View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Bedfordshire Repository

Chapter 10: Compositions Created with Constraint Programming

Torsten Anders

Abstract: This chapter surveys music constraint programming systems, and how composers have used them. The chapter motivates and explains how users of such systems describe intended musical results with constraints. This approach to algorithmic composition is similar to the way declarative and modular compositional rules have successfully been used in music theory for centuries as a device to describe composition techniques. In a systematic overview, this survey highlights the respective strengths of different approaches and systems from a composer's point of view, complementing other more technical surveys of this field. This text describes the music constraint systems PMC, Score-PMC, PWMC (and its successor Cluster Engine), Strasheela and Orchidée -- most are libraries of the composition systems PWGL or OpenMusic. These systems are shown in action by discussing the composition process of specific works by Jacopo Baboni-Schilingi, Magnus Lindberg, Örjan Sandred, Torsten Anders, Johannes Kretz and Jonathan Harvey.

Keywords: constraint programming; music constraint programming; rule-based; compositional rule; algorithmic composition; OpenMusic; PWGL

1. Introduction

This chapter surveys approaches that use constraint programming for algorithmically composing music. In a nutshell, constraint programming is a method to implement compositional rules – rules as found in music theory textbooks; rules formulated by composers to model their compositional style; or piece-specific rules. Constraint solvers efficiently search for musical solutions that obey all constraints applied.

Fernández and Vico (2013) propose a taxonomy of artificial intelligence (AI) methods for algorithmic composition that distinguishes between symbolic AI, optimisation techniques based on evolutionary algorithms and related methods, and machine learning. Constraint programming is a symbolic method. Other AI methods are discussed in other chapters of this book (e.g., machine learning in chapter 8, evolutionary algorithms in chapter 9, and other symbolic approaches in chapter 10).

Rule-based algorithmic composition is almost as old as computer science. The pioneering *Illiac Suite* (1956) for string quartet already used a generate-and-test algorithm for the composition process, where randomly generated notes were filtered in order to ensure they meet constraints of different compositional styles, such as strict counterpoint for the second movement, and chromatic music for the third movement (Hiller and Isaacson 1993). Ebcioğlu (1980) proposed likely the first system where a systematic search algorithm was used for composing music (florid counterpoint for a given cantus firmus). Ebcioğlu later extensively modelled Bach chorales (Ebcioglu 1987).

Algorithmic composition with constraint programming has been surveyed before. Pachet and Roy (2001) review harmonic constraint problems. Fernández and Vico (2013) provide a comprehensive overview of music constraint problems and systems in the context of AI methods in general. Anders and Miranda (2011) survey the field in detail, and carefully compare music constraint systems. However, these surveys tend to focus on the technical side, as they are published in computer science journals.

This chapter complements these surveys by focussing on how several composers employed constraint programming for their pieces. Constraint programming systems are briefly presented to introduce techniques used for those compositions.

In this context, the composer is responsible for the final aesthetic result, and computers merely assistant in the composition process. Composers therefore cherry-pick or manually edit the musical results. To emphasise such artistic responsibility and liberty of composers, this field is often called computer-aided composition instead of algorithmic composition, but in this chapter we keep the term algorithmic composition for consistency with the rest of this book.

2. What is Constraint Programming?

Constraint programming (Apt 2003) is a highly declarative programming paradigm that is well suited to automatically solve combinatorial problems, called constraint satisfaction problems. The general idea is easy to understand: constraint programs are much like a set of equations (or inequations) with variables in algebra: a solver finds values for all variables such that all equations (inequations) hold.

Each *variable* is defined with a *domain*, a set of possible values it can take in a solution. Variable domains typically contain multiple values initially, so that the variable value is unknown. In a musical application, the domain of a pitch variable may be, say, all pitches between C4 (middle C) and B4. The search process by and by reduces variable domains in order to find a solution.

Constraints restrict relations between variables. Examples include unary relations (e.g., `isOdd`), binary relations (e.g., <, =, +, −), and relations between more elements (e.g., `allDistinct`). A constraint *solver* searches for one or more *solutions*, where each variable is bound to a single value of its domain without violating any of its constraints.

The general constraint literature clearly distinguishes between variables of different domains (quasi types), such as integer variables, Boolean variables (variables of truth values), float variables, and set (of integer) variables. Modern constraint programming systems define individual constraints by algorithms that depend on such "type" information, which reduce variable domains without search depending on the constraints applied to them (constraint propagation, (Tack 2009)).

By contrast, in most constraint systems developed for music composition, constraints are simply test functions returning a Boolean. In this context we can therefore often ignore these domain distinctions. A disadvantage of not using constraint propagation is a reduced speed of the search process, but the search is nevertheless fast enough to be useful in practise. An advantage of simpler constraints is a greater flexibility. Variable domains can consist of any values (e.g., in a later section we will discuss an orchestration example, where variable domains are symbols). Perhaps most importantly, with this simple approach virtually any function of the host programming language can be used for defining user-constraints.

The existence of efficient solvers had an important impact on the success of constraint programming in general. For musicians, a major appealing factor is the relative ease by which common music theory rules can be encoded so that automatically generated music complies with such rules.

2.1. A Minimal Counterpoint Definition

The following presents a two-part counterpoint definition, to provide a practical example. The example is musically somewhat simplistic (much simpler than text-book examples, e.g., Fux (1965)), but the point here is to demonstrate underlying principles of the actual implementation, and not a convincing musical solution. Later, we will discuss more advanced examples, but in less

detail.

For simplicity, this is a first species counterpoint example, i.e., all durations are the same, and every note in both parts has a simultaneous note in the other part.

The resulting music is unknown before the search: every pitch of both parts is represented by a variable with a domain that includes, say, all white keys on the keyboard between A3 and G5. It is useful to represent pitches numerically – that way concepts like intervals are easily defined. Commonly, pitches are represented as MIDI note numbers (Rothstein 1995).

