TORSTEN ANDERS and EDUARDO R. MIRANDA, University of Plymouth

Constraint programming is well suited for the computational modeling of music theories and composition: its declarative and modular approach shares similarities with the way music theory is traditionally expressed, namely by a set of rules which describe the intended result. Various music theory disciplines have been modeled, including counterpoint, harmony, rhythm, form, and instrumentation. Because modeling music theories "from scratch" is a complex task, generic music constraint programming systems have been proposed that predefine the required building blocks for modeling a range of music theories. After introducing the field and its problems in general, this survey compares these generic systems according to a number of criteria such as the range of music theories these systems support.

Categories and Subject Descriptors: J.5 [**Computer Applications**]: Arts and Humanities—*Music*; D.2.6 [**Software Engineering**]: Programming Environments

General Terms: Design, Theory

Additional Key Words and Phrases: Algorithmic composition, computer-aided composition, constraint programming, music representation, music theory

ACM Reference Format:
Anders, T. and Miranda, E. R. 2011. Constraint programming systems for modeling music theories and composition. ACM Comput. Surv. 43, 4, Article 30 (October 2011), 38 pages.
DOI = 10.1145/1978802.1978809 http://doi.acm.org/10.1145/1978802.1978809

1. INTRODUCTION

Musicians and computer scientists alike have been fascinated by the modeling of music composition with computer programs for decades. The *Illiac Suite* [Hiller and Isaacson 1958]—a computer-generated composition for string quartet—testifies that this research field is almost as old as computer science. The computational modeling of music theories is interesting for multiple reasons. For musicians, the computational modeling helps to a better understanding of music theories. The resulting programs can serve as tools for computer-aided composition. For computer scientists, modeling music is interesting as the requirements of this domain can lead to new computational models and languages.

In the field of computer-aided composition (CAC, also known as algorithmic composition) composers formalize their musical intentions and implement these formal specifications as computer programs. These programs output music, and the composers then use this output in their pieces, possibly after manually editing it. Systematic surveys of techniques in CAC—with historical annotations—are provided by Roads

The work was supported by the EPSRC project "Learning the Structure of Music" (LeStruM), EPD063612-1.
Authors' addresses: T. Anders (corresponding author), E. R. Miranda, University of Plymouth, Interdisciplinary Centre for Computer Music Research (ICCMR), Smeaton Building, Room 206, Drake Circus, Plymouth PL4-8AA, United Kingdom; email: Torsten.Anders@beds.ac.uk.

DOI 10.1145/1978802.1978809 http://doi.acm.org/10.1145/1978802.1978809

[1996], Miranda [2001], and Taube [2004]. Assayag [1998] outlines the history of computer-aided composition. Papadopoulos and Wiggins [1999] offer a systematic overview with a focus on systems based on techniques from artificial intelligence. Berg [2009] surveys seminal algorithmic composition systems.

For centuries, explicit knowledge about music composition has been expressed by the means of rules. More than 1000 years ago, rules had been used already to describe the composition of an organum (an early polyphonic form) in the anonymous treatise *Musica enchiriadis* (about 900). Naturally, rules alone are not sufficient; they should be complemented by examples. Yet even today, rules are an important device when describing a musical style or when teaching the craft of composition.

Rules are well suited to describe music for two reasons: they are declarative, and they describe the multidimensional nature of music in a modular way. Firstly, rules are declarative in the sense that they rarely specify procedurally how to create a certain result. Instead, they only describe important features that the intended result should display. Secondly, the formalization of a task as complex as composition is greatly simplified when the task is stated in a modular way. When the task description is broken down into rhythmic rules, melodic rules, rules on the harmony and so forth, then the various musical dimensions are formalized one by one.

Because rules are such a long-established concept, programming approaches that support rule-like programming constructs have attracted much attention among composers and scholars for modeling music composition. These approaches translate the advantages of rules into the world of computer programming: implementing music theory models becomes declarative and modular.

As rule-based approaches are declarative, they free programmers to concentrate on *what* they want to do in a musical sense, that is, they do not need to define *how* to achieve this outcome. By contrast, it is very difficult to computationally model music theories by a procedural programming approach where the programmer details how to obtain a certain result. In addition, changing or adding a single rule can require redesigning the entire procedural program. These difficulties are shared by object-oriented programming, for that matter.

Another advantage of using rule-based approaches lies in the fact that music theory models are defined in a modular fashion. Multiple rules can even affect the same parameter value. For instance, a music theory model may restrict the note pitches of a score by melodic rules on the one hand and by harmonic rules on the other. Each of these separate rules affect the same parameter values, namely the pitches. However, no rule necessarily determines the parameter values fully. Search finds one or more solution that fulfils all rules.

Constraint Programming (CP) has proven a particularly successful programming paradigm for realizing ruled-based systems (another paradigm is logic programming). The attraction of CP is easily explained. CP allows users to model complex problems in a simple way. A problem is stated by a set of *variables* (unknowns) and *constraints* (relations) between these variables. In this article, we use the term *rule* for the music theoretical concept and the term *constraint* for its implementation by CP.

From a constraint programming research point of view, music is a challenging application domain. The complexity of music can be compared to the complexity of language, yet the modeling of both domains requires different approaches [Lerdahl and Jackendoff 1983]. A special music-related issue of the *Constraints* journal [Pachet and Codognet 2001], and a Musical Constraints Workshop that was part of the 7th International Conference on Principles and Practice of Constraint Programming in Cyprus (CP'2001) underline how music appeals to computer scientists.

Addressing the requirements of music can lead to new programming languages. For example, in order to realize his Bach harmonization system CHORAL in an efficient

$$X + Y = 7 \ \wedge \ X < Y$$

Fig. 1. A simple CSP example; possible problem solutions are $X = 3$, $Y = 4$ and $X = 1$, $Y = 6$.

way (Section 3.3), Ebcioglu [1987] first designed and implemented BSL (Backtracking Specification Language), a logic programming language that is fundamentally different from Prolog. There is a simple mapping that translates a BSL program to a formula of first-order predicate calculus. Executing a BSL program amounts to proving its corresponding first-order formula. While Prolog implements the backtracking search strategy, BSL uses backjumping for reducing the search space (Section 4.1.4).

Constraint programming research for music modeling can also lead to novel computational models. $PiCO$ [Rueda et al. 2001] is a calculus that integrates concurrent objects and constraints as primitive notions. This calculus is intended as a precise formal foundation for composition systems, because constraints and objects model two important approaches for constructing musical structures. This research also developed Cordial [Rueda et al. 1997], a visual programming language that shares semantics with $PiCO$. Another example is the *ntcc* calculus [Palamidessi and Valencia 2001] that extends the temporal concurrent constraint calculus *tcc* [Saraswat et al. 1994] by nondeterminism and unbounded finite delay. The design of the *ntcc* calculus was inspired by the requirements of musical applications [Rueda and Valencia 2004]. Nevertheless, instead of focusing on underlying programming models and languages, this article details the (range of) music theories supported by existing systems.

Plan of Article

The rest of this article is organized as follows. Section 2 provides a brief and informal introduction to CP. A wide range of existing musical constraint problems is outlined in Section 3, which sketches the common music theory disciplines, and surveys research that implements specific music theories. A number of systems allow users to implement their own music theories. These systems are described in Section 4; in Section 5 they are compared according to orthogonal evaluation criteria that indicate, for example, the range of music theories these systems support. The article ends with a conclusion (Section 6).

2. WHAT IS CONSTRAINT PROGRAMMING?

Constraint programming [Apt 2003; Dechter 2003] is a programming paradigm that introduces techniques to solve constraint satisfaction problems. A *Constraint Satisfaction Problem* (CSP) consists of a set of *variables* and mathematical relations between these variables which are referred to as *constraints*. Usually, a CSP presents a combinatorial problem. A *constraint solver* finds one or more solutions for the problem. A *solution* of a CSP shows for each variable of the problem a determined value that is consistent with all constraints. A constraint programming system (abridged: *constraint system*) allows its user to define and solve CSPs.

A simple numeric example may illustrate these concepts (Figure 1). The example introduces the two variables X and Y and restricts their value by two basic arithmetical operations, connected by a conjunction.[1]

Note that the term *variable* has a clearly different meaning in mainstream programming paradigms and languages (e.g., C or Java) on the one hand, and in the field of CP on the other hand. In mainstream programming languages, a variable denotes a *stateful* computational entity: such a variable has always a specific value and a program can alter the value of a variable with an assignment statement at any time. By contrast,

[1]As a convention, constrained variables are written with upper-case letters.

in CP the notion of a variable is more similar to the notion of a variable or *unknown* in mathematics. More specifically, a variable in CP is similar to a variable in first-order logic [Kelly 1997]. The value of a variable may be unknown or partially known. For example, it may only be known that $X \in \{1 \ldots 10\}$. However, a variable never changes its value; it is *stateless*.[2] The remainder of this text will always use the term variable in this latter meaning.

Sometimes, the term *constrained variable* will be used to explicitly denote a variable in the context of CP. A constrained variable is a variable that has a domain, that is a set of values it may take in a solution. Some constraint systems support variables with an infinite domain (e.g., the domain of all real numbers in some interval). Nevertheless, in all systems surveyed in this text, the domain of a variable is a finite set of potential variable values. The values in the domain are often all of the same type (e.g., Boolean domain, integer domain, or domain of finite sets of integers), while other systems support domains with mixed types.

In principle, constraints can be arbitrary mathematical relations. Examples include numerical relations, set relations, logic relations, and tree or graph relations. Constraint systems predefine a set of constraints and often allow the user to extend this set.

A CSP does not necessarily result in only a single solution. Instead, the restrictions and dependencies expressed by a set of constraints reduce the set of solution candidates.

Existing constraint systems can efficiently solve CSPs, a fact that has greatly contributed to the popularity of CP. Because the search space—that is the set of (partial) solution candidates—of a CSP is often huge, an efficient constraint solver has great impact on the usability of a constraint system.

There exists much literature on CP. For example, Apt [2003] provides a general overview of the field with many CSP examples. Frühwirth and Abdennadher [2003] survey different CP approaches and systems. Dechter [2003] primarily explains how constraint solvers find solutions. A collection of surveys covering the full breadth and depth of the CP research field is presented in Rossi et al. [2006]. Finally, a less formal introduction into the field is given in the Web tutorial by Bartak [1998], which also links to further resources such as related journals, conferences, Web links, etc.

3. MUSICAL CONSTRAINT SATISFACTION PROBLEMS

Using CP, a composition task is stated by: (i) a music representation in which some musical aspects are unknown (and therefore represented by variables) and (ii) constraints that restrict these variables. For instance, a chord can be expressed by a set of notes, and the note pitches can be variables. Some harmonic constraints may specify how the chord pitches are related to each other, other constraints define the relation to the pitches of other chords and so forth.

A musical CSP implements a model of a music theory that describes certain musical aspects such as the rhythm, harmony, the formal structure, or the instrumentation. The theory model does not necessarily need to be consistent with any existing musical style. For instance, a composer may develop some musical CSP (and implicitly define a theory model) in an ad hoc manner in order to generate some subpart of a composition in a novel style. However, the music theory model must be fully formalized (e.g., fully expressible in mathematical notation). When a musical CSP is solved, the variables in the music representation are determined in a way that complies with the modeled

[2]Even if the domain of a variable will be reduced during the search process, the domain will not change in any other way. Yet, in the actual implementation of some systems discussed in what follows the concept of the variable and its domain is somewhat decoupled. In such systems, during search the variable is indeed statefully bound to different values of the domain before constraints are applied.

theory. The result of a musical CSP can be, for example, a musical segment (e.g., a sequence of plain pitches), an analysis of a given piece (e.g., a harmonic analysis), or a full-scale composition.

Many music theories have been modeled and implemented by CP. In addition, composers have already made extensive use of CP. Examples include Antoine Bonnet (e.g., for *Épitaphe* for 8 brass instruments, 2 pianos, orchestra, and electro-acoustics, 1992–1994 [Bresson et al. 2005]); Magnus Lindberg (*Engine* for chamber orchestra, 1996 [Rueda et al. 1998]); Georges Bloch (*Palm Sax* for seven saxophones, [Rueda et al. 1998]), Örjan Sandred (*Kalejdoskop* for clarinet, viola, and piano, 1999 [Sandred 2003]); Jacopo Baboni Schilingi (*Concubia nocte, in memoria di Luciano Berio* for soprano and live computer, 2003);[3] Johannes Kretz (*second horizon* for piano and orchestra, 2002 [Kretz 2003]); and Hans Tutschku (*Die Süsse unserer traurigen Kindheit*, music theatre, 2005).[4]

The CP paradigm is well suited to the needs of computer-aided composition. Composers often prefer a way of working that is situated somewhere between composing "by hand" and formalizing the composition process such that it can be delegated to the computer; CP supports this way of working very well [Anders and Miranda 2009b]. For example, composers can determine some aspects of the music (e.g., certain pitches) by hand and restrict other aspects by constraints. Alternatively, composers may specify the high-level structure (e.g., the formal structure) manually and let the computer fill in the details. Furthermore, composers usually do not fully formalize certain aspects of the composition process before they start composing. Instead, the formalization is often an integral aspect of the composition process itself. A composition task defined by means of CP can be shaped in a highly flexible way during the composition process by the adding, removing, and changing of individual constraints.

This section demonstrates the variety of musical CSPs that have been published. We first describe a simple (but nontrivial) musical constraint satisfaction example in full detail. Then we briefly introduce classical music theory disciplines, and survey work that has implemented these theories using CP. Classical music theory disciplines focus on pitches, and therefore most of the presented musical CSPs primarily constrain note pitches in some way.

3.1. A First Example: All-Interval Series

This section presents a first musical CSP example that stems from serial music composition. In this technique, the composer organizes the pitches in a composition with the help of a tone row (also known as twelve-tone series). A *tone row* is a sequence of 12 tones that arranges the twelve chromatic pitch classes (tone names) in a particular order, where each pitch class occurs exactly once. A row can be transformed in many ways. The most common transformations are transposition, retrograde (reversal in time), and inversion (mirrored along a pitch axis). Composers use tone rows as a device to achieve coherency in music, even if the music abandons conventional harmony. A seminal textbook on serial composition is Perle [1991], and Křenek [1952] provides a practical introduction to this composition technique.

There have been many cases where composers took great care in designing their tone rows. A prominent example of special tone rows is the set of *all-interval series*. In an all-interval series, not only the 12 pitch classes but also the eleven intervals between them are pairwise distinct (i.e., each interval occurs only once). Another example for

[3]Personal communication at PRISMA (Pedagogia e Ricerca sui Sistemi Musicali Assistiti) meeting, January 2004 at Centro Tempo Reale in Florence.
[4]Personal communication at PRISMA meeting, June 2006 at IRCAM, Paris.