Only two constraints are defined: a melodic constraint, and a harmonic constraint. Melodic intervals are limited to 2 semitones at most (steps, or repetitions). That constraint can be defined by an inequation that first computes the interval (the absolute difference) measured in semitones between two consecutive note pitches of one part, $pitch_1$ and $pitch_2$, and then limits that interval to be at most 2 semitones, as shown in equation (1).

$$|pitch_2 - pitch_1| \quad (1)$$

The harmonic constraint requires all simultaneous note pairs to form consonances. That constraint computes again an interval, but this time between two simultaneous pitches p_1 and p_2, and requires that this interval is an element in a set of consonant intervals, say, the intervals unison, minor third, major third, fifth, minor sixth, major sixth and octave, all measured in semitones.

$$|p_2 - p_1| \in \{0,3,4,7,8,9,12\} \quad (2)$$

In the music constraint programming systems introduced below, these constraints would be largely defined as above (though they will use the syntax of different programming languages – we used mathematical notation only for clarity). However, the above definitions are not complete, and it will turn out that different systems clearly differ in these missing parts.

In music constraint systems, the pitch variables of the two parts would be organised in some music representation that defines their relations (e.g., which pitch variable belongs to which part and at what position). Different systems clearly differ in their music representations, which lead to specific capabilities and limitations.

We did not actually model above how the harmonic constraint is applied to pitches of simultaneous notes, or the melodic constraint to pitches of consecutive note pairs in the same part. Different music constraint systems implement different paradigms to control the application of a constraint to variable sets in the score.

Finally, we did not discuss how a constraint solver actually solves the above example. The search strategies of constraint solvers of different music constraint systems also differ clearly.

Figure 1 shows a visual example implementation. The constraint solver receives a score that specifies the rhythm of the two voices, and the pitch domains as a sequence of MIDI note numbers representing the white keys on the keyboard between A3 and G5. The constraint boxes hide the details of the constraint implementation and their application to variables, but the melodic interval threshold and the possible harmonic intervals are shown as arguments to these boxes. This implementation uses the constraint system Score-PMC, and ready-made constraints from a collection by Jacopo Baboni-Schilingi; both are discussed in more detail below.

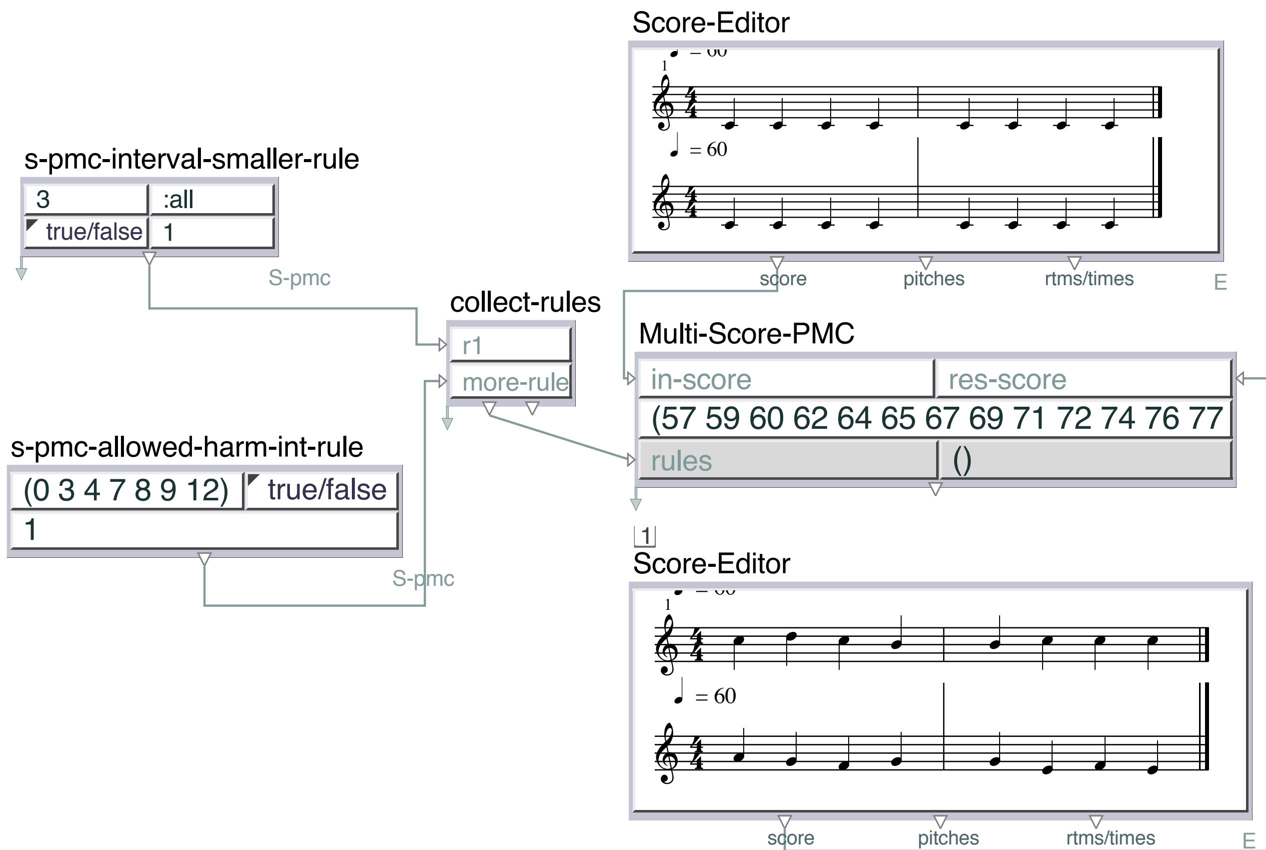

Figure 1: Implementation of a minimal counterpoint constraint problem with Score-PMC and predefined constraints by Baboni-Schilingi.