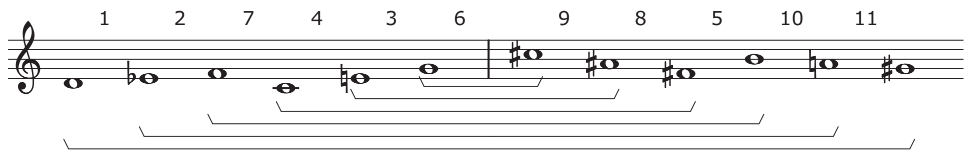

Fig. 2. All-interval series, whose two halves also form transposed retrogrades of each other.

(a) $PitchClasses :=$ a list of 12 undetermined integers, each with the domain $\{0, \ldots, 11\}$

$Intervals :=$ a list of 11 undetermined integers, each with the domain $\{1, \ldots, 11\}$

/* Constrains relation between $PitchClasses$ and $Intervals$ */

$$\bigwedge_{i=1}^{11} inversionalEquivalentInterval(PitchClasses_i, PitchClasses_{i+1}, Intervals_i)$$

/* All elements in $PitchClasses$ and $Intervals$ are pairwise distinct */

$\wedge\ distinct(PitchClasses)$

$\wedge\ distinct(Intervals)$

/* No series transpositions: first pitch fixed to 0 */

$\wedge\ PitchClasses_1 = 0$

/* Interval between outer tones is always 6, a tritone [Křenek 1952, p. 50]. This redundant constraint can reduce the search space considerably, depending on the solver */

$\wedge\ PitchClasses_{12} = 6$

(b) /* Constraints $Interval$ to an inversional equivalent interval between the pitch classes PC_1 and PC_2 (a fifth upwards and a fourth downwards are the same interval, namely 7). */

$inversionalEquivalentInterval(PC_1, PC_2, Interval) := \ Interval = (PC_2 - PC_1) \bmod 12$

Fig. 3. A musical CSP example: an all-interval series definition.

special tone rows are symmetric rows, in which some section of the row is a strict transformation of another section.

Figure 2 shows an all-interval series example (cited in Gervink [1995] and Křenek [1952]). It can be seen that each pitch class occurs only once in the series. Similarly, also every interval between the pitches is unique (reported by integers above the staff, measured in semitones). These intervals are computed in such a way that they are inversional equivalent: complementary intervals such a fifth upwards and a fourth downwards count as the same interval (namely 7). In this particular case, the series is even both an all-interval series and a symmetric series (namely transposed retrograde).

We now present a full all-interval series definition, in order to demonstrate how a musical CSP may look (Figure 3). The definition first creates a music representation, consisting of the two lists *PitchClasses* and *Intervals*. The list *PitchClasses* represents the sequence of pitch classes, that is the all-interval series solution. The list *Intervals* represents the sequence of the intervals between these pitch classes. Each pitch class is represented by an integer between 0 and 11 (i.e., between *c* and *b*). Each interval is represented by an integer between 1 and 11 (0 would be unison, but this interval does not occur in a twelve-tone row). The pitch classes and intervals are the constrained variables in this problem: their values will only be found during the search by the constraint solver.

The rest of the definition consists of a conjunction of constraints on this music representation. First, the relation between all pairs of consecutive pitch classes and the intervals between them is restricted by the constraint *inversionalEquivalentInterval* (see Figure 3(b)). The constraint *distinct* enforces that all variables in the lists *PitchClasses*

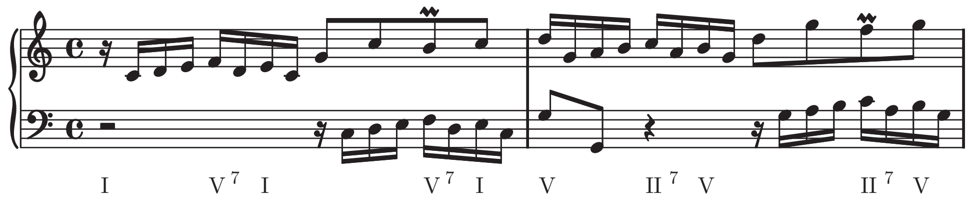

Fig. 4. Polyphonic music consists of multiple voices which accompany each other; Baroque music expresses an implicit harmonic progression (J.S. Bach, Inventio 1, beginning).

and *Intervals* are pairwise distinct. Finally, two additional constraints determine the first and the last pitch class in order to prohibit series transpositions, and to amplify the information available to the constraint solver for efficiency. This definition holds the 3856 solutions known from the literature [Morris and Starr 1974].

This article introduces some new terminology in order to simplify the discussion of constraints and their effect in a musical CSP. We will use this terminology later when describing and comparing different systems in which users can implement their own CSPs. This text refers to sets of variables (or score objects) that are interrelated in the same way as instances of the same *score context*.[5] Examples include the sets of consecutive note pairs in a melody, sets of simultaneous notes in a score, sets of intervals between simultaneous notes, sets of all score objects that belong to a single bar and so forth.

This text also introduces the term *constraint scope* to denote all score contexts to which a single constraint is applied (a set of sets of variables or score objects). In music CP, a constraint is often uniformly applied to multiple sets of variables or score objects. For example, the constraint *inversionalEquivalentInterval* is applied to 11 variable sets in Figure 3.

CP is only one way to implement music theory models. Morris and Starr [1974] present a specially developed algorithm to compute all-interval series. Nevertheless, music CP has the advantage that the user only needs to state the problem declaratively without developing a special algorithm, which is often a very time-consuming activity. For example, a constraint system user may easily add additional constraints that require the solution to be both an all-interval series and a symmetrical series (as the example row in Figure 2). By contrast, using an algorithmic approach often requires a laborious algorithm redesign when the problem specification changes. Nevertheless, a specially designed algorithm is often more efficiently executed.

The rest of this section surveys classical music theory disciplines and constraint systems that implement them.

3.2. Counterpoint

Polyphonic music consists of multiple voices that accompany each other. Figure 4 presents an example. Over the centuries, many counterpoint textbooks have been written that teach how to compose polyphonic music. Different textbooks often cover different musical styles. Today, two style families are taught most frequently. One approach is oriented at Renaissance music and the composer Palestrina in particular. Fux [1725] wrote the seminal treatise on this approach, while the classical textbook by Jeppesen [1930] covers the style of Palestrina more accurately. Whereas in the first approach harmonic considerations are at best secondary, the other—and historically younger—approach teaches how to compose polyphonic music that expresses a harmonic

[5]The term score context is inspired by Lilypond [Nienhuys and Nieuwenhuizen 2003] and PWConstraints [Laurson 1996], however, the term is redefined here with a much more general meaning.

progression. Baroque music, for example, usually follows this approach (Figure 4). Important textbooks on harmonic counterpoint are Schoenberg [1964] and Piston [1947]. Finally, some authors teach several different contrapuntal styles [Motte 1981].

The 20th and 21st century saw further developments of polyphonic music. These are rarely covered by counterpoint textbooks, but nevertheless have been addressed in musical CSPs. Important examples include the dodecaphonic technique of Schoenberg and others [Perle 1991] (refer to the preceding all-interval series example), its descendants like serialism or Stockhausen's formula composition [Conen 1991], the micropolyphony developed by Ligeti [Bernard 1994], and Nancarrow's rhythmical counterpoint [Gann 2006].

There exist several systems creating polyphonic music by means of CP. Scholastic counterpoint (e.g., the faithful application of Fux [1725]) features a particular strict set of rules when compared with other music theory subdisciplines, for example, rhythm or form. A strict rule set makes formal modeling more easy, which explains why counterpoint has been of great interest for designers of rule-based systems. Nevertheless, harmonic counterpoint has rarely been addressed because it is more complex, as it implies a theory of harmony model.

Ebcioglu [1980] proposes a system for creating two-part florid counterpoint: to a given cantus firmus the system composes a matching voice which is rhythmically independent. The author lists almost 50 implemented counterpoint constraints, which include complex high-level constraints such as "the pitches of different local maxima (i.e., melodic peaks) within three measures of the voice are unique". Sources for constraints were Joseph Marx and Charles Koechlin. Because their rules were insufficient for automatic composition, Ebcioglu added rules of his own. The search strategy embeds heuristics that prefer steps to skips and note pitches that have not occurred before. Two solutions, presented by Ebcioglu [1980], demonstrate that the system achieves its goal—to create typical "conservatory level" counterpoint exercises—pretty well.

A system for creating species counterpoint was introduced by Schottstaedt [1989], who aimed to follow the rule set of Fux [1725] as closely as possible. The system implements all five species for up to six voices. However, the author modified the original Fuxian rule set (more then 40 rules are quoted in article) to get closer to Fux' actual examples. In accordance with music theorists (including Fux) that state that rules are merely guidelines and no absolutes, the system assigns each constraint a numeric penalty value to denote its relative importance: the system searches for a solution with a small accumulated penalty. Compared with other counterpoint studies, Schottstaedt [1989] achieved relatively advanced examples (e.g., fifth species for five voices). Still, the shown musical results reveal some limitations of the system's rule set: in particular, the rhythmic structure is atypical for Palestrina style (almost march-like), and the melodies contain many large skips, in contrast to the Fuxian examples.

Polyphonic music in the style of Josquin des Prez is addressed by Laurson [1996], who implemented several rules from the Josquin chapter of Motte [1981]. Nevertheless, the goal of this research is not so much to simulate a specific historical style, but to study the problem of polyphonic CSPs in general. The resulting system PWConstraints and its subsystem Score-PMC are further discussed shortly.

Several researchers addressed polyphonic problems outside the canon of conventional counterpoint. Musical CSPs that create dense Ligeti-like textures have been proposed by Laurson and Kuuskankare [2001] and Chemillier and Truchet [2001]. Jones [2000] developed a pragmatic tool for composers that implements atonal counterpoint. Finally, Chemillier and Truchet [2001] modeled a two-voice canon following rules that are typical for the Nzakara harp repertoire from the Central African Republic using CP.

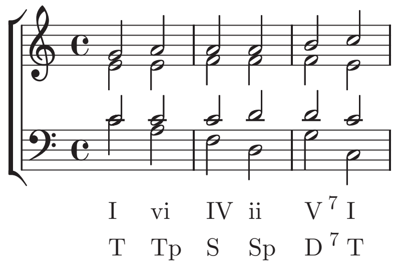

Fig. 5. Different approaches to harmony point out different harmonic relations: comparison of Roman numerals vs. functional analysis (see footnote 6).

3.3. Harmony

As with counterpoint, there exists a huge amount of literature on the subject of harmony. Authors differ less in their focus on a certain style; the history of harmony shows a more continuous development when compared with counterpoint. However, there exist different approaches to explain harmonic phenomena. Most authors describe harmonic progressions as progressions of chord roots and analyse all chords in terms of their relationship with the tonic. This idea goes back to Rameau [1722], but different conclusions are drawn by later authors. One approach is based on the assumption that chord roots indicate one of the seven (major or minor) scale degrees, which are conventionally notated by Roman numerals, lowercase letters are commonly used for minor chords. Schoenberg [1911] wrote a particularly accomplished textbook using this approach. Another approach (often called functional harmony) only accepts three different main harmonic functions, namely the tonic, dominant and subdominant; usually notated with their initials T, D, and S. This approach explains all chords as variants of these main functions [Riemann 1887]. The approach based on scale degrees highlights the diatonic interval between chord roots, whereas functional harmony denotes which chords can substitute each other. Figure 5 compares both approaches.[6] Finally, Schenker [1935] and Schoenberg [1969] put particular emphasis on larger-scale structures in harmonic progressions.

Like counterpoint, the development of harmony and its study are still ongoing and these developments are of particular interest for composers using constraint systems. For example, the harmonic language of "atonal music"[7], music in 12-tone equal temperament without a tonal center and often consisting of highly complex chords, is described in the influential work by Forte [1973] in terms of pitch class sets.[8] Microtonal music, and in particular music in just intonation, is another important example for the ongoing development. The seminal work on just intonation is Partch [1974], while Doty [2002] wrote a tutorial on the subject, and an extensive encyclopedia on just intonation theory is presented by Monzo [2005].

[6]The root of the fourth chord in Figure 5 is *D*, which is the second degree (*ii*) in C major. The interval between the root of this and the next chord is a fourth, which is reflected by their ordinals (*ii* and V^7, the 7 indicates that a seventh is added to the chord). This fourth chord in Figure 5 is also the relative minor of the subdominant, in Riemann's terminology the subdominant parallel (*Sp*): Riemann argues that variants of a chord (e.g., *S* and *Sp*) can substitute each other.

[7]Schoenberg—who was the first to radically break out of traditional tonic-related harmony—detested the term "atonal music"; he preferred "pantonal music" instead.

[8]Numeric pitch classes have been very successful for the computational modeling of music, including tonal music. For example, the C-major triad C, E, G is represented by the pitch class set $\{0, 4, 7\}$. Many systems discussed later in this article make use of this representation internally.

Much research has been carried out on constraint-based harmonization. Pachet and Roy [2001] provide a survey on this subject. The rest of this subsection reviews constraint-based harmonization systems.

CHORAL [Ebcioglu 1987; 1992] is a system that creates four-part harmonizations in the style of Johann Sebastian Bach for given choral melodies. CHORAL received much attention for the musical quality of its output: according to its author, CHORAL accomplished the competence of a talented music student. Ebcioglu's detailed analysis of compositions by Bach resulted in the impressive amount of about 350 constraints implemented by the system. These address two subtasks: the harmonization (creating the chord skeleton, style-appropriate modulation, and cadencing) and melody generation (with special care of the outer voices).

The often-cited article by Tsang and Aitken [1991] proposes a lucid system with a small set of 20 constraints, which creates four-part harmonizations of a choral melody.

The system designed by Ramirez and Peralta [1998] also automatically harmonizes a given melody with an appropriate chord sequence. The system finds a sequence of absolute chord names such as C, Dm, G, C,[9] whereby the system is limited to single key melodies and only considers diatonic triads in the solution. To increase the musical quality, the system further constrains solutions to follow standard chord patterns (e.g., I, II, V, I) stored in a database.

Coppelia [Zimmermann 2001] creates homophonic chord progressions that also feature a rhythmical structure. The music theory model is split into two layered submodels which are implemented by two independent applications. The subsystem Aaron creates a harmonic plan, represented by harmonic functions in the tradition of Hugo Riemann (such as T, S_3, D^7, T) and complements this plan by additional information (e.g., the duration of each chord and further restrictions on single voices such as "the soprano melody shall move downward"). COMPOzE [Henz et al. 1996], the second subsystem, creates the actual four-voice chord progression from this harmonic plan.

Phon-Amnuaisuk presents another system that creates choral harmonizations in the style of Johann Sebastian Bach [Phon-Amnuaisuk 2001; 2002]. Phon-Amnuaisuk criticizes CHORAL [Ebcioglu 1992] asserting that this system is hard to modify. To realize a more adaptable constraint system design, he proposes a control language that regulates the temporal order of decisions during the composition process (variable ordering). For the four-voice Bach choral example, the search process may first create the harmonic skeleton for the given melody, then outline the bass skeleton, create a draft of the other voices, and eventually create the final version of each voice by adding ornamentations such as passing notes.

Anders and Miranda [2009a] model Schoenberg's guidelines for convincing chord progressions [Schoenberg 1911]. While most other systems harmonize a given melody—often creating a new chord for each melody note (choral harmonization)—this model creates a harmonic progression from scratch. The work models Schoenberg's explanation of his recommendations instead of the actual rules, and that way generalizes these recommendations beyond diatonic progressions, even for microtonal music.

A CSP based on dodecaphonic pitch class sets is described by Laurson [1996]. In this CSP, a solution consists of a sequence of (possibly overlapping) pitch class sets: such solutions could be used by a composer to organize the harmonic structure of music.

3.4. Melody and Form

Melody-writing is highly style-dependent, and this subject is traditionally less established in music theory than counterpoint and harmony. Nevertheless, the subject

[9]The authors also forbeared to formalize melodic and voice-leading rules: the system only creates a chord name sequence.

is covered, for example, by some general textbooks on composition. For instance, Schoenberg [1943, 1967, 1995] (the three textbooks are sorted according to their intended audience from entry-level to advanced) explains how in classical music a melody expresses the underlying harmony and how a melody is composed from motifs and their variations. Whereas Schoenberg teaches melody composition in a more systematic way, Motte [1993] studies various aspects of melodies from different musical styles (ranging from Gegorian chant to Ligeti, including children's and political battle songs).

Research on modeling melody and form is still at an early stage. At the end of their survey on constraint-based harmonization, Pachet and Roy [2001] point out: "However, what remains unsolved is the problem of producing musically nice or interesting melodies."

The work of Löthe [1999] constitutes one of the few literature examples on rule-based melody composition. Löthe's system creates minuet melodies (in early classical style) over a given harmonic progression. The author describes several example rules (based on several sources including music theory literature from the classical period, for example, Koch [1793]) in detail and demonstrates the effect of different rule sets with musical examples. However, these rules are actually only rules from harmonic counterpoint applied to a single melodic line; they do not address musical form.

Motifs and their variation are very important for classical melody composition and form. Anders [2009] argues that the foundation for the successful modeling of harmony was the modeling of harmonic concepts such as pitches, intervals, chords, and scales. As a first step towards a model of classical melody composition, the author presents a formal model of musical motifs. The model defines the relation between the music representation of a specific motif instance, the distinctive features of a more abstract motif description, and how these features are varied in different motif instances. Because the model is implemented by constraints, the resulting motifs can also depend on nonmotivic musical conditions such as harmonic, melodic, or rhythmic constraints.

3.5. Rhythm

Over centuries, western music focused on developing the pitch structure instead of rhythm, which may be the reason why also music theory largely neglected the rhythmic aspect. In one of the rare textbooks on the subject, Cooper and Meyer [1960] clearly define rhythmical terms (such as pulse, meter, rhythm, accent, stress, tie, syncopation, and suspension) and explain their relations in a theory of rhythm which studies the hierarchical nature of rhythmical organization. Yet, the 20th century saw considerable developments of the rhythmical aspect. Examples include the "additive" rhythmical permutations of motifs resulting in changes of the metric structure introduced by Stravinsky [Boulez and Nattiez 1990] and later further developed by Messiaen [1944], Nancarrow's already mentioned rhythmical counterpoint, the phasing technique in the music of Reich [2002], and the astonishingly complex rhythmical structures in the work of Ferneyhough [1995].

Some musical CSPs in the literature are of a purely rhythmical nature. Truchet et al. [2001] propose a polyrhythmic problem in which each voice literally repeats a rhythmical pattern but common onsets between the voices are avoided.

A system completely devoted to rhythmical CSPs is OMRC [Sandred 2000, 2003]. For example, Sandred [2004] proposes a constraint-based quantification of the rhythms of everyday gestures (e.g., extracted from the sound of a passing train) and forces these gestures into readable music notation. Constraints may control what time signatures are allowed and how often the time signature may change. Additionally, the composer may apply further constraints. For example, the composer may demand that the quantified result will be build from precomposed motifs.

3.6. Instrumentation and Orchestration

The art of instrumentation involves writing for different instruments taking particular playing techniques and limitations of instruments into account. The discipline orchestration additionally addresses how to sonically balance instruments. Berlioz and Strauss [1904] wrote a seminal textbook on orchestration. Instrumentation and orchestration are still developing today. An important example is the music of Lachenmann, which introduced many new sounds by novel playing techniques.

Laurson and Kuuskankare [2000, 2001] present musical CSPs in which melodic, harmonic, and voice leading constraints are complemented by instrumentation constraints, and they are presenting solutions showing idiomatic instrumental writing. The authors discuss guitar and brass instrument fingering in two case studies. For example, writing music for the guitar in a way that is well playable requires instrument-typical considerations. Guitar music is performed on six strings (tuned in a particular way) on which only four left-hand fingers are placed. Furthermore, the fingers can only be stretched up to a certain amount and moving the fingers requires a certain amount of time.

Recent work by Carpentier and Bresson [2011] proposes an orchestration system. Users provide a target sound (for example, a recording or sound synthesis output) and the system searches for an instrument combination (a symbolic score) that imitates this sound. Users can restrict the solution by quantitative constraints on the attributes of all notes of a time slice, such as how many instruments and pitches can be involved, or what their minimum dynamics should be. Nevertheless, the system does not take instrumentation knowledge (as found in instrumentation treatises) into account.

4. GENERIC MUSIC CONSTRAINT SYSTEMS

The systems surveyed in the preceding Section 3 demonstrate that much research has been conducted into implementing music theory models by CP. Most of these systems were designed exclusively to solve a single CSP or a small range of problems (e.g., the automatic harmonization of a given melody).

Although some authors report that it is “not particularly difficult to write such a program” [Schottstaedt 1989], the design of a system that solves a complex CSP with reasonable efficiency is demanding. For example, Ebcioglu [1992] deemed it necessary to first develop a new programming language for this task (namely BSL).

Moreover, music constraint systems share important requirements. Firstly, any music constraint system requires some constraint solver. Secondly, domain-specific CSPs share a considerable amount of domain-specific knowledge: all musical CSPs require modeling of musical knowledge. For instance, concepts such as note, pitch, or voice are required in a large number of musical CSPs. Whenever a musical CSP is defined “from scratch”, all this knowledge must be modeled anew. Domain-specific optimizations of the search process must also be carried out again. Consequently, several more generic systems for music CP were proposed since the early nineties that greatly simplify the definition of musical CSPs.

This article introduces the term *generic music constraint system* to indicate a system that is designed to solve a considerable number of musical CSPs, in contrast to a system designed specifically to solve a single CSP or a small set of problems. A generic system is often developed for users who would hardly consider the design of a new constraint system from scratch—such as composers and music theorists—but who can greatly contribute to the research in this field.

A generic music constraint system allows users to define and solve their own musical CSPs. In order to be more generic, these systems usually aim to be style-neutral: they support a wide range of musical styles.

Generic music constraint systems implement the following components: a music representation, a constraint formalism, and a search strategy. Each system features a *music representation*: it represents score information such as the pitches, notes, or voices in the solution. This information can be known in the CSP definition, or it can be unknown and expressed by constrained variables. For example, the durations of notes in the score or the underlying harmonic structure may be unknown. A *constraint formalism* specifies how constraints are defined and applied to the music representation. Finally, the solver of each system implements a particular *search strategy*. In the following, we detail several generic music constraint systems, and we will always look at these three components. In addition, we will also discuss the user interface of some systems.

The following five systems are discussed next: PWConstraints (Section 4.1); Situation (Section 4.2); MusES with BackTalk (Section 4.3); OMClouds (Section 4.4); and Strasheela (Section 4.5). These systems are studied in chronological order, though their development history may overlap with one another, as these systems have been developed over several years.

For the sake of completeness, a number of further systems should be mentioned briefly. These systems are either not in active use anymore, or extend some other system. The number of existing systems underlines the interest in CP for modeling music theories. Carla [Courtot 1990] is a pioneering generic music constraint system. Arno [Anders 2000] supports CSPs on the music representation of the composition system Common Music [Taube 1997] using the constraint solver Screamer [Siskind and McAllester 1993]. Finally, some systems extend PWConstraints. JBS-Constraints is a large collection of predefined constraints for PWConstraints [Schilingi 2009]. Sandred, the author of OMRC (see preceding), recently proposed a new system PWMC that extends the capabilities of PWConstraints by introducing an extra music representation [Sandred 2009, 2011].

4.1. PWConstraints

This section introduces the CP language PWConstraints [Laurson 1996; Rueda et al. 1998]. We study this system in much detail, because it constitutes a fine example for introducing important problems and approaches of generic music constraint systems in general. Other systems are then presented more briefly.

PWConstraints was originally developed as a library on top of PatchWork [Laurson 1996; Assayag et al. 1999], a visual programming language for computer-aided composition implemented in Common Lisp. PatchWork meanwhile developed into PWGL [Laurson et al. 2009] and the library is now called PWGLConstraints. Nevertheless, this article uses the old name PWConstraints to refer to both versions of the system because from the user's point of view these systems are very similar. Also, most of the available literature actually discusses the old system.

PWConstraints consists of two main music constraint systems that share a similar search strategy: PMC and Score-PMC. PMC is a general CP language for constraining simple data-types, in particular lists of integers. Score-PMC is a special CP language for polyphonic CSPs.

Programming constructs in PatchWork (and PWGL) present themselves as graphical boxes, which is typical for many visual programming languages. Consequently, each main part of PWConstraints comes as such a box: there is a box for PMC and for Score-PMC. However, PWConstraints complements its graphical interface by a textual interface: the actual CSP is defined by textual Lisp code, using functions/macros provided by PWConstraints.

4.1.1. The Music Representation of PMC. The music representation format of PMC consists of a list of constrained variables. PMC supports variables with universal domain,

$$[\{57, \ldots, 69\}, \{57, \ldots, 69\}, \ldots]$$

Fig. 6. Declaring the domains of melody pitches for PMC.

that is variable domains can consist of arbitrary values. Common domain examples are integers and symbols. The number of variables (i.e., the length of the list) is always determined in the CSP definition.

As an example, let us assume a composer wants to create a choral melody. She is only interested in the note pitches, and therefore a plain sequence of pitches is sufficient. All melody pitches are situated in the interval between $a3$ (an octave below concert pitch) and $a4$ (concert pitch). Using MIDI pitch numbers [Selfridge-Field 1997], the domain $\{a3, \ldots, a4\}$ is encoded as the domain of integers $\{57, \ldots, 69\}$ (Figure 6).

Users can freely interpret variable values. For example, a sequence of integers can be interpreted as the MIDI-pitches of a melody or as duration values. A list of lists of integers can be interpreted, for instance, as a chord sequence (where a list of integers represents the chord pitches by MIDI numbers without specifying the chord duration) or as a sequence of rhythmic motifs (where a list of integers represents a sequence of note durations).

Users may determine certain variables in the CSP definition. For example, in a list of pitch variables users may specify that any solution begins and ends with a certain pitch by specifying this pitch as the only domain value for these variables.

4.1.2. The Music Representation of Score-PMC. PMC is only suited for CSPs that can be defined by constraining a variable sequence. Various musical CSPs can be expressed that way, particularly music theory models that belong to some preparatory stage for the actual composition process, for example, the creation of a purely rhythmical figure, a tone row (see preceding), or a harmonic progression.

However, most western music is polyphonic by nature. Here, the term polyphony is used in a more general meaning than usual: most music is organized in multiple layers, for example, multiple voices, a melody plus accompaniment, or any other musical texture where multiple events are played simultaneously.

A constraint system must take the polyphonic nature of music into account for most CSPs outside the category of preparative CSPs. Yet, polyphonic CSPs are hard to express by only a sequential music representation. A constraint system with only a sequentially representation can in principle represent complex polyphonic solutions—after all, music representation formats like a MIDI file [Selfridge-Field 1997] or a Csound score [Boulanger 2000] are sequential and can express highly complex scores. Nevertheless, it is still very hard to express polyphonic CSPs with a sequential music representation format.

Polyphonic CSPs often require constraining complex score contexts (remember that sets of score objects that are interrelated in the same way are instances of the same score context, Section 3.1). For instance, a common contrapuntal constraint permits dissonant note pitches in situations where several conditions are met that involve various musical aspects: a note may be dissonant in cases where it is a passing tone on a weak beat and below a certain duration. The score context of this constraint involves harmonic information (whether a certain note is dissonant); melodic information (whether this note is a passing tone); metric information (whether this note is on an weak beat); and rhythmical information (the note's duration). It is hard to express and constrain such a complex score context in a sequential representation, because the information required to deduce which score objects or variables belong to such a context is missing.

This problem is overcome by adding information to the music representation. Research into music representation suggests an important approach for adding information that marks score contexts: the hierarchic organization of the representation in a

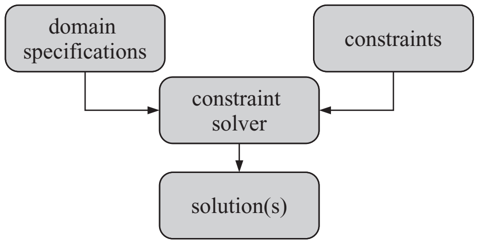

Fig. 7. In PWConstraints, the variable domain specifications and the constraints of a CSP are handed independently to the constraint solver.

tree or even a graph [Dannenberg 1993; Wiggins et al. 1993]. For example, the information which note belongs to which voice can be expressed adequately by grouping all notes of a specific voice in a voice container.

The PWConstraints subsystem Score-PMC addresses this problem by defining a more elaborate music representation that explicitly supports additional score contexts.[10] In contrast to the one-dimensional music representation of PMC, the music representation of Score-PMC resembles more a two-dimensional "musical map". Here, the main "dimensions" are notes in sequential order in a voice, and simultaneous notes. More specifically, the music representation of Score-PMC supports the following contexts.

—*Melodic Context.* For any voice in a polyphonic score, the music representation stores the horizontal order of notes.
—*Harmonic Context.* A set of notes that are sounding concurrently at the attack time of a given note.
—*Metric Context.* The metric context of a given note expresses the rhythmic pattern this note belongs to (e.g., a succession of an eighth-triplet) and the position of this note in the pattern (e.g., second note of the triplet).

An important restriction of Score-PMC lies in the fact that note *pitches* are the only variables in the music representation. All other aspects—in particular the rhythmic structure of the score—must be fully determined in the CSP definition; nevertheless, the rhythmic structure can be arbitrarily complex. The reason for this limitation is the search strategy implemented by Score-PMC, which is optimized for CSPs with predefined rhythmical structure (see Section 4.1.4).

In Texture-PMC [Laurson and Kuuskankare 2001], an alternative extension of PW-Constraints, other note parameters can also be constrained (e.g., rhythmic values). However, the program still requires a predefined rhythmical structure, which is potentially overwritten during the search process. Internally, all variables are still encoded as note pitches.

Score-PMC does not support user extensions. For example, users cannot define additional score contexts that may be required for complex CSPs.