3. Constraining a Single Parameter

This section and the following two sections present different approaches to using constraint programming for music composition. Alternating subsections introduce music constraint systems, and discuss compositions created with those systems.

Music constraint programming systems have been designed with a certain range of musical constraint problems in mind, which they can solve. The presented systems are intended to let users such as composers model their own constraints and then combine multiple constraints to their own music theories. In order to make that easier, most systems provide basically a template that conveniently solves a certain class of problems, while other problems can only be solved awkwardly or not at all.

By contrast, constraint programming systems and libraries in general (outside music) are far more flexible, and support more advanced solving strategies. Examples of widely used general constraint programming systems include several Prolog implementations, e.g., ECLiPSe (Apt and Wallace 2007) and SICStus (Carlsson 2014), and the C++ library Gecode (Schulte, Tack, and Lagerkvist 2015). An interesting recent development is the constraint modelling language MiniZinc, which provides a high-level and relatively user-friendly front end for various state-of-the-art solvers (Nethercote et al. 2007). However, these systems are designed for experienced computer programmers, which make them inaccessible for most users of music constraint systems.

Many music systems have been designed as libraries of the visual composition systems PWGL (Laurson, Kuuskankare, and Norilo 2009), and OpenMusic (Assayag et al. 1999). Such integration

provides direct access to powerful tools such as score editors, and export functionality to commercial music notation software. It also allows for the combination of constraint programming with other algorithmic composition paradigms. Some of these libraries have already a long history that goes back to the common predecessor of PWGL and OpenMusic, PatchWork (Laurson 1996).

3.1. Constraining Sequences: PMC

PMC is a built-in constraint solver of PWGL for solving general constraint problems. It is inherited from PatchWork, and has been partly ported to OpenMusic as the library OMCS.

PMC is particularly suitable for constraining compositional material before it is part of a score, because its music representation is always a flat sequence (list) of variables (typically with numeric domains, but any types are supported). Such representation is useful, for example, to search for twelve-tone rows, the pitches of a chord, the durations forming a rhythm, the pitch sequence of a melody, and so on.

Even though PMC is an integral part of a visual programming language, its constraints are interestingly defined by textual code in the programming language Common Lisp (the language in which PatchWork itself, and its successors are defined). The melodic constraint of equation (1) above can be readily translated into Lisp syntax, but it is then still incomplete.

Constraints are applied to variables in sequences by complementing them with *pattern matching* expressions (simple cousins of regular expressions): a constraint is applied to all variable sets that match its given pattern. PMC's pattern-matching language allows for wildcards: `*` matches any number of variables. For example, melodic constraints that restrict consecutive variable pairs can be applied with the following pattern matching expressions: `* ?1 ?2`, where `*` matches any number of variables preceding the actual match (including none), while `?1` and `?2` denote two variables to be matched. The Lisp code part of the constraint then uses these symbols to refer to those variables. The importance of pattern matching for PMC is also reflected in its name, which stands for pattern matching constraints.

Note that the harmonic constraint of equation (2) above cannot be implemented in PMC (at least not alongside a melodic constraint): PMC's purely sequential music representation cannot express melodic and harmonic relations at the same time.

PMC solves constraint problems with the classical backtracking algorithm (Dechter 2003), which by and by completes a partial solution. The algorithm selects a variable and a value of its domain, and tests whether all constraints applied to it hold. If so, this domain value is (for now) considered the solution for that variable, and the algorithm continues with the next variable. However, if any constraint fails, the algorithm by and by tries other domain values of the same variable until the algorithm finds one of them that satisfies all constraints, so it can be the solution. The algorithm then proceeds to the next variable. If no domain value of a certain variable fulfils all constraints, then the algorithm must *backtrack* by re-visiting the previously visited variable to continue with its other domain values. Backtracking performs a complete search: if a solution exists, then the algorithm finds one.

An interesting feature of PMC, highly useful for musical purposes, is its support of heuristic constraints: a solution should obey *heuristic constraints* if this is (easily) possibly, otherwise such constraints can be broken. A similar concept in the general constraint literature are soft constraints (Meseguer, Rossi, and Schiex 2006). While strict constraints in PMC simply return a Boolean (true or false), heuristic constraints return a number indicating its weight (used if multiple heuristic constraints are in conflict).

The solver favours domain values that meet heuristic constraints by trying them first: domain values are tested in the order of the numeric values returned by the heuristic constrains. While this

approach does not necessarily find the best solution, it quickly finds reasonable approximations.

3.2. Embedded Constraint Problems: Jacopo Baboni-Schilingi

Baboni-Schilingi extended PMC (and Score-PMC, presented below) by a sizeable collection of ready-to-use constraints (about 120 constraints for PMC).[1] Examples include constraints disallowing various cases of repetitions; diverse constraints controlling pitches and melodic intervals (e.g., inspired by classical counterpoint); and constraints controlling short value subsequences (useful for enforcing structure, like motifs).

All constraints come as user-friendly graphical PWGL boxes (outputting Lisp code required by PMC) with various arguments to control the effect of these constraints. Also, all constraints can be easily switched to heuristic constraints, which is useful for avoiding over-constrained problems.

Baboni-Schilingi likes to develop patches that invite quick editing in many ways to tweak the output or create variations.[2] The result is then output to commercial music notation software for further editing (via MusicXML). Patches typically generate musical material for a short section, where musical parameters such as the rhythm, pitches, but also music notation details like articulations, expressions, and grace notes are given as independent sequences – written manually, or created algorithmically.

He often uses constraint programming with PMC to algorithmically generate such parameter sequences. For example, *de la nature du sacre* for string quartet and computer (2012) ends with a fast section of short musical cells (motifs) with irregular accents due to constant metre changes. Material for this section was created with a patch where six individual cells are composed manually (sequences of pitches, durations, and time signatures). The patch generates a sequence of cells that is constrained to play at least four different cells before a cell could be repeated.

Two interesting approaches are the constraint-based transformation (refining) of parameter sequences that have been generated with other algorithmic techniques, and heuristic profile constraints. For instance, *Aura-phoenix* for violin and computer (2015) contains long downward violin gestures that were generated in multiple steps. A rough version of the pitch sequence was created by interpolating two given pitch sequences (i.e., generating intermediate sequences – concatenated to a long sequence). PMC was then used to refine this sequence: direct repetitions should be avoided, but otherwise the overall shape of the rough version should be followed. A heuristic profile constraint expressed a preference to roughly follow the original pitch sequence. Profile constraints are an important means for Baboni-Schilingi to control the overall development, while other constraints control local contexts (Schilingi 2009).

Note that random sequences of cells with restricted repetition as sketched above could also be obtained by other methods, for example, Common Music's `heap` pattern (Taube 2004) (see also chapter 10). By contrast, sequences that comply with two or more constraints on the same values are very challenging to create with methods that do not search for a solution.

3.3. Constraining Pitches in a Polyphonic Score: Score-PMC

While PMC is not able to constrain musical relations in a score that go beyond mere sequential relations, its sibling Score-PMC has been designed for constraining polyphonic scores. For this purpose, Score-PMC features a music representation that supports multiple parts, where both the

---

[1] Baboni-Schilingi's libraries are freely available at http://baboni-schilingi.com/index.php?/recherches/software/ (accessed 2 April 2015).

[2] This subsection is based partly on personal communication, and patches created by the composer for recent compositions.

rhythmic structure, the pitch structure, and even details like articulations are represented.

However, most of this information is static during the search process. The only variables are the pitches in the score, whose domain consist in integers representing MIDI notes.

With Score-PMC we can implement both constraints defined in expression (1) and (2) above. Like with PMC, constraints are defined by Lisp expressions, and simple melodic constraints are largely defined as in PMC. For constraints that depend on other information (e.g., simultaneous pitches across parts, or the meter), Score-PMC features an extended pattern-matching formalism (Laurson and Kuuskankare 2005).

Score-PMC also uses backtracking. For an efficient search process, it first computes a suitable order in which notes should be visited during the search process, which progresses more or less in score time. The search jumps between voices for an efficient search, instead of completing parts one after each other, because otherwise conflicts of harmonic constraints are detected too late, resulting in unnecessary work. This efficient order depends on the rhythmic structure, which is the reason why Score-PMC depends on a completed rhythmic score before the search starts.

Score-PMC can also search for rhythms, but in that mode pitches are not represented any more (Laurson and Kuuskankare 2001). While this functionality is interesting for computing certain textures (e.g., the rhythmic structure of Ligeti-like counterpoint), it is still limited to searching for only a single parameter.

3.4. Engine *by Magnus Lindberg*

Magnus Lindberg used Score-PMC to compose *Engine* for chamber orchestra (1996) (Laurson and Kuuskankare 2009). Lindberg was interested in working with constraints programming, because it forced him to analyse and better understand his own compositional style, and also to avoid mannerisms of his style.[3]

As Score-PMC requires that the rhythmic structure is fully pre-composed, Lindberg created rhythms with a self-developed library that featured a simple representation of rhythms as sequences of fractions. Such data can be manually composed, generated or transformed in many ways.

Score-PMC was then used to compose individual sections of the piece. Constraints were assigned per section; exemplar constraints are discussed here. While the different constraints can be organised in traditional music theory categories (melodic, harmonic, and voice-leading constraints), Lindberg aimed for a clearly personal style. For example, he developed a personal collection of melodic constraints that permit certain interval successions in order to generate music with distinct characteristics. Other melodic constraints ensured that individual tones did not stand out too much: octaves should be avoided between local pitch minima and maxima that are close to each other, and repetitions between two or more consecutive notes were prohibited. Harmonic constraints required simultaneous pitches to form given pitch class sets. Further harmonic constraints refined the result (for example: no octaves). Longer chord formations tended to be more constrained than short ones. Finally, voice-leading constraints controlled the relation between adjacent parts. Example constraints forbid voice crossing, or required a minimum/maximum distance between adjacent parts.

Overall, for the composer it was important to localise the effect of constraints. Pre-composed rhythmic structures already showed certain musical ideas, and single sections could display a counterpoint of different textures and characteristic gestures that called for different constraints. An easy approach is to apply constraints only to certain parts, but constraints could also depend on the

---

[3] Cf. the programme notes of this composition, available at http://www.musicsalesclassical.com/composer?category=Works&workid=7693 (accessed 2 April 2015).

rhythmic situation. For example, some phrases with longer durations (or notes interrupted by rests) allowed for large pitch skips, while some rapid phrases required smaller intervals. More generally, a melodic interval constraint could depend on its “rhythmic interval”: a short note followed by a short note could be constrained differently than a short note followed by a long note, and so on.

Results were exported into commercial music notation software (Finale, via the intermediate format Enigma), where it was edited manually. Also the orchestration was done manually in Finale.

4. Constraining Multiple Parameters

Score-PMC always requires a pre-composed rhythmic score, because it computes the order in which all variables are visited before the search starts (*static variable ordering*). By contrast, systems presented in this section compute which variable to visit next only when this information is actually needed (*dynamic variable ordering*), which allows them to efficiently search for durations, pitches and possibly further musical parameters of a polyphonic score in parallel.