4.1.3. The Constraint Formalism.

After introducing the music representations of PWConstraints as given before, this section explains how constraints are defined and applied to these music representations.

In PMC, a CSP is defined by complementing the declaration of the variable domains by constraints that pose restrictions on a solution. Figure 7 points out that the domains and constraints are given separately to the constraint solver. Handing multiple constraints to the solver implicitly expresses a conjunction of all these constraints. The

[10]PWConstraints also uses the term *score context*, but uses it in a more narrow meaning, which only denotes the contexts listed in this subsection.

$[*,\ \mathit{Pitch}_1,\ \mathit{Pitch}_2]$	Pattern-matching expression
let $\mathit{Interval}$	Constraint body
in $\mathit{Interval} = \lvert \mathit{Pitch}_1 - \mathit{Pitch}_2 \rvert$	
$\wedge\ \mathit{Interval} \in \{0, \ldots, 7\}$	

Fig. 8. A PMC constraint definition example which states that the interval between any two consecutive pitches must not exceed a fifth (i.e., seven semitones).

actual variables are created by the system "behind the stage". Score-PMC uses the same approach, but in addition to the variable domains and the constraints a fully determined score is given to the constraint solver. As only note pitches are variables in Score-PMC, this score determines all other aspects, in particular the rhythmical structure.

Every music constraint system must support some mechanism to apply constraints to variables in the music representation. This mechanism also controls the scope of a constraint (remember that the scope of a constraint is the set of all variable sets to which a single constraint is applied).

PWConstraints introduces a pattern matching mechanism for this purpose. Pattern matching is well-known, for example, from the UNIX shell, where a pattern is a placeholder for a character sequence. The pattern `*test.txt` matches the file names `mytest.txt` as well as `my-other-test.txt`.

The pattern matching mechanism of PMC introduces a mini language that denotes the position of elements in the sequential music representation. This language consists of only three constructs.[11] There are two place-holders: the place-holder symbol ? matches exactly one sequence element, and the symbol $*$ matches zero or more elements. The third construct consists of pattern-matching variables.

An example will illustrate this pattern-matching language. In the expression $[?, *, P_1, P_2]$, the place-holder ? matches the first element in the sequential music representation. The symbol $*$ matches either no element, or the second, or the second and the third and so forth. Consequently, the two pattern-matching variables P_1 and P_2 match any pair of two successive elements, except the first pair. Note that no final $*$ is required that would match the rest of the sequence.

A PWConstraints constraint definition consists of a pattern-matching part and a constraint body. Figure 8[12] presents the definition of a melodic constraint that restricts the interval between two successive pitches Pitch_1 and Pitch_2 to a fifth as maximum (i.e., seven semitones).[13] The pattern matches any two consecutive elements in a sequence of variables (the music representation of PMC). The pattern-matching variables Pitch_1 and Pitch_2 bind the free variables in the constraint body definition. The body of a constraint is always an expression that returns a Boolean value. For every solution to a CSP, PWConstraints makes sure that the body of a constraint returns true for every match of its corresponding pattern matching expression. Because the pitches are encoded numerically, users can define numeric relations between them.

[11]PWConstraints also introduces a fourth construct, index variables, for convenience. However, all patterns using index variables can be reproduced with the three constructs introduced in this section.

[12]In PWConstraints, constraints are defined in Lisp; this text uses a notation based on first-order logic instead (and leaves out some formal details for simplicity).

[13]This constraint is a simplified version of a Palestrina-style counterpoint rule, which is reflected by many counterpoint treatises. In its strict form, it permits intervals between a minor second and a fifth, or an octave. Upwards, also the minor sixth is allowed. Additionally, the rule prohibits all diminished and augmented melodic intervals [Jeppesen 1939].

The example introduces a new constrained variable *Interval* and applies a conjunction of two constraints.

As explained earlier, the music representation of PMC is a flat list of variables. A constraint application mechanism based on pattern matching is convenient for such a data structure. However, polyphonic CSPs often include nonsequential score object sets (e.g., simultaneous notes which occur in different voices). PWConstraints' polyphonic subsystem Score-PMC therefore defines a hierarchical music representation and supports the context's melodic note sequence, simultaneous notes, and the metric position of a note (see preceding). Score-PMC simplifies the application of constraints to these contexts by extending the pattern matching language of PMC [Laurson and Kuuskankare 2005]. Instead of matching only individual variables, additional language keywords support the matching of notes (a compound data structure) in a sequence of melody notes, simultaneous note sets, and note sets at specific metric positions.

The music representation of Score-PMC provides an interface of functions and macros for accessing further score information. In a typical scenario, note objects are given to a constraint body via pattern matching and note parameters such as their pitches are then accessed by functions within the body. The scope of a constraint (i.e., which sets of variables it affects) is thus specified with a pattern-matching expression, optionally complemented by accessor functions.

Highly complex polyphonic CSPs can be expressed with Score-PMC. However, its constraint formalism has some limitations. The pattern-matching language is axiomatically limited to what can be expressed by a pattern. For example, a single PWConstraints pattern cannot express "any pair of consecutive variables in a sequence such that pairs do not overlap". Besides, PWConstraints users access variables only within a constraint definition: a constraint cannot be applied directly to variables. Also, a constraint definition always defines both the actual constraint (the body) and its scope (the pattern-matching expression): the actual constraint and its application are always coupled. Consequently, constraints cannot be nested, because a pattern-matching expression can only occur at the top level of a constraint definition.

4.1.4. The Search Strategy. The solver of PWConstraints performs a *chronological backtracking* (BT) search [Dechter 2003]. This algorithm first visits a variable and selects a value from its domain. The algorithm checks whether this value satisfies all constraints on this variable. In that case the algorithm proceeds to the next variable. If any constraint fails, the algorithm tries sequentially the other domain values for the current variable, and as soon as a domain value fulfils all constraints, the algorithm proceeds to the next variable. However, if no domain value satisfies all constraints, then the algorithm met a dead-end. In that case, *backtracking* occurs: the algorithm turns back to the previously visited variable and continues to try out its other domain values. When all variables are determined, the algorithm found a solution to the CSP.

BT performs a *complete* search (i.e., if there exist solutions, the algorithm finds one). BT can find one, multiple, or all solutions. However, BT has several well-known weaknesses [Dechter 2003].

—BT always detects a conflict too late: only after all variables affected by a constraint are determined, the constraint may either succeed or fail. This problem is addressed by *consistency enforcing* and *constraint propagation*. Ovans [1990] likely applied consistency enforcing the first time for musical applications (see also Sections 4.3.3 and 4.5.3 that follow).

—Failures with the same cause occur repeatedly (*thrashing*): BT does not analyze which variable causes a constraint to fail. This problem is addressed by techniques like *intelligent backtracking* (or *backjumping*). Ebcioglu [1987] pioneered backjumping for musical applications in his language BSL (Section 1).

—When after backtracking the algorithm checks variables that it already checked before, it does not remember which value combinations failed before and will check them again (*redundant work*). This problem is addressed by techniques like *backmarking*.
—Finally, BT performs decisions in an order that was fixed before the search started (*static variable ordering*, also known as static variable selection). The variable ordering of PMC is the sequential order of the variables in the music representation. A static variable ordering cannot take into account information that is only gained during the search process; static variable orderings can therefore lead to a larger search tree. This problem is addressed by *dynamic variable orderings* (see Section 4.5.3).

Nevertheless, BT has the advantage that a minimal amount of information must be kept in the search process, which is beneficial for problems with a large number of variables but with relatively few constraints (underconstrained CSP). Such "easy problems" are not uncommon for music.

Variables with universal domain allow for composite domain values. Composite variable domains could be used in principle to express and constrain a hierarchic structure, even if the music representation is purely sequential (as in PMC). However, composite domains severely impair efficiency: determining such a variable amounts to an inefficient generate-and-test. A more efficient approach uses a composite data structure containing multiple variables that can be determined independently (as in Score-PMC).

Score-PMC also performs BT. However, the system first computes an efficient order in which the notes are visited during the search process (static variable ordering). To this end, Score-PMC evaluates the rhythmic structure of the solution. Using this rhythmic structure, Score-PMC calculates a search order for its polyphonic music representation such that the search process always proceeds "from left to right", that is notes with a smaller start time value are visited first. This search strategy is the reason why Score-PMC requires that the temporal structure of any CSP to be fully determined in the problem definition.

In addition to strict constraints (which a solution must always obey), PWConstraints supports heuristic constraints which allow users to express mere preferences and to avoid overconstrained situations (where multiple constraints contradict each other). The body of a heuristic constraint returns a number instead of a Boolean value. For "better" solutions the constraint returns a higher number. In case of a conflict between heuristic constraints, the more important heuristic constraint (the one with a higher heuristic value) is preferred to hold. Heuristic constraints affect the *value ordering*, that is the order in which domain values are checked. During the search process, a domain value of the current variable with the highest heuristic value is checked first. Thus, heuristic constraints guide the search to find "better" solutions earlier, but they do not guarantee to find an optimal solution.

4.2. Situation

Situation [Rueda et al. 1998; Assayag et al. 1999] was originally conceived in collaboration between the composer Antoine Bonnet and the computer scientist Camilo Rueda as a constraint system for solving a range of harmonic CSPs. Situation was first developed as a PatchWork library, and was later extended and ported as OMSituation [Bonnet and Rueda 1999] to the PatchWork successor OpenMusic [Assayag et al. 1999]. Nevertheless, this text will refer to any version of the system simply as Situation. Situation is written in Common Lisp.

Early versions of the system supported quasi ready-made components to define harmonic CSPs. For example, instead of defining constraints from scratch, the user utilizes predefined "constraint templates", that is constraints that expect arguments from the

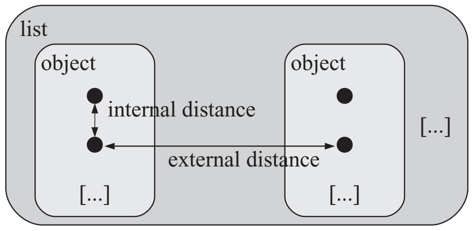

Fig. 9. The music representation of Situation consists of objects containing points and distances between these points.

user to detail their effect. Although such an approach limits the CSPs the user can define, it also simplifies the definition process for the user. Later, the system was extended to support a larger set of CSPs (e.g., additionally rhythmical CSPs). Still, the design of the system is better understood with its history in mind.

4.2.1. Music Representation. The music representation of Situation consists of a sequence of *objects* (Figure 9). An object has an internal structure: it consists of one or more *points*, and Situation defines *distances* between these points. In many CSPs, a point represents a note pitch, an object represents a chord containing multiple pitches, and a distance represents an interval between two note pitches.

Situation distinguishes between internal distances and external distances. An *internal distance* represents the distance between two points of a single object (also called vertical distance: interpreted as the distance between two neighboring note pitches in a single chord), and an *external distance* represents the distance between two points from different objects (also called horizontal distance: interpreted as the distance between matching note pitches of two consecutive chords).

The points and the distances constitute the variables in a CSP. Their domain consists of numbers.[14] Implicit constraints control the relation between the points and distances.

Users may freely interpret the meaning of points and the distances between them. For instance, points may represent pitches measured in MIDI key-numbers, or frequency values measured in *hertz*. Points may also represent note start times measured in beats, in which case distances represent temporal intervals. Situation supports this flexibility by letting the user control which implicit constraints actually hold between points and distances. For example, in case points represent MIDI key-numbers, then the relation between points and distances is governed by *addition* constrains. If, however, points represent frequencies, then *multiplication* constraints govern the relation between points and distances. Nevertheless, the music representation of Situation is particularly suited for CSPs on chord sequences: an explicit representation of pitches and the intervals between them is suitable for many such problems.

Situation also offers an alternative representation mode that consists of a plain sequence of variables. This representation mode supports variables with universal domains, like the music representation of PMC (Section 4.1.1).

4.2.2. Constraint Formalism. Situation provides predefined constraints that are not just constraint primitives like numeric operations (e.g., =, >, or +) but generalized constraints for composition purposes. For example, Situation defines a set of constraints that impose patterns on various viewpoints of the music (e.g., constraints affecting a

[14]Internally, each point is represented by its own variable, whereas a sequence of distances is represented by a single variable whose domain consists of number sequences.

(a) $[\langle \textit{range specification} \rangle_1, [\langle \textit{arguments} \rangle_1], \langle \textit{range specification} \rangle_2, [\langle \textit{arguments} \rangle_2], \ldots]$

(b) $[0, 1, 4, [\langle \textit{arguments} \rangle_1], 6\_9, [\langle \textit{arguments} \rangle_2]]$

Fig. 10. Typical syntax of a constraint application mini-language in Situation (a); example with concrete range specifications (b).

voice profile by constraining the number of consecutive upward and downward movements or the number of repeated melodic intervals). Situation complements these predefined constraints by means for user-defined constraints (Boolean functions).

Similar to PWConstraints, Situation provides a convenient and powerful mini-language to express the scope of a constraint. However, Situation defines individual mini-languages for different constraints, and it also defines many more symbols than PWConstraints' pattern-matching language. Nevertheless, for many constraints their application mini-language has a similar structure: an alternating sequence of index range specifications complemented by further arguments (Figure 10(a)).

The index range specifications control the scope of a constraint. For example, the user may apply a constraint to the first, second, and fifth object in the music representation with $\langle \textit{arguments} \rangle_1$ and to the seventh to tenth object with $\langle \textit{arguments} \rangle_2$ (Figure 10(b); indices are 0-based, index 0 points to the first object). The language used in the $\langle \textit{arguments} \rangle$ expressions is constraint-dependent and therefore not further discussed here.[15]

Unlike the constraint application mechanism of PWConstraints, the application mechanism of Situation is fully generic: a single constraint can be applied to any set of variables in the music representation. Nevertheless, due to the sequential structure of its music representation, Situation supports only score contexts defined by positional relations of objects (and their points plus distances). Consequently, it is difficult to express constraints that constrain complex score contexts as required, for example, in polyphonic CSPs. It will be very hard to define a contrapuntal constraint that requires access to score information such as the harmonic context, and the metric position of a note. By contrast, Score-PMC does support complex polyphonic CSPs.

4.2.3. Search Strategy. Compared with PWConstraints, Situation's constraint solver performs a more sophisticated search strategy. Situation applies a variation of *forward-checking* [Dechter 2003], a limited form of constraint propagation. *Minimal forward checking* [Dent and Mercer 1994] delays the reduction of the domain of variables (i.e., the forward checking) by means of *lazy evaluation* [Abelson et al. 1985] until these variables are actually visited during the search. That way, the algorithm provides the benefits of forward checking (to prune of domains, i.e., to reduce the search space) by reducing the work the algorithm has to perform (searching through possibly large domains for consistent values).