4.1. PWMC

Sandred (2003) extensively studied how to model rhythm with constraints, and developed an OpenMusic library for rhythmic constraint problems (OMRC, (Sandred 2000)), before he started to work on a new constraint system. PWMC (Sandred 2010) is a PWGL library that solves polyphonic constraint problems, where time signatures, durations, and note pitches can be variables.

PWMC is relatively user-friendly. Constraint problems – including custom constraints – can be expressed by visual programming (experienced users can also write textual Lisp code for more concise definitions).

Users control, which variables are affected by a certain constraint with special boxes for various score contexts. For example, melodic constraints are applied to consecutive pitch variables in a part with the box `access-melody`. The actual constraint is defined independently in a PWGL abstraction (a sub-patch). Even inexperienced programmers can easily switch the abstraction into “Lambda mode” so that it returns the abstraction definition wrapped in a function. As a Lisp dialect, PWGL supports functional programming, where functions themselves are values that can be passed around (Abelson, Sussman, and Sussman 1985). This function is given to a box like `access-melody` as an argument. Other arguments of constraint applicator boxes such as `access-melody` control various further details, like to which part the constraint should be applied, and whether it should hold only in certain situations. For example, a harmonic constraint may be applied to every note, or only to the first note in each bar.

Users can express groupings of durations and pitches by motifs. Interestingly, rhythmic and melodic motifs are independent and not “synchronised”, much as in isorhythm the color and talea are not synchronised. Melodic motifs can also be freely transposed for variety, but intervals between their tones will not change.

The order in which variables are visited during the search process can have a crucial impact on efficiency (Beek 2006). PWMC therefore allows users to customise the variable ordering by so-called strategy rules. For example, one strategy rule sets the search to complete multiple voices more or less in parallel. For polyphonic constraint problems, such search approach can be orders of magnitude faster than an approach that first finds all durations before searching for pitches (Anders 2011). For the actual search process, PWMC uses the solver PMC, but PMC’s flat sequential representation is internally mapped to a richer music representation.

Sandred currently develops a successor of PWMC called Cluster Engine,[4] which provides largely similar features, but searches more quickly for solutions, due to a custom search algorithm. Multiple solvers search quasi in parallel for different parameter sequences (e.g., the durations, pitches, and time signatures of each part). Solvers can force each other to backtrack, resulting in a kind of back-jumping (Beek 2006): in case of a fail, the algorithm tries to analyse which variable actually caused the fail and directly jumps back to that variable. Doing so avoids redundant work at intermediate variables, as it may otherwise repeatedly run into the same failure. The library Cluster Rules[5] complements Cluster Engine by a collection of predefined constraints for various purposes comparable to Baboni-Schilingi's constraint collection for PMC.

4.2. Constraining Both Rhythm and Pitch: Örjan Sandred

Constraint programming allowed Örjan Sandred to algorithmically compose rhythm in a way that carefully balances simplicity and complexity. *Kalejdoskop* for clarinet, viola and piano (1999) was composed with OMRC (Sandred 2006). The composition uses uneven durations of individual notes and rhythmic motifs (e.g., a motif may last in total nine semiquavers or sixteenth notes). This leads to rhythms that do not agree with a regular meter. Sandred proposed a constraint that creates some alignment between multiple rhythms running in parallel in order to limits the overall rhythmic complexity. He used this technique in *Kalejdoskop* to rhythmically organise analytical information (i.e., information only implicitly contained in the final score, like the harmonic rhythm), but this technique can also be used for directly controlling the rhythmic relations between multiple parts.

Figure 2 shows the rhythm at the beginning of *Kalejdoskop*: the rhythm of the phrase level (form layer) is shown at the top with long durations; the harmonic rhythm in the middle; and the actual rhythm, which is the accumulation of all note onsets of all parts in the lowest staff. The composer constrained the rhythmic complexity by aligning the rhythms of pairs of layers: whenever an event starts at a higher level, a new event must also start at the lower level (marked by arrows in Figure 2). However, new events can start in the lower level between note onsets in the upper level. The result is a rhythmic texture that can freely use uneven durations and motifs, but multiple parts support that uneven rhythm in a semi-homophony. The pitch structure is also controlled with constraints in *Kalejdoskop*, but constraints for pitches and rhythm are not interdependent.

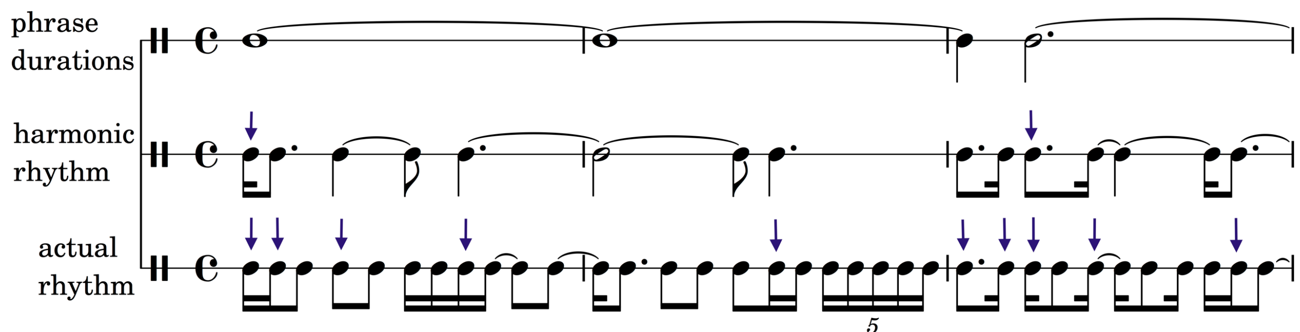

Figure 2: The hierarchical structure of rhythmic information at the beginning of Örjan Sandred, Kalejdoskop *(Sandred 2006)*