Situation's *first-found forward checking* algorithm [Rueda and Valencia 1997] adapts minimal forward checking for hierarchical domains. A variable for distances (a variable whose domain consists of number sequences, see preceding) has often a large domain. Therefore, Situation optionally organizes this domain in a hierarchical manner for efficiency. Such a hierarchically structured domain combines domain value subsets that share some property into a subtree of the domain. That way, a constraint on this common property can be propagated for all domain values in the subtree at once. Per default, this common property is the sum of the sequence elements (i.e., the sum of

[15]Besides the index range specifications, a number of other factors affect the scope of a constraint including the arguments associated with an index range specification, the number of arguments expected by a function implementing a user constraint, and the optional access of already determined objects (predecessors of current object) within a user-constraint definition.

$$\forall i \in \{1, \ldots, length(Notes) - 1\} : myConstraint(Notes_i, Notes_{i+1})$$

Fig. 11. Constraint applied to multiple variable sets by a loop with explicit variable access, as proposed by MusES with BackTalk.

object distances), because this property is constrained in many harmonic CSPs. The user can specify a different common property.

In addition to strict CSPs, Situation also supports CSPs with soft constraints. For soft constraints, a numeric preference value is specified for each constraint.

4.3. MusES with BackTalk

The combination of the constraint system BackTalk [Roy and Pachet 1997] and the music representation MusES [Pachet 1993; Pachet et al. 1996] forms a highly expressive and extendable music constraint system which has been applied to several musical problems such as automatic harmonization [Pachet and Roy 1995, 1998]. BackTalk and MusES are implemented in the SmallTalk programming language.

4.3.1. Music Representation. As shown previously, PMC and Situation group score objects in their music representation in sequences. For example, a Situation score is always a sequence of score objects where each object contains points and distances. Score-PMC defines a hierarchically nested music representation, but at each hierarchic layer score objects are arranged in a certain sequential order.

MusES uses a different approach. Every temporal object in MusES (e.g., each note) stores its start time and duration. Temporal objects can be grouped in a temporal collection. A temporal collection ensures that its contained objects are always sorted according to their start time and duration [Pachet et al. 1996]. This design allows for an easy and efficient access to, for example, all objects in a temporal collection within a certain time span. Based on this ability, MusES defines an exhaustive set of temporal relations between temporal objects.

However, the fact that temporal collections implicitly sort their contained objects restricts the expressiveness of these collections. Firstly, temporal collections cannot be nested in MusES (it would disturb the automatic temporal ordering). A CSP can nevertheless define multiple temporal collections (e.g., for representing multiple voices). Secondly, other relations (e.g., positional relations) are missing in the representation. For example, information such as the "following note" in a melody is not easily accessible from its temporal objects [Pachet 1994].

MusES is highly extendable. While the representations of PWConstraints and Situation are provided "as is" (only few alterations are possible, e.g., the number of points in a Situation object can be changed), MusES users can replace the built-in MusES objects with their own objects. MusES consists of a set of SmallTalk classes and users can extend the representation incrementally by defining subclasses of existing MusES classes.

4.3.2. Constraint Application. Constrained variables are accessible directly in BackTalk, whereas variables are only accessible indirectly in PWConstraints and Situation via the constraint application mechanisms of these systems (e.g., pattern matching). As a result, in BackTalk common control structures can be used for applying constraints to variables. For example, BackTalk users can use a loop for applying a melodic constraint to every pair of consecutive notes. The running index i of the loop is used to access consecutive note pairs in a melody (refer to Roy and Pachet [1997]). Figure 11 shows such a loop in first-order logic notation.

For complex constraints, however, it becomes tedious to explicitly access all the variables involved. Even for this simple example we need to perform arithmetic operations on indices. To present a more complex example, a voice-leading constraint like "avoid open parallel fifths and octaves" involves pairs of consecutive notes in multiple voices and at the same time the intervals between simultaneous notes. Expressing these score contexts by nested loops would result in a complex program. The constraint application mechanisms of PWConstraints and Situation shown before are more convenient for complex cases. For example, Score-PMC provides suitable applicators for voice-leading constraints.

Conversely, the constraint application mechanisms of PWConstraints and Situation are limited in their generality. By contrast, control structures such as nested loops are fully generic.

4.3.3. Search Strategy. BackTalk constitutes a system for solving CSPs over finite domains of arbitrary SmallTalk objects. Because the efficiency of search strategies is highly problem-dependent, BackTalk implements a library of constraint solving algorithms from which users can choose the algorithm that best suits their purposes.

In a first stage, the solver reduces the domains of the variables and that way reduces the search space (often drastically!), without removing any solutions. Several generic *arc-consistency* enforcing algorithms [Dechter 2003] are provided for this purpose (AC-3, AC-4, ..., AC-7).

In a second stage, the solver performs the actual search where the solver makes decisions that can fail and need to be taken back (backtracking). Again, several algorithms are provided for this stage including chronological backtracking, forward-checking, backjumping, and backmarking.

MusES substantially improves the efficiency of the search process by also representing analytical information explicitly. For example, MusES features objects that represent intervals, scales, and chords. Besides being beneficial for the definition of CSPs, such a representation scheme provides the consistency enforcing algorithm with additional information which is useful for reducing the search space (e.g., only certain pitch interval combinations form a chord) [Pachet and Roy 1995].

4.4. OMClouds

OMClouds [Truchet et al. 2001; Truchet and Codognet 2004] constitutes a music constraint system that is particularly easy to use. It extends the composition system OpenMusic, and is implemented in Common Lisp.

4.4.1. Search Strategy. The search strategy applied by OMClouds differs clearly from the strategies of the systems discussed earlier. Whereas most music constraint systems support a complete search, OMClouds is based solely on a heuristic search strategy that iteratively improves an initial random solution to a CSP. More specifically, OMClouds' constraint solver performs a local improvement search strategy [Codognet and Diaz 2001; Codognet et al. 2002]. This strategy quickly finds an approximated solution to a constraint problem that fulfils many or most of the constraints imposed.

Internally, a constraint is implemented by a cost function, which returns a numeric value indicating how much its constraint is violated. In a nutshell, the algorithm starts by determining all variables of the CSP to some random value from their domain. Then, the algorithm computes the accumulated cost of all cost functions for every variable to find the variable whose current value conforms least a solution to the CSP. This variable is updated for the next iteration. OMClouds extends this basic idea by a *taboo search* method [Glover and Laguna 1997]: unsuccessful variable values are not tried again for a while, in order to avoid local minima in the search space.

On the user level, however, a CSP is expressed by strict constraints (e.g., $X = Y$). Only internally, these constraints are translated into cost functions (e.g., $X = Y$ is transformed into $|X - Y|$).

The heuristic approach of OMClouds makes the definition of CSPs easier because no overconstrained situation can occur. In a system that only supports strict constraints, the CSP must explicitly handle special cases in which some constraint can be neglected. In OMClouds, constraint definitions can be less carefully formulated: the heuristic search strategy does not reject a solution because some constraints are not fulfilled for certain variables.

Adaptive search does not execute a complete search. If the system is asked multiple times for a solution to the same problem it usually comes up with different solutions, but the system cannot output all solutions. Moreover, the system does not guarantee to find an optimal solution even if such a solution exists. For example, constraints like "all elements of a list are pairwise distinct" are hard to fulfil and the algorithm may never find an exact solution to a CSP with this constraint (e.g., OMClouds may never find a true all-interval series).

4.4.2. Music Representation. In OMClouds, variable domains can consist of any values in principle. Nevertheless, some of its constraints support only integers due to the implementation of their cost function. In addition, all variables share the same domain. This restriction simplifies the definition of CSPs, but also limits what CSPs can be defined.

Similar to PMC (see Section 4.1.1), OMClouds' music representation consists of a flat list of variables.[16] This straightforward representation format was chosen probably in order to avoid any bias caused by a more expressive music representation. A purely sequential representation can express several musical structures. However, as has been explained before, such a structure is poorly suited for more complex musical CSPs, in particular polyphonic problems.

4.4.3. Constraint Formalism. OMClouds supports easy-to-use means for applying constraints to the variables of the CSP. All constraints of a CSP are merged by an implicit conjunction into quasi a single constraint, which is applied to all variables in a homogeneous way. For example, constraints can be applied to every single variable or to every pair of two consecutive variables. However, all constraints share the same constraint scope. Also, OMClouds does not provide any means to apply a constraint only to a specific subset of variables (e.g., only on the first and third variable), in contrast to the constraint application mechanisms of other systems like PWConstraints and Situation. Again, this design results in a system that is particularly easy to use, but which is also very limited in what CSPs can be defined with it.

4.4.4. User Interface. OMClouds is not only easy to use because it supports only a severely limited range of musical CSPs. In addition, CSPs are defined fully with a visual programming language in OMClouds. Using this visual language, users do not need to memorize lots of keywords, functions, and so forth, as these can be selected in a menu. The language uses a simple and uniform syntax: boxes are connected by patch chords.

4.5. Strasheela

The design of Strasheela [Anders 2007] aims for a highly generic music constraint system. Strasheela supports various CSPs that are very hard or even impossible to realize

[16]In fact, OMClouds features three representation cases: the solution constitutes either a sequence of variables, a cycle of variables, or a permutation of variables. In all cases the solution is represented by a flat list. The second case, however, slightly changes the constraint application mechanism while the third case enforces an additional constraint. In the second case—the cycle—all elements (including first and last) have a predecessor and successor. In the third case—the permutation—the number of variables in the music representation equals the number of domain values specified and every domain value appears only once in a solution. That way, OMClouds can avoid an `all-different` constraint on the solution, which is hard to fulfil for its heuristic search.

in other systems, such as problems in which the rhythmical, harmonic, contrapuntal, and formal structure are constrained at the same time. Nevertheless, like previous systems, Strasheela seeks to find a balance between generality and efficiency in order to be useful in practice.

As Strasheela is a relatively recent system, its design benefited from examining the design of earlier systems. Strasheela is implemented in the programming language Oz, a language with built-in support for CP.

4.5.1. Music Representation. An important design goal for Strasheela's music representation is that users must be able to control which information is stored in the music representation. Instead of providing a single music representation template, as in Score-PMC or Situation, Strasheela provides a set of building blocks for various musical concepts. Users select those building blocks required for their particular CSPs, and construct the representation by using these. A wide range of music theory concepts is already modeled as building blocks including elements such as notes, or rests, analytical concepts such as intervals, scales, chords, or meter, grouping concepts such as containers that arrange their content sequentially or in parallel in time, as well as concepts for organizing musical form such as motifs.

The set of these building blocks can be freely extended. The building blocks are instances of classes, and users can extend existing classes by inheritance, a facility also provided in MusES.

Strasheela's music representation design combines benefits of existing systems that previously excluded each other. Pachet [1994] compares the design of MusES with the design of the music representation SmOKe [Pope 1992]. SmOKe supports arbitrarily nested event lists and the events in an event list can be arranged in any order. However, a SmOKe event does not provide access to its start time: this information is only known by its event list. By contrast, any information available on a MusES score object can be accessed from the object directly. However, MusES only supports a single-level hierarchic nesting.

Strasheela supports arbitrary hierarchic nesting of score objects, and in addition any information available on a score object can be accessed from the object directly. These features have been realized by two design principles: bidirectional links and constraint propagation. Strasheela allows for hierarchic nesting of containers. However, the hierarchically nested score objects in Strasheela are bidirectionally linked: every container provides access to its contained items and vice versa. Furthermore, parameters exchange knowledge about their values (which are constrained variables), due to constraint propagation. For instance, information about temporal parameters such as start-time, duration, and end-time is propagated between temporal containers and events.

Based on this design, Strasheela supports MusES' exhaustive set of temporal relations, as well as any other score context. Their implementation can be less efficient in Strasheela than in MusES (they may potentially traverse the full score), but it is still sufficiently fast, because objects must be accessed only once (after a constraint is applied, the search process does not require accessing objects again).

Strasheela also supports constraining the hierarchic structure of the score, if only in a very limited way. The duration of temporal score objects can be constrained to 0, and such objects are considered nonexisting.[17] Whole sections of the score can be "removed" with this approach.

4.5.2. Constraint Application. The design of Strasheela combines the convenience of special constraint application mechanisms found in PWConstraints and Situation on the

[17]For efficiency, nonexisting score objects can only occur at the end of a container in order to avoid symmetries.

$$mapSimNotePairs(collect(MyScore, test{:}\ isNote), isConsonant)$$

Fig. 12. Constraining pairs of simultaneous notes with the higher-order constraint applicator *mapSimNotePairs*: simultaneous note pairs are constrained to be consonant.

one hand, with the generality of using arbitrary control structures proposed by BackTalk on the other hand. Strasheela advocates constraint applicators based on higher-order programming [Anders and Miranda 2011]. A constraint is a procedure and a constraint applicator is a higher-order procedure expecting a music representation and a constraint as arguments. This approach is fully generic, because internally a constraint applicator can conduct any traversal of the score and thus define any constraint scope. At the same time, this approach is convenient because these details are encapsulated in a constraint applicator.

Figure 12 shows an example that applies a constraint to all pairs of simultaneous notes. The higher-order constraint applicator *mapSimNotePairs* expects a list of score objects, in the example all notes in *MyScore*. For each note, *mapSimNotePairs* internally accesses its simultaneous notes and applies the constraint *isConsonant* to each such note pair. Note that this constraint applicator can be used even if the rhythmical structure is undetermined in the CSP definition. *mapSimNotePairs* delays the constraint application until the necessary rhythmical information is found during the search process.

Strasheela predefines constraint applicators for a wide range of use cases. The system also provides higher-order variants of the application mechanisms of PWConstraints and Situation. Most importantly, however, users can define new constraint applicators if necessary.

4.5.3. Search Strategy. Strasheela uses the CP model of its implementation language Oz [Schulte 2002]. This model combines constraint propagation with search.

Constraint propagation performs logical deductions that reduce variable domains without removing any solution, much like the arc-consistency enforcing algorithms employed by BackTalk. In contrast to the generic arc-consistency algorithms, however, constraint propagation algorithms are highly optimized for a specific domain and constraint and are thus more efficient. BackTalk uses an arc-consistency algorithm only once for prefiltering the domains before the actual search starts. The improved efficiency of constraint propagation allows for a different approach where propagation and the actual search take turns: before every single decision made by the search, constraint propagation runs and reduces variable domains.

The improved efficiency of constraint propagation has a trade-off: the variable domains are restricted to specific "types". Strasheela presently supports variables with nonnegative integers and sets of integers. Further domains can be defined; available domains for propagators are real intervals [Díaz et al. 2005], graphs [Dooms et al. 2005], and maps (domain of functions) [Deville et al. 2005],[18] but doing so requires low-level programming in C++. Systems like PMC or BackTalk, on the other hand, allow for universal domains containing arbitrary values.

Whereas BackTalk provides a range of search algorithms to choose from, the Oz CP model goes even further: this model provides abstractions that make the search process programmable at a high level [Schulte 2002]. The *variable* and *value ordering* (the order in which variables and their domain values are considered during the search process) has a great impact on the size of the search tree and thus on the efficiency of the search [van Beek 2006]. Good ordering heuristics result in a search tree with

[18]The references given in this sentence point to Oz-related literature that is of immediate relevance for Strasheela. Schulte [2002, p. 1] provides more general citations on variable domains.