Sandred later developed PWMC (see above) that allowed him to constrain both the rhythm and pitch structure in subsequent pieces, as for example in *Whirl of Leaves* for flute and harp (2006).[6] For this piece he composed rhythmic and melodic motifs that characterise different sections, but which are not synchronised, as discussed above. Only a small subset of constraints can be discussed here, but it should be noted that the majority of constraints affect pitches (melodic and harmonic

4 Cluster Engine is available at http://sandred.com/Downloads.html (accessed 26 June 2015).

5 Cluster Rules is available at http://github.com/tanders/cluster-rules (accessed 2 April 2015).

6 The discussion of this composition is based on personal communication and slides provided by the composer.

constraints). Several constraints are inspired by conventional rules, without actually being conventional. For example, the music expresses on underlying harmony, often with a slow harmonic rhythm. A short “seed” chord progression was composed manually, largely quartal harmony with 4-6 different pitch classes. The issue of complex chord progression rules is avoided by allowing this pre-composed progression to traverse both forwards and backwards. The resulting harmony is represented explicitly by an extra part in the constraint problem (removed in the actual composition later), so that constraints can explicitly refer to that analytical information.

Some constraints directly link the rhythmic and pitch structure. For example, in a central section that reappears in varied form multiple times, changes of the underlying harmony are clearly marked by a short gesture in a different texture (suddenly skips, in a homophonic setting of octaves between the instruments). This texture change is forced by certain constraints – such constraints that react to events happening only occasionally (here, harmonic changes) represent another way to control the musical form. Another constraint links meter and harmony, again inspired by traditional rules: in some sections, pitches on downbeats must exist in the underlying quartal harmony, while other pitches can come from a slightly larger pitch set of a scale associated with the harmony. Constraints can also affect certain musical characteristics. For example, in some contrasting more quiet sections, higher notes in the melody must be longer than shorter notes, and that way stand out.

4.3. An Extensible System: Strasheela

The main design goal of Strasheela was to allow for a wider range of musical constraint problems than previous systems (Anders and Miranda 2011). Previous systems provide a constraint problem template: a certain class of problems is defined relatively easily, but other problems are only awkwardly defined, or cannot be defined at all. Strasheela offers instead a software framework: it offers building blocks that simplify the definition of music constraint problems, but these building blocks can be extended or freely redefined.

Strasheela’s music representation models musical concepts as objects (Pope 1991). Core objects are notes, and temporal containers. Such containers arrange other objects sequentially (e.g., in a part), or simultaneously in time (e.g., in a chord, or multiple parallel parts) (Dannenberg 1989). Users can extend existing objects or define their own, and Strasheela itself already provides various such extensions (e.g., objects to represent harmonic information). Users create the music representation for a constraint problem by freely arranging score objects in a hierarchy.

Users apply constraints to variables in the score by functional programming techniques much like PWMC. The differences are that variables are also directly accessible (i.e., constraints can also be applied directly), and Strasheela supports an interface for users to define their own constraint applicators from scratch, so that every score context, every possible combination of variables, can be constrained (Anders and Miranda 2010b).

Strasheela’s search performs constraint propagation (depending on the constraints applied, variable domains are reduced without search), which greatly reduces the search space and thus speeds up the search process. Also, while a number of search strategies suitable for efficiently solving various problems are predefined, users can freely program dynamic variable and value orderings of score parameters with convenient building blocks (Anders 2011).

Strasheela’s flexibility will be briefly demonstrated by sketching its support for motifs. Strasheela offers multiple motifs models. The pattern motif model is basically a generalisation of PWMC’s motifs, where a sequence of parameters (e.g., the pitches of notes in a part) is constrained to consist of given subsequences. Multiple parameters can be constrained at the same time (e.g., pitches, and durations, but also analytical parameters such as chord roots), and motifs of multiple parameters can be un-synchronised (as in PWMC), or synchronised. Motif definitions can contain other variables, so that the actual motifs themselves can be searched for too. How the motifs are actually mapped to

variables in the score can be freely defined. For example, pitch contour motifs are possible that constrain only the direction of melodic intervals.

The variation motif model defines an extra music representation for motifs that explicitly represents analytical features such as the identity and variation of a motif by variables (Anders 2009). Users define what a motif identity means (e.g., which parameters are involved, and how they are constrained), and which variations are possible (defined as a set of transformation functions).

In the prototype model, motifs are represented by sub-constraint problems that define their own music representation and constraints. Such sub-problems can then be arranged freely in higher-level temporal containers and further constrained.

All these models have their own strengths and limitations. For example, the overall number of motifs can only be constrained in the pattern motif model, only for variation motifs can the identity and variation of motifs be constrained independently, and only prototype motifs can be polyphonic.

Strasheela's implementation language Oz (Roy and Haridi 2004) features a built-in constraint system, which has been state-of-the-art at the time Strasheela development started. The nowadays widely used C++ constraint library Gecode is the successor of this constraint system. Strasheela's implementation language greatly simplified realising much of Strasheela's strengths, such as its custom score search strategies.

Strasheela is more flexible than other music constraint systems (Anders and Miranda 2011), but that flexibility comes at a certain price. Its framework approach requires more programming experience than previously introduced systems. While it provides flexible export functionality into various formats for music notation and sound synthesis, it misses the ecosystem of a widely used composition environment (e.g., other user libraries, or powerful editors). Also, its interface is a textual programming language that is rarely used for music.

4.4. Microtonal Music: **Tempziner Modulationen** *by Torsten Anders*

For the composition *Tempziner Modulationen* (2011), Torsten Anders was interested in exploring 7-limit harmony (Erlich 1998). Several 7-limit intervals sound consonant, but unusual (e.g., the harmonic seventh with frequency ratio 7:4, and the subminor third, 7:6).