Table I. System Comparison: Search Strategy Algorithms Supported by Solver

	Search Strategy
PMC	chronological backtracking[a]
Score-PMC	chronological backtracking with static variable ordering optimised for polyphonic CSPs
Situation	forward checking with lazy evaluation optimised for hierarchic domains
BackTalk	multiple algorithms for pre-filtering domains, and for search tree traversal
OMClouds	heuristic local improvement algorithm[b]
Oz	concurrent domain filtering (constraint propagation), user definable dynamic variable & value orderings, user definable search tree traversals

[a]Forward checking is possible in principle, but only for explicitly defined forward-checking constraints.
[b]An approximated solution is found quickly, but a strict solution is possibly never found.

relatively few nodes. In the constraint model of Oz, variable and value orderings can be defined freely by users. Oz supports dynamic orderings: the decision which variable is visited next and how its domain is reduced by the search is only made when this decision is due.

Anders [2007] shows that musical CSPs can greatly benefit from a suitable problem-dependent ordering. For example, the author compares the efficiency of different variable orderings for a florid counterpoint CSP where both the rhythmical structure and the pitch structure are constrained. In one variable ordering, first all rhythmical parameters are determined and then the pitches. The second ordering implements a dynamic variant of the variable ordering of Score-PMC: variables are visited "from left to right" in the order of the start time of their score objects. For the florid counterpoint CSP in question, the second ordering is three orders of magnitude faster than the first ordering. Nevertheless, different CSPs may require different variable orderings. For example, harmonic counterpoint CSPs are often solved more efficiently when the harmonic structure is fully determined before addressing the actual note pitches. Strasheela provides special score variable orderings which cover a wide range of musical CSPs.

5. COMPARISON OF GENERIC MUSIC CONSTRAINT SYSTEMS

The preceding section introduced a number of generic music constraint systems one after the other. This section summarizes this survey and compares these systems according to a number of orthogonal criteria.

The comparison first looks at the search strategies implemented by these systems, because later comparison sections depend on it (Section 5.1). The next two sections compare the expressivity of these systems: what musical CSPs are supported? Instead of listing a potentially infinite list of music theories supported, abstract criteria are studied such as which score information is supported by a system and what subset of this information can be constrained (the music representation format, Section 5.2), and how are constraints applied (Section 5.3). Finally, Section 5.4 compares how easy the systems are to use.

5.1. Search Strategy

Music constraint systems employ very different constraint solvers. Table I lists these algorithms. The table indicates a tendency towards more sophisticated approaches in more recent systems.

The development of two aspects stands out. On the one hand, domain filtering algorithms become gradually more sophisticated. PWConstraints uses backtracking without any domain filtering, Situation makes use of forward checking and optimizes this technique with lazy evaluation, BackTalk provides a range of different algorithms for prefiltering domains, and Oz (and thus Strasheela) supports constraint propagation

Table II. System Comparison: Does the Solver Find All Solutions? Does the Solver Find a Best Solution (according to some user-defined criterion) without Traversing the Full Search Tree?

	All solutions?	Best solutions?
PMC	yes	no
Score-PMC	yes	no
Situation	yes	yes[c]
BackTalk	yes	no
OMClouds	no	no
Oz	yes	yes

[c]Requires knowledge of internal detail only documented in the source code.

Table III. System Comparison: Are Hard and Soft Constraints Supported?

	Hard constraints?	Soft constraints?
PMC	yes	yes: modelled with value ordering heuristics
Score-PMC	yes	yes: modelled with value ordering heuristics
Situation	yes	yes
BackTalk	yes	yes: modelled with value ordering heuristics
OMClouds	no	yes
Oz	yes	yes: modelled with reified constraints

where filtering algorithms are optimized for specific domains and constraints. Propagation algorithms greatly improve the efficiency of the search and are today used by virtually all major CP systems (e.g., SICStus Prolog [Carlsson et al. 2009], ECLiPSe [Cheadle et al. 2003], GNU Prolog [Diaz and Codognet 2001], Gecode [Schulte et al. 2010]).

On the other hand, the actual search process becomes more flexible in later systems. PWConstraints and Situation support a single search algorithm with a static variable ordering, BackTalk supports a wide range of algorithms to choose from, and in Oz the search process is programmable on a high level of abstraction. A special case, not directly comparable to the algorithms of the other systems, is the heuristic local improvement algorithm of OMClouds.

Different search approaches lead to different features of the solver. For example, all systems perform a full search, except OMClouds. Systems that perform a full search can also search for all solutions (Table II).

Sometimes it is interesting to find a best solution according to some user-specified criterion (e.g., a function comparing two solutions). However, collecting first all solutions in order to find the best is computationally expensive. Situation and Oz (Strasheela) support a best-solution search (branch-and-bound algorithm [Schulte 2002]), which during the search process always constrains the next solution to be better than the previous. Together with constraint propagation this approach greatly improves the efficiency of finding a best solution as it does not need to find all solutions first.

Modeling music theories requires hard constraints (which are always fulfilled) and soft constraints (rules which might be broken, e.g., in an otherwise overconstrained situation). For example, hard constraints are required for modeling an all-interval series or for stating which pitch classes belong to a chord. Nevertheless, many rules in music theory are only guidelines and are better implemented by soft constraints. Table III shows that most systems provide hard constraints as a primitive, and soft constraints are implemented by some special technique. Again, the purely heuristic OMClouds is the exception: it does not support hard constraints.

It would have been interesting to compare the efficiency of the presented generic music constraint systems in a benchmark. However, such a performance comparison would meet severe difficulties. Firstly, there does not exist any set of established and well-defined benchmark problems for musical CSPs. It would therefore be necessary to first define a set of such problems. However, each of these systems supports a different range of musical CSPs, as detailed in the article. For a benchmark it would be necessary to choose CSPs that are supported by each presented system, which is only the case for CSPs where the music representation is a flat sequence of variables (e.g., the all-interval series).[19] Measuring the performance for such simple CSPs would not be very interesting nor would this comparison be fair (e.g., more simple systems could be advantaged, because the more complex music representation and search of more expressive systems adds some overhead). Besides, measuring the performance of a heuristic system such as OMClouds is problematic even for simple CSPs such as the all-interval series. While the system arrives at an approximated solution very quickly, it potentially never finds an exact all-interval series. It is difficult to formally decide at what time the solution is good enough in order to terminate a performance measurement (as OMClouds potentially never finds an exact solution, it does not stop searching by itself). Further, some systems allow for various optimizations (e.g., BackTalk supports various search strategies, and one of the design goals of Strasheela was even to make the search programmable by users). What optimizations would be allowed for the benchmark? If arbitrary optimizations would be allowed, then for fairness the benchmarked CSPs are best implemented by someone who intimately knows the respective system (e.g., its authors). However, performance measurements have only been published for few systems (e.g., Pachet and Roy [1995] and Anders [2007], see also Section 6), and these publications only outline the CSP whose performance has been measured: they do not provide the detail required to reproduce this CSP exactly in other systems for a fair performance comparison.

5.2. Music Representation

The music representation format of a system greatly influences which musical CSPs it can define, as has been shown before for the PWConstraint system (Section 4.1.2). This section compares the representations of generic music constraint systems by looking at three criteria: which information is explicitly represented, which score contexts can be accessed from this explicit information, and which subset of the explicit information is constrainable.

5.2.1. Explicitly Represented Information. All systems provide some composite data structure that expresses the explicitly represented score information. For example, musical concepts such as individual parameters (e.g., durations, pitches), elementary score objects (e.g., notes, rests), or compound score objects (e.g., chords, motifs) are represented by objects[20] in this data structure. In all systems discussed here, solutions are expressed directly in this format instead of being somehow encoded. Consequently, each system is limited to those CSPs for which solutions can be expressed by the music representation of this system.

Table IV compares the data structure formats. Many systems are highly extendable. Therefore, instead of looking at simple notions such as “which note parameters are supported by system X?” this table compares structural properties of these data structures. The table reports whether or not representations support objects to have

[19]In principle, PMC and OMClouds allow for composite data as variable domains, which could be used to express a hierarchic music representation (e.g., a homophonic chord sequence). However, the search then partly amounts to an inefficient generate-and-test, because values nested in a composite domain value are not tested individually.

[20]This term is used in this section in a general meaning and does not necessarily denote objects in an object-oriented programming sense.

Table IV. Comparison of the Music Representation Format of Different Systems: How Can Information be Expressed Explicitly? The Information Represented by Variables is Compared Separately Below

	Attributes	Score Topology	Type
PMC	none	flat sequence	uniform
Score-PMC	predefined	tree (fixed nesting)	predefined
Situation	user defined	flat sequence	uniform
MusES with BackTalk	user definable	one or more temporal collections[d]	user definable, hierarchical
OMClouds	none	flat sequence	uniform
Strasheela	user definable	user defined[e]	user definable, hierarchical

[d]Each container (`TemporalCollection`) stores temporal objects (e.g., notes, chords) sorted by their start time and duration, i.e. no explicit non-temporal order is represented.
[e]Multiple topologies supported (e.g., nested event lists, tree of generic temporal containers, acyclic graph of arbitrary containers).

attributes, a type, and how objects are arranged (the score topology). For example, the music representation of PMC is a flat list of values. By contrast, the representation of Score-PMC consists of various score object types such as notes, chords, parts, etc., for which a fixed set of features is defined, and which are arranged hierarchically in a predefined way. Note that the table distinguishes between “user defined” and “user definable”: in the latter case, users can either use predefined settings or define their own.

Hierarchic nesting makes it possible to group score objects that somehow belong together (e.g., which notes form a motif, chord, or voice, which sections form a movement, and so forth). CSPs which make use of such information require that this information is represented. The attributes and the score topology both contribute to the hierarchic nesting of a score, but both are listed separately, to make the table more easy to read. Note that the score topology of many systems is a flat sequence, and thus grouping cannot be expressed by the topology.[21] In such systems it is difficult to model music theories that rely on grouping information such as counterpoint (which relies on relations between different parts), or musical form. MusES is unique in that it always temporally sorts the elements of a collection, and that a CSP can consist of multiple composite scores. Only Score-PMC and Strasheela allow for hierarchic nesting, and Strasheela is the only system that supports an arbitrarily nested score.

When modeling theories it can be important to distinguish between different score object types (e.g., notes versus rests, or chords versus scales). Type information can be expressed either by a data type or simply by a type attribute, but objects with different types can also differ in their attribute set. Several systems only allow for a single type in a CSP (in Situation, users can control the set of object attributes, but all objects uniformly share the same attribute set). MusES and Strasheela define a type hierarchy by defining a class hierarchy in the object-oriented programming sense for score objects: inheritance simplifies the definition of new types that share similarities with existing types.

5.2.2. Accessible Score Contexts. This article introduced the term *score context* to denote sets of related score objects. Music theories often constrain complex score contexts, as has been shown before (Section 4.1.2).

A score context can be constrained directly, but it is often also used for accessing derived information. For example, most representations store only note pitches explicitly, but melodic intervals between two pitches can be derived if consecutive note pairs in a voice are accessible.

[21]In principle, it is also possible to express grouping by special attributes (e.g., a note parameter that indicates its part), but such an approach makes it hard to add further information on higher-level objects (e.g., the part itself) or to introduce further hierarchic levels.

Table V. System Comparison: Which Score Contexts (i.e., sets of related score objects) are Accessible in the Music Representation?

	Accessible Score Contexts
PMC	positionally related values (only via constraint application mechanism)
Score-PMC	a fixed set of predefined contexts (melodic, harmonic and metric context)
Situation	positionally related objects (only via constraint application mechanism), and their attributes[f]
MusES with BackTalk	any set of temporally related score objects, and their attributes
OMClouds	positionally related values (only via constraint application mechanism)
Strasheela	any set of score objects, and their attributes

[f]The attributes of Situation objects include explicitly represented distances between attributes (see Section 4.2.1).

Table VI. System Comparison: Which Information in the Music Representation can be Expressed by Variables and Is Thus Constrainable?

	Variable Occurrence	Variable Domain
PMC	element in sequence	universal
Score-PMC	note pitch (key-number)	integer
Situation	attribute of object in sequence	number[g]
MusES with BackTalk	anywhere	universal
OMClouds	element in sequence	integer
Strasheela	any value suiting the variable's domain[h]	specific domains[i]

[g]A domain consists, for example, of integers, floats, or ratios (possibly mixed). Alternatively, Situation supports variables of universal domains. However, choosing this option simplifies the music representation into a flat list and disables constraint propagation.
[h]For example, constrained variables can occur as attributes of score objects or locally in constraints.
[i]In principle, universal domains are possible. However, constraint propagation is only supported for specific domains (e.g., finite domain integers, and finite sets). The set of supported domains can be extended (requires low-level programming in C++).

Table V lists the accessible score contexts of each system. Naturally, the accessible score contexts depend on the music representation format just discussed: if the score topology is a flat sequence, then only positionally related objects are accessible (e.g., pairs of consecutive objects). However, some systems further restrict the accessible score contexts. In several systems score contexts are only accessible via the constraint application mechanism, and the mechanisms of PWConstraints and OMClouds are not fully generic (Section 4.1.3 and 4.4.3). Also, note that for efficiency reasons only temporally related contexts are supported by MusES. For example, the melodic context of consecutive notes is difficult to access from a temporal collection where notes can also occur simultaneously (Section 4.3.1). Only Strasheela supports accessing arbitrary score contexts.

5.2.3. Constrainable Information. So far, we discussed the nature of the information that can be expressed by the representation scheme of the different systems. Table VI details where variables can occur in these representations, that is which pieces of information can be constrained.

Obviously, variables can only occur within the boundaries of the music representation format of each system. For instance, the music representation of PMC is a flat sequence of values, and these values can be variables. However, not every score object can always be substituted by a variable. For example, only note pitches but no rhythmic parameters can be variables in Score-PMC for efficiency reasons: Score-PMC requires a determined rhythmical structure for computing its static variable ordering before the search starts (Section 4.1.4).

Systems differ not only in where variables are supported, but also in the supported variable domains. The supported domains depend on the search strategy (Section 5.1).

Table VII. System Comparison: How are Constraints Applied to the Score?

	Constraint Application Mechanism
PMC	pattern matching language for sequence
Score-PMC	extended pattern matching language for hierarchic score
Situation	multiple index-based mini-languages
MusES with BackTalk	arbitrary control structures (e.g., loops)
OMClouds	uniform for-all constraint application
Strasheela	arbitrary control structures, convenient higher-order procedures

Systems that use chronological backtracking (PMC and Score-PMC) can in principle support a universal domain, which consists of arbitrary values. In Score-PMC, however, note pitches must have an integer domain (MIDI note numbers). The solution integer pitches are automatically inserted as 12-tone equal temperament pitches in the score editor of its host system, and microtonal music is therefore not supported by Score-PMC.[22] BackTalk uses generic arc-consistency enforcing algorithms and therefore also supports variables with universal domain. Systems that use domain-specific propagation algorithms for efficiency (Situation and Strasheela) support only those variable domains for which propagation has been implemented.

Note that some information is not constrainable in any of these systems. In particular, the score topology is fixed before the search, and the type of score objects cannot be constrained.

5.3. Constraint Application

Existing systems differ widely in the means provided for applying constraints to the variables in the music representation (Table VII). PWConstraints, Situation, and OMClouds provide special mini-languages to this end. These are convenient for many cases, but less suitable for others (Section 4.1.3 and 4.2.2). The application mechanism of OMClouds is particularly limiting: basically, all variables are constrained in a homogeneous way (Section 4.4.3).

By contrast, MusES with BackTalk and Strasheela allow for directly accessing the variables in the music representation. Arbitrary control structures can be used in these systems for applying constraints to these variables. While MusES with BackTalk uses common control structures such as loops, Strasheela also provides a collection of convenient higher-order procedures for various purposes (e.g., some procedures model mini-languages of PWConstraints and Situation).

5.4. Usability

There are two important design goals for a music constraint system. On the one hand, designers aim at making their systems suitable for a wide range of composition problems. The history of Situation exemplifies this goal: in the beginning the system supported only harmonic CSPs, and it was later generalized to support rhythmic CSPs as well. On the other hand, systems should of course be easy to use.

These two design goals are contradictory to a certain degree. Users of music constraint systems include composers and music theorists (often with little programming experience) besides computer scientists or mathematicians. When defining musical CSPs, these users always write computer programs. Making musical constraint systems easy to use can therefore mean making computer programming more easy for nonprogrammers.

[22]The only system with explicit support for microtonal music is Strasheela: Strasheela's export formats and predefined constraints take microtonal pitches in various representations into account, and there are many microtonal Strasheela examples available.

Table VIII. System Comparison: Programming Language Syntax for CSP Definitions and Programming Environment

	Syntax	Environment
PMC	visual & textual	CAC system PatchWork, now PWGL
Score-PMC	visual & textual	CAC system PatchWork, now PWGL
Situation	visual & textual	CAC system PatchWork, now OpenMusic
MusES with BackTalk	textual	SmallTalk programming environment
OMClouds	purely visual	CAC system OpenMusic
Strasheela	textual	Oz programming environment

Visual programming languages have been very successful in making programming more accessible to nonprogrammers in the field of computer music in general. Nevertheless, while visual programming languages simplify the programming syntax, they cannot substitute the computer science concepts on which the generality of music constraint systems is founded. More generic systems tend to apply computer science concepts which are more advanced for nonprogrammers. For example, Strasheela's constraint application mechanism is more generic than PMC's mechanism, but its underlying concept (higher-order programming) is also more abstract than the pattern-matching mechanism of PMC. While any computer science concept can be integrated into a visual language in principle (e.g., OpenMusic supports higher-order programming), users must understand these concepts in order to employ them successfully. Visual programming is thus no magic bullet for making programming more easy: computer science concepts per se are not more easy to comprehend in a visual than a textual language.

Existing systems typically lean towards only one of these two design goals (Table VIII). OMClouds is the most easy to use among the existing systems. It provides a purely visual programming language. Further, due to its purely heuristic search CSPs are more easy to define: users cannot define overconstrained problems. At the same time, OMClouds is also most restricted in the range of CSPs supported, as has been shown earlier. MusES with BackTalk and Strasheela lean towards the other design goal. These systems are particularly expressive, as has been shown before. However, these systems have been designed for experienced programmers. PWConstraints and Situation are situated more in the middle in this respect. A superficial indication is their syntax, which combines visual and textual programming. Whereas MusES with BackTalk and Strasheela are real programming systems, PWConstraints and Situation quasi provide CSP templates where the user can change many settings but never change the overall structure of the program. This approach makes these systems less expressive than MusES with BackTalk or Strasheela, but at the same time it simplifies their use.

Besides the ease of the CSP definition, other factors also contribute to the usability. An important aspect is the programming environment into which the music constraint system is embedded. For example, many systems are integrated in Computer-Aided composition (CAC) systems such as PWGL or OpenMusic, which also provide libraries for other algorithmic composition techniques, and where expressive score editors are available (Table VIII). By contrast, MusES with BackTalk and Strasheela are build directly on top of general-purpose programming languages.

Also seemingly minor points can have a decided impact on the usability. For example, in case of an overconstrained problem with no solution, systems supporting propagation (e.g., Strasheela) often immediately report that there is no solution, while backtracking-based systems may instead search for a very long time. Debugging CSPs is difficult in every system, but some systems (e.g., PWConstraints) can provide feedback on which constraints caused failure.

6. CONCLUSION

Constraint Programming (CP) is a highly suitable paradigm for computationally modeling music theories and composition. The CP paradigm has been used for several decades in this field, and many music theory subdisciplines have been addressed including counterpoint, harmony, rhythm, and instrumentation (Section 3).

Nevertheless, many complex aspects of music theory and composition still await constraint-based modeling. In particular, more research on the modeling of instrumentation and orchestration is required. Other neglected fields include harmonic counterpoint, and the modeling of melody and musical form.

While music theories are often modeled "from scratch" using CP, generic music constraint systems have been developed for more than a decade (Sections 4 and 5). These systems predefine important building blocks shared by many musical Constraint Satisfaction Problems (CSP) and that way greatly simplify their definition. In doing so they make music CP accessible for a larger audience. Each of these systems has a similar structure: it implements a music representation, defines mechanisms to define and apply constraints to variables in the music representation, and provides a constraint solver. However, the systems differ in the actual design of these components.

As a result, generic music constraint systems differ greatly in the range of music theories they support. For example, few systems support contrapuntal CSPs, as such problems require a more expressive music representation. The two systems MusES with BackTalk (Section 4.3) and Strasheela (Section 4.5) are particularly expressive: they are suitable for highly complex music theories such as a fully-fledged theory of harmony or counterpoint. Important for this expressivity is that these systems effectively provide software libraries: users have great freedom in arranging the components provided, and these systems are also highly extendable. However, these systems are designed for experienced programmers. OMClouds (Section 4.4) is the exact opposite in terms of expressivity and usability. The range of music theories supported is the smallest among the five systems reviewed, while at the same time OMClouds is the most easy-to-use system. Its visual language and its purely heuristic search, which prevents overconstrained problems, makes OMClouds best suited for users without prior programming experience. The two systems PWConstraints (Section 4.1) and Situation (Section 4.2) are situated between these two extremes. These systems define CSP templates that restrict the range of theories supported. Nevertheless, the template of a system like PWConstraints' subsystem Score-PMC still allows for a wide range of complex contrapuntal CSPs.

For the practical use, an important concern is the speed at which musical CSPs can be solved. Simple problems (e.g., an all-interval series or first-species Fuxian counterpoint) are solved in a couple of milliseconds [Anders 2007]. More complex problems such as the harmonization of a melody or two-voice florid counterpoint take a few seconds [Pachet and Roy 1995; Anders 2007]. However, musical CSPs can be extremely complex, and finding a solution for a complex problem can take a long time. An efficient search strategy is highly problem-dependent, and the more generic systems therefore provide extensive control over the search process (Sections 5.1). However, such optimizations are again best done by an experienced programmer.

In general, systems that support a greater range of music theories and more complex theories tend to be harder to use, because they require more programming experience from their users. Yet, potential users such as composers and music theorists—who could greatly contribute to this field—are often not trained as programmers. For future research, it would be interesting to make the expressive power available in music CP today more accessible for this audience.

Some recent research developments on CP in general would be highly useful for music CP, but have not been adopted in existing system so far. For example, no system supports that the hierarchic structure of the score can be constrained freely, but such a feature would be highly useful for modeling musical form. Variables with graph domain [Dooms et al. 2005] might be a suitable approach for this. Soft constraints are particularly important for music. Existing systems that are based on hard constraints model soft constraints using various techniques (Section 5.1). Support for true soft constraints [Bistarelli 2004] that can be combined with hard constraints would be beneficial. Finally, music CP is computationally expensive because the search space can be huge. So far, the performance of systems was quasi automatically improved over time because each new generation of CPUs ran on a higher clock rate than the previous generation. This development has meanwhile slowed down. Future constraint systems will improve their performance by using multiple processors in parallel with parallel search [Schulte 2002].

ACKNOWLEDGMENTS

We would like to thank four anonymous reviewers for their helpful feedback.

REFERENCES

Abelson, H., Sussman, G. J., and Sussman, J. 1985. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

Anders, T. 2000. Arno: Constraints programming in common music. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco.

Anders, T. 2007. Composing music by composing rules: Design and usage of a generic music constraint system. Ph.D. thesis, School of Music & Sonic Arts, Queen's University Belfast.

Anders, T. 2009. A model of musical motifs. In *Proceedings of the Conference on Mathematics and Computation in Music (MCM'07)*. T. Klouche and T. Noll, Eds., CCIS 37. Springer, Berlin.

Anders, T. and Miranda, E. R. 2009a. A computational model that generalises schoenberg's guidelines for favourable chord progressions. In *Proceedings of the 6th Sound and Music Computing Conference*. F. Gouyon, A. Barbosa, and X. Serra, Eds. http://smc2009.smcnetwork.org/programme/pdfs/151.pdf (accessed 1/10).

Anders, T. and Miranda, E. R. 2009b. Interfacing manual and machine composition. *Contemp. Music Rev. 28,* 2, 133–147.

Anders, T. and Miranda, E. R. 2011. Constraint application with higher-order programming for modeling music theories. *Comput. Music J.*

Apt, K. R. 2003. *Principles of Constraint Programming*. Cambridge University Press, Cambridge.

Assayag, G. 1998. Computer assisted composition today. In *Proceedings of the 1st Symposium on Music and Computers*. http://www.ircam.fr/equipes/repmus/RMPapers/Corfou98/ (accessed 1/10).

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. 1999. Computer assisted composition at IRCAM: From patchwork to openmusic. *Comput. Music J. 23,* 3, 59–72.

Bartak, R. 1998. On-Line guide to constraints programming. http://kti.mff.cuni.cz/~bartak/constraints/ (accessed 1/09).

Berg, P. 2009. Composing sound structures with rules. *Contemp. Music Rev. 28,* 1, 75–87.

Berlioz, H. and Strauss, R. 1904. *Instrumentationslehre*. Edition Peters, Leipzig.

Bernard, J. 1994. Voice leading as a spatial function in the music of Ligeti. *Music Anal. 13,* 2, 227–253.

Bistarelli, S. 2004. *Semirings for Soft Constraint Solving and Programming*. Springer, Berlin.

Bonnet, A. and Rueda, C. 1999. *OpenMusic. Situation.* version 3, 3rd Ed. IRCAM, Paris.

Boulanger, R. 2000. *The Csound Book. Perspectives in Software Synthesis, Sound Desing, Signal Processing, and Programming*. MIT Press, Cambridge, MA.

Boulez, P. and Nattiez, J.-J. 1990. *Orientations: Collected Writings*. (Translated by Martin Cooper.) Harvard University Press, Cambridge, MA.

Bresson, J., Agon, C., and Assayag, G. 2005. OpenMusic 5: A cross-platform release of the computer-assisted composition environment. In *Proceedings of the 10th Brazilian Symposium on Computer Music*. http://www.cefala.org/sbcm2005/papers/12361.pdf (accessed 1/10).

CARLSSON, M., WIDEN, J., ANDERSSON, J., ANDERSSON, S., BOORTZ, K., NILSSON, H., AND SJOLAND, T. 2009. SICStus Prolog user's manual. Release 4.1.1. http://www.sics.se/isl/sicstuswww/site/documentation.html (accessed 1/10).

CARPENTIER, G. AND BRESSON, J. 2011. Interacting with symbolic, sound and feature spaces in orchidee, a computer-aided orchestration environment. *Comput. Music J.*

CHEADLE, A. M., HARVEY, W., SADLER, A. J., SCHIMPF, J., SHEN, K., AND WALLACE, M. G. 2003. Eclipse: An introduction. Tech. rep. IC-PARC-03-1, IC-Parc, Imperial College London.

CHEMILLIER, M. AND TRUCHET, C. 2001. Two musical CSPs. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. http://recherche.ircam.fr/equipes/repmus/cpws/chemillier.ps (accessed 1/10).

CODOGNET, P. AND DIAZ, D. 2001. Yet another local search method for constraint solving. In *Proceedings of the AAAI Symposium Using Uncertainty in Computation*. T. Walsh and C. Gomes, Eds., AAAI Press, Menlo Park, CA.

CODOGNET, P., DIAZ, D., AND TRUCHET, C. 2002. The adaptive search method for constraint solving and its application to musical CSPs. In *Proceedings of the International Workshop on Heuristics (IWH02)*. http://www.normalesup.org/∼truchet/publis/iwh02.ps.gz (accessed 1/10).

CONEN, H. 1991. *Formel-Komposition: Zu Karlheinz Stockhausens Musik der siebziger Jahre*. Number 1 in Kölner Schriften zur Neuen Musik. Schott, Mainz.

COOPER, G. AND MEYER, L. B. 1960. *The Rhythmic Structure of Music*. University of Chicago Press, Chicago, IL.

COURTOT, F. 1990. A constraint based logic program for generating polyphonies. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco, CA.

DANNENBERG, R. B. 1993. Music representation issues, techniques, and systems. *Comput. Music J. 17,* 3, 20–30.

DECHTER, R. 2003. *Constraint Processing*. Morgan Kaufmann, San Francisco, CA.

DENT, M. J. AND MERCER, R. E. 1994. Minimal forward checking. In *Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence*. 432–438. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.1561&rep=rep1&type=pdf (accessed 1/10).

DEVILLE, Y., DOOMS, G., ZAMPELLI, S., AND DUPONT, P. 2005. CP(Graph+Map) for approximate graph matching. In *Proceedings of the 1st International Workshop on Constraint Programming Beyond Finite Integer Domains*. F. Azevedo, C. Gervet, and E. Pontelli, Eds. 31–47. http://www.info.ucl.ac.be/∼pdupont/pdupont/pdf/beyondfd05.pdf (accessed 1/10).

DIAZ, D. AND CODOGNET, P. 2001. Design and implementation of the GNU prolog system. *J. Funct. Logic Programm*. http://pauillac.inria.fr/∼diaz/gnu-prolog (accessed 1/10).

DÍAZ, J. F., GUTIERREZ, G., OLARTE, C. A., AND RUEDA, C. 2005. Using constraint programming for reconfiguration of electrical power distribution networks. In *Proceedings of the 2nd International Conference on Multiparadigm Programming in Mozart/OZ (MOZ'04)*. P. V. Roy, Ed., Lecture Notes in Computer Science vol. 3389. Springer, Berlin.

DOOMS, G., DEVILLE, Y., AND DUPONT, P. 2005. CP(Graph): Introducing a graph computation domain in constraint programming. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 3709. Springer, Berlin.

DOTY, D. B. 2002. *The Just Intonation Primer. An Introduction to the Theory and Practice of Just Intonation*, 3rd Ed. Just Intonation Network, San Francisco, CA.

EBCIOGLU, K. 1980. Computer counterpoint. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco.