Tempziner Modulationen for Fokker-organ and Carrillo-piano is composed in 31-tone equal temperament (31-TET). 31-TET is almost identical to quarter-comma meantone (Barbour 2004), a dominant tuning system of Renaissance and early Baroque music, so all intervals of traditional music are present. Additionally, 31-TET closely approximates 7-limit intervals (Fokker 1955), for example, the harmonic seventh is only 1 cent off.

Figure 3 shows an excerpt. Note that 31-TET is notated with traditionally accidentals, but tones that are enharmonically equivalent in 12-TET denote different pitches in 31-TET. For example, the interval C–A♯ denotes the harmonic seventh (C-B♭ is still the minor seventh), while C–D♯ is the subminor third. The subminor 7th chord in the third bar of the excerpt uses these intervals.

Figure 3: Excerpt from Torsten Anders, Tempziner Modulationen. *The upper three staves show the actual composition and the lower two an analysis of the underlying harmony. Chord and scale tones are shown with small notes and root notes as normal notes.*

Instead of memorising a large number of microtonal chords and scales with many transpositions, and then studying their relations, Anders modelled a suitable music theory in Strasheela (Anders and Miranda 2010a). This theory is based on ideas of conventional music theory, but with a twist: the resulting music is tonal in the sense of Tymoczko (2011), but it is decidedly non-diatonic.

The music theory for this composition constrains complex analytical information. For example, chords and scales represent their 31-TET pitch class set, root, transposition interval, identity and other harmonic features (see Fig. 3). All these features are variables that can be constrained.

Various constraints control chord progressions. Only a small set of different chord types is permitted per section (harmonic consistency), the roots of consecutive chords must differ, and the first and last chords per section are either manually set or otherwise closely controlled. Consecutive chords are connected by a small voice-leading size, defined here as the minimal sum of the intervals by which two pitch class sets differ (typically limited to a minor third overall to connect five-tone chords). This last constraint leads to smooth chord progressions.

Further constraints may be applied to chords. In some sections, chords must fall into an underlying scale (see Fig. 3). A few sections are concluded by a cadence – a short chord progression that sounds all pitch classes of the scale. In other sections, chords are not limited to an underlying scale, but all chords of a phrase share some centre pitch class, and between phrases this pitch class moves only by small intervals (a chromatic semitone or less).

Many more constraints where used (e.g., rhythmic constrains controlling the position of durational accents (Anders 2014)), but space limitations prevent discussing these in more detail.

The relatively large pitch domains caused by the microtonal tuning system did not pose any performance problem, because constraint propagation drastically reduced these domain sizes.

5. Constraint-Based Orchestration

This section discusses orchestration using constraint programming.

5.1. second horizon *by Johannes Kretz*

The solver OMCS (an OpenMusic port of PMC, see above) allows for arbitrary data types as domain values, including symbols, e.g., instrument names. Johannes Kretz (2006) used this feature to automatically orchestrate sections of the composition *second horizon* for piano and orchestra (2002).

Notes of the piano part were pointillistically spread across other parts. Only few simple constraints were applied. The range of instruments had to be respected. For a better blend, only instruments that belong to predefined families could be used together in a chord. Here, the families where simply woodwinds versus brass (strings where composed separately).

Also, the allocation of instruments in chords corresponded always to some standard arrangement. For example, if flute and oboe were playing together, the flute's pitch was always above the oboe. The purpose was to rule out unusual combinations (e.g., the bassoon above the oboe).

While these constraints are obviously over-simplifying the subtleties of orchestration, they demonstrate the flexibility of PMC. To be clear, constraint problems on symbols can also be modelled with integer variables by mapping every symbol uniquely to an integer, but the resulting definition is somewhat less intuitive.

Constraint programming has also been used for other tasks in the composition process of this piece (e.g., other interesting uses of heuristic profile constraints), but space limitations make it impossible to discuss these here.

5.2. Searching for Orchestrations that Imitate a Target Sound: Orchidée

The orchestration environment Orchidée (Carpentier and Bresson 2010) is very different from the other systems presented here so far. It is designed for supporting the orchestration process only, but in contrast to the purely symbolic systems discussed so far it also depends on signal processing.

The basic idea of Orchidée is that users state an orchestra (a set of instruments), and a target sound (typically a recording, but also synthesised sounds are possible). The system then searches for orchestrations (a mix of instruments from the orchestra alongside with dynamics, and playing techniques) that imitate the target sound. Orchidée is currently limited to static sounds: it can only compute individual orchestration "time slices".

The system depends on multiple sound features (spectral centroid, attack time, and so forth) automatically extracted from the target sound, and from a database of orchestral sound samples. It compares the perceptual similarity of multiple features of the target sound, and the combined features of mixtures of orchestra samples. During a search process based on an evolutionary algorithm this similarity is maximised (Carpentier, Assayag, and Saint-James 2010).

Users can further restrict the solutions with symbolic constraints. For example, users can state the maximal number of instruments to be involved, certain pitches that should be played, and what the minimal dynamics should be.

Constraint problems in general can have multiple solutions, and it is important for users to explore different solutions. In the case of Orchidée this is particularly important: different solutions may imitate different timbral features of the target sound more closely (e.g., one solution imitates better the attack, and another the spectrum), but there may be no single ideal solution. Also, Orchidée often results in highly unconventional solutions, so different solutions should be considered. The system therefore provides graphical tools for exploring the solution space.

5.3. Speakings *by Jonathan Harvey*

In the composition *Speakings* for large orchestra and electronics (2008), the orchestra imitates certain aspects of speech sounds, derived from, for example, baby babbles, radio interviews and poetry readings (Nouno et al. 2009). Jonathan Harvey and his team of assistants from IRCAM (Paris) achieved such imitations on the one hand with certain orchestrations using Orchidée, and on the other hand with real-time sound processing techniques based on analysis/resynthesis.