EBCIOGLU, K. 1987. Report on the CHORAL project: An expert system for harmonizing four-part chorales. Tech. rep. 12628, IBM Thomas J. Watson Research Center.

EBCIOGLU, K. 1992. An expert system for harmonizing chorales in the style of J.S. Bach. In *Proceedings of the Conference on Understanding Music with AI: Perspectives on Music Cognition*. M. Balaban, K. Ebcioglu, and O. Laske, Eds., MIT Press, Cambridge, MA, Chapter 12, 295–332.

FERNEYHOUGH, B. 1995. *Collected Writings*. (Edited by James Boros and Richard Toop). Routledge, London.

FORTE, A. 1973. *The Structure of Atonal Music*. Yale University Press, New Haven, CT.

FRÜHWIRTH, T. AND ABDENNADHER, S. 2003. *Essentials of Constraint Programming*. Springer, Berlin.

FUX, J. J. 1965, orig. 1725. *The Study of Counterpoint. From Johann Joseph Fux's Gradus ad Parnassum*. (Translated and edited by Alfred Mann.) W.W. Norton & Company, London.

GANN, K. 2006. *The Music of Conlon Nancarrow*. Cambridge University Press, Cambridge, UK.

GERVINK, M. 2003, orig. 1995. Die strukturierung des tonraums. Versuche einer systematisierung von zwölftonreihen in den 1920er bis 1970er Jahren. In *Systemische Musikwissenschaft. Festschrift Jobst P. Fricke zum 65. Geburtstag*. W. Auhagen, B. Gätjen, and K. W. Niemöller, Eds. Köln. http://www.uni-koeln.de/phil-fak/muwi/fricke/ (accessed 1/10).

GLOVER, F. AND LAGUNA, M. 1997. *Tabu Search*. Springer, Berlin.

HENZ, M., LAUER, S., AND ZIMMERMANN, D. 1996. COMPOzE—intention-based music composition through constraint programming. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Computer Society Press.

HILLER, L. AND ISAACSON, L. 1993, orig. 1958. Musical composition with a high-speed digital computer. In *Machine Models of Music*, S. M. Schwanauer and D. A. Lewitt, Eds. MIT Press, Cambridge, MA.

JEPPESEN, K. 1939. *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*. Prentice-Hall, New York.

JEPPESEN, K. 1971, orig. 1930. *Kontrapunkt*, 4th ed. Breitkopf & Härtel, Leipzig.

JONES, D. E. 2000. A computational composer's assistant for atonal counterpoint. *Comput. Music J. 24,* 4, 33–43.

KELLY, J. 1997. *The Essence of Logic*. Prentice Hall, Upper Saddle River, NJ.

KOCH, H. C. 2000, orig. 1782–1793. *Versuch einer Anleitung zur Komposition*. Georg Olms Verlag, Hildesheim.

KRETZ, J. 2003. Continuous gestures of structured material. Experiences in computer aided composition. In *Proceedings of the PRISMA 01 Conference*.

KŘENEK, E. 1952. *Zwölfton-Kontrapunkt-Studien*. B. Schott's Söhne, Mainz.

LAURSON, M. 1996. PATCHWORK: A visual programming language and some musical applications. Ph.D. thesis, Sibelius Academy, Helsinki.

LAURSON, M. AND KUUSKANKARE, M. 2000. Towards idiomatic instrumental writing: A constraint based approach. In *Proceedings of the Symposium on Systems Research in the Arts*.

LAURSON, M. AND KUUSKANKARE, M. 2001. A constraint based approach to musical textures and instrumental writing. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. http://recherche.ircam.fr/equipes/repmus/cpws/laurson.ps (accessed 1/10).

LAURSON, M. AND KUUSKANKARE, M. 2005. Extensible constraint syntax through score accessors. In *Journées d'Informatique Musicale*. Association Française d'Informatique Musicale, Paris.

LAURSON, M., KUUSKANKARE, M., AND NORILO, V. 2009. An overview of PWGL, A visual programming environment for music. *Comput. Music J. 33,* 1, 19–31.

LERDAHL, F. AND JACKENDOFF, R. 1983. *A Generative Theory of Tonal Music*. MIT Press, Cambridge, MA.

LÖTHE, M. 1999. Knowledge based automatic composition and variation of melodies for minuets in early classical style. In *Proceedings of the Advances in Artifical Intelligence: 23rd Annual German Conference on Artificial Intelligence (KI-99)*. W. Burgard, T. Christaller, and A. B. Cremers, Eds., Lecture Notes in Computer Science, vol. 1701. Springer, Berlin, 159–170.

MESSIAEN, O. 1944. *The Technique of My Musical Language*. (Translated by John Satterfield.) Alphonse Leduc, Paris.

MIRANDA, E. R. 2001. *Composing Music with Computers*. Focal Press, Burlington, MA.

MONZO, J. 2005. Encyclopedia of microtonal music theory. http://tonalsoft.com/enc/encyclopedia.aspx (accessed 1/10).

MORRIS, R. AND STARR, D. 1974. The structure of all-interval series. *J. Music Theory 18,* 2, 364–389.

MOTTE, D. D. L. 1981. *Kontrapunkt*. Bärenreiter-Verlag, Kassel/Basel.

MOTTE, D. D. L. 1993. *Melodie*. dtv/Bärenreiter, Kassel/Basel.

NIENHUYS, H.-W. AND NIEUWENHUIZEN, J. 2003. Lilypond, A system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM'03)*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6160&rep=rep1&type=pdf (accessed 1/10).

OVANS, R. D. 1990. An object-oriented constraint satisfaction system applied to music composition. M.S. thesis, Simon Fraser University.

PACHET, F. 1993. An object-oriented representation of pitch-classes, intervals, scales and chords: The basic MusES. Tech. rep., LAFORIA-IBP-CNRS, Universite Paris VI.

PACHET, F. 1994. The MusES system: An environment for experimenting with knowledge representation techniques in tonal harmony. In *Proceedings of the 1st Brazilian Symposium on Computer Music (SBC&M'94)*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.2026&rep=rep1&type=pdf (accessed 1/10).

PACHET, F. AND CODOGNET, P. 2001. *Constraints J.* 6, 1.

Pachet, F., Ramalho, G., Carrive, J., and Cornic, G. 1996. Representing temporal musical objects and reasoning in the MusES system. *J. New Music Res. 5,* 3, 252–275.

Pachet, F. and Roy, P. 1995. Mixing constraints and objects: A case study in automatic harmonization. In *Proceedings of TOOLS-Europe'95 Conference*. I. Graham, B. Magnusson, and J.-M. Nerson, Eds., Prentice-Hall, 119–126.

Pachet, F. and Roy, P. 1998. Formulating constraint satisfaction problems on part-whole relations: The case of automatic musical harmonization. In *Proceedings of the Workshop on Constraints for Artistic Applications (ECAI'98)*. http://www.few.vu.nl/∼eliens/design/multimedia/@archive/ecai98/pachetroy.pdf (accessed 1/10).

Pachet, F. and Roy, P. 2001. Musical harmonization with constraints: A survey. *Constraints J. 6,* 1, 7–19.

Palamidessi, C. and Valencia, F. D. 2001. A temporal concurrent constraint programming calculus. In *Proceedings of the Conference on Principles and Practice of Constraint Programming (CP'01)*. Springer, Berlin.

Papadopoulos, G. and Wiggins, G. 1999. AI methods for algorithmic composition: A survey, a critical view and future prospects. In *Proceedings of the AISB Symposium on Musical Creativity*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.8064&rep=rep1&type=pdf (accessed 1/10).

Partch, H. 1974. *Genesis of a Music: An Account of a Creative Work, Its Roots and Its Fulfillments*, 2nd Ed. DaCapo Press, New York.

Perle, G. 1991. *Serial Composition and Atonality: An Introduction to the Music of Schoenberg, Berg, and Webern*. 6th Ed. University of California Press, Berkeley, CA.

Phon-Amnuaisuk, S. 2001. An explicitly structured control model for exploring search space: Chorale harmonisation in the style of J.S. Bach. Ph.D. thesis, Centre for Intelligent Systems and their Application, Division of Informatics, University of Edinburgh.

Phon-Amnuaisuk, S. 2002. Control language for harmonisation process. In *Proceedings of the 2nd International Conference on Music and Artificial Intelligence (ICMAI'02)*, C. Anagnostopoulou, M. Ferrand, and A. Smaill, Eds., Lecture Notes in Computer Science, vol. 2445, Springer, Berlin.

Piston, W. 1947. *Counterpoint*. W. W. Norton, New York.

Pope, S. T. 1992. The smoke music representation, description language, and interchange format. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco.

Rameau, J. P. 1984, orig. 1722. *Treatise on Harmony (Traité de l'Harmonie Réduite à ses Principes Naturels)*. Dover, New York. translated, with and introduction and notes, by Philip Gossett.

Ramirez, R. and Peralta, J. 1998. A constraint-based melody harmonizer. In *Proceedings of the Workshop on Constraints for Artistic Applications (ECAI'98)*. http://www.math.vu.nl/∼eliens/research/multimedia/@archive/vrml-reference/@archive/ecai98/ramirez.pdf (accessed 1/10).

Reich, S. 2002. *Writings on Music, 1965-2000*. (Edited by Paul Hillier.) Oxford University Press, Oxford, UK.

Riemann, H. 1887. *Handbuch der Harmonielehre*. Breitkopf & Härtel, Wiesbaden, Germany.

Roads, C. 1996. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, Chapter 18 and 19, 819–910.

Rossi, F., Beek, P. V., and Walsh, T. 2006. *Handbook of Constraint Programming*. Elsevier, Amsterdam.

Roy, P. and Pachet, F. 1997. Reifying constraint satisfaction in smalltalk. *J. Object-Orient. Program. 10,* 4, 43–51.

Rueda, C., Alvarez, G., Quesada, L. O., Tamura, G., Valencia, F. D., Díaz, J. F., and Assayag, G. 2001. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints 6,* 1, 21–52.

Rueda, C., Lindberg, M., Laurson, M., Block, G., and Assayag, G. 1998. Integrating constraint programming in visual musical composition languages. In *Proceedings of the Workshop on Constraints for Artistic Applications (ECAI'98)*. http://www.cs.vu.nl/∼eliens/poosd/@online/@share/archive/ecai98/rueda.ps (accessed 1/10).

Rueda, C., Tamura, G., and Quesada, L. O. 1997. The visual model of cordial. In *Proceedings of the XXIII Conferencia Latinoamericana de Informática (CLEI'97)*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.8836 (accessed 1/10).

Rueda, C. and Valencia, F. 2004. On validity in modelization of musical problems by CCP. *Soft Comput. Fusion Found. Method. Appl. 8,* 9, 641–648.

Rueda, C. and Valencia, F. D. 1997. Improving forward checking with delayed evaluation. In *Proceedings of the XXIII Conferencia Latinoamericana de Informática (CLEI'97)*.

Sandred, O. 2000. *OMRC 1.1. A Library for Controlling Rhythm by Constraints* 2nd Ed. IRCAM, Paris.

SANDRED, O. 2003. Searching for a rhythmical language. In *Proceedings of the PRISMA 01 Conference*. EuresisEdizioni, Milano.

SANDRED, O. 2004. Interpretation of everyday gestures—Composing with rules. In *Proceedings of the Music and Music Science Conference*. www.sandred.com/Sandred-gestures.pdf (accessed 1/10).

SANDRED, O. 2009. Approaches to using rules as a composition method. *Contemp. Music Rev. 28,* 2, 149–165.

SANDRED, O. 2011. PWMC, A constraint solving system for generating music scores. *Comput. Music J*.

SARASWAT, V. A., JAGADEESAN, R., AND GUPTA, V. 1994. Foundations of timed concurrent constraint programming. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94)*. 71–80. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.8333&rep=rep1&type=pdf (accessed 1/10).

SCHENKER, H. 1956, orig. 1935. *Der freie Satz*. Universal Edition, Wien.

SCHILINGI, J. 2009. Local and global control in computer-aided composition. *Contemp. Music Rev. 28,* 2, 181–191.

SCHOENBERG, A. 1943. *Models for Beginners in Composition*. Belmont, Los Angeles.

SCHOENBERG, A. 1964. *Preliminary Exercises in Counterpoint*. St. Martin's Press, New York.

SCHOENBERG, A. 1967. *Fundamentals of Musical Composition*. Ed. Gerald Strang and Leonard Stein. Faber and Faber, London.

SCHOENBERG, A. 1969. *Structural Functions of Harmony*, 2nd Ed. W. W. Norton, New York.

SCHOENBERG, A. 1986, orig. 1911. *Harmonielehre*, 7th Ed. Universal Edition, Wien.

SCHOENBERG, A. 1995. *The Musical Idea and the Logic, Technique, and Art of Its Presentation*. (Edited, translated, and with a commentary by Patricia Carpenter and Severine Neff.) Columbia University Press, New York.

SCHOTTSTAEDT, W. 1989. Automatic counterpoint. In *Current Directions in Computer Music Research*, M. V. Mathews and J. R. Pierce, Eds., The MIT Press, Cambridge, MA.

SCHULTE, C. 2002. *Programming Constraint Services*. Lecture Notes in Artificial Intelligence, vol. 2302. Springer-Verlag, Berlin.

SCHULTE, C., TACK, G., AND LAGERKVIST, M. Z. 2010. Modeling with gecode. http://www.gecode.org/doc-latest/modeling.pdf (accessed 1/10).

SELFRIDGE-FIELD, E. 1997. *Beyond MIDI. The Handbook of Musical Codes*. MIT Press, Cambridge, MA.

SISKIND, J. M. AND MCALLESTER, D. A. 1993. Screamer: A portable efficient implementation of nondeterministic common lisp. Tech. rep. IRCS-93-03, University of Pennsylvania Insitute for Research in Cognitive Science.

TAUBE, H. 1997. An introduction to common music. *Comput. Music J. 21,* 1, 29–34.

TAUBE, H. 2004. *Notes from the Metalevel*. Swets & Zeitlinger Publishing, Lisse, The Netherlands.

TRUCHET, C., ASSAYAG, G., AND CODOGNET, P. 2001. Visual and adaptive constraint programming in music. In *Proceedings of International Computer Music Conference*. International Computer Music Association, San Francisco.

TRUCHET, C. AND CODOGNET, P. 2004. Solving musical constraints with adaptive search. *Soft Comput. 8,* 9, 633–640.

TSANG, C. P. AND AITKEN, M. 1991. Harmonizing music as a discipline of constraint logic programming. In *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco.

VAN BEEK, P. 2006. Backtracking search algorithms. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds., Elsevier B.V., Amsterdam.

WIGGINS, G., MIRANDA, E., SMAILL, A., AND HARRIS, M. 1993. A framework for the evaluation of music representation systems. *Comput. Music J. 17,* 3, 31–42.

ZIMMERMANN, D. 2001. Modelling musical structures. *Constraints 6,* 1, 53–83.

Received April 2009; revised October 2009; accepted December 2009