Orchidée was used, for example, to imitate the sound (formats) of a simple three-note mantra (sung

by the composer). The mantra is repeated many times, and the orchestration progressively evolves. Over time, the resulting sound becomes louder, brighter and also closer to the recorded mantra, the target sound. Such a progression has been realised by changing symbolic constraints, and by tweaking the sound features taken into account.

6. Discussion

This chapter presented how several composers have used constraint programming for their pieces. Instrumentations ranged from chamber music to orchestral works. The constraint systems used during the composition process have always been introduced first, so that their use could be discussed. The systems differ in what kind of constraints they allow for (e.g., searching for only pitches, or also rhythmic values), which is reflected in their use.

6.1. Similarities and Differences in Compositional Uses

While composers and pieces discussed here differ greatly, they share some common tendencies. Local pitch score contexts are commonly constrained, such as the pitches of simultaneous notes or the pitches of two or more consecutive notes in a melody. For example, melodic intervals are commonly constrained, and so are repetitions (both in various ways). Also, traditional music theory rules form an important inspiration, though the actual constraints and results commonly twist tradition.

Composers obviously aim for a personal voice, and their use of constraint programming thus clearly differs. For example, some composers aim for characteristic melodic interval successions. However, some differences are more structural.

Rhythmic constraints are less common so far, on the one hand because some systems do not support them, but possibly also because traditional music theory largely neglects rhythm. Composers who do use rhythmic constraints tend to develop their own music theories for that purpose (e.g., Sandred's metrical hierarchy and Anders' durational accents). While we can describe rhythms and their transformation, we seemingly lack widely accepted concepts for rhythmic rules.

Shaping the musical form is a central concern for composers. Diverse approaches have been proposed for controlling musical form with constraints, including heuristic profile constraints, constrained-based reactions to certain musical events, and several motif models. Most composers discussed here used motifs (i.e., groups of pitch and/or rhythmic values), but otherwise their means for controlling form clearly differ.

Note that the proposed approaches primarily address the formal development within sections. All composers discussed here controlled the global form manually, and used constraint programming only for generating sections (some composers even only very short sections, e.g., lasting only a single bar).

6.2. Strengths of Constraint Programming

When compared with other algorithmic composition methods, constraint programming has particular strengths. Perhaps most importantly, constraint programming can easily control multiple contexts of a single compositional parameter. For example, it is easy to control melodic intervals within a single part and harmonic relations between parts, where both constraints affect the same pitches. With other algorithmic composition methods such control is much more difficult.

Constraints are declarative and modular, like music theory textbook rules are declarative and modular. Each constraint only describes how the result should look, but not how this result is reached. Also, each constraint is independent from other constraints, and only describes one aspect of the result. Such strengths allow users to model highly complex compositional theories, including

traditional theories or theories inspired by tradition, which is more difficult to do by other approaches that do not use search.

Constraint programming also suits the mindset of certain composers well, in particular those with classical training. For them it is rather natural to think in constraints, due to the importance rules have in their training. Further, constraint programming allows for a close combination of manual and algorithmic composition, which gives composers great flexibility to shape the result, as all variables in the music representation can be either constrained or manually set (or both) (Anders and Miranda 2009).

6.3. Challenges

There are also noteworthy challenges connected with constraint programming. While the idea of constraints is easy to understand for composers, actually implementing new constraints can be difficult, as the desired outcome must be described exactly and in detail.

Some constraint systems (e.g., PWMC, Strasheela, and Cluster Engine) allow speeding up the search process for specific constraint problems with custom search strategies. However, programming custom search strategies requires the expertise of an expert. These systems therefore offer a default search strategy, or provide users with predefined search strategies to select from for different categories of problems.

Constraint programs in general are hard to debug. Programming environments offer only limited help. Instead, users have to carefully analyse their programs and results. Contradictions between constraints can happen, and in case of hard constraints (e.g., no heuristics) this leads to a fail (no solution). Such contradictions can be difficult to find in a large set of possibly complex constraints. A somewhat crude by effective way around starts by disabling all constraints, and then enables them again by and by to identify the culprit.

Stochastic approaches have an important place in algorithmic composition and music theory modelling. Constraint programming is good at enforcing strict relations, but probabilities cause difficulties. Several approaches for stochastic constraint programming have been proposed in the general constraint literature (Rendl, Tack, and Stuckey 2014): these approaches distinguish between stochastic variables that follow some random distribution, and decision variables that can be constrained – constraints between stochastic variables cannot be directly enforced. More promising is the approach of Sandred at al. (2009), where a stochastic constraint ensures a random distributions between variables (it depends on the order in which the solver visits variables, in contrast to the previous approach). They used this approach to create new renderings of the 4th movement of the *Illiac Suite* (1956), which follows Hiller and Isaacson's (1993) Markov chain probability tables complemented by constraints on harmony and voice leading.

The music constraint systems presented above cannot be used in realtime (or only in a limited way), but realtime-support likely comes in future. Constraint propagation (without search) has already been used for a long time in realtime applications. For example, MidiSpace (Pachet and Delerue 1998) users can control music spatialisation and mixing in realtime, and constraints arrange for mixing consistency. Reasonably simple search problems can be solved within milliseconds today. Constraint problems can quasi react to user input on the fly, when the problems are cut into time slices, as a pilot demonstrated for interactive first-species counterpoint (Anders and Miranda 2008). The speed of constraint solvers and computers increased further since then. Missing is still the integration of high performance solvers in realtime composition environments.

In future, constraint programming is possibly used more often for composing directly with sound. Aucouturier and Pachet (2006) proposed a system – also running in realtime – for concatenating audio segments (samples), where constraints shape the result by restricting metadata of samples.

This approach has been used, for instance, for an interactive drum machine that reacts to a MIDI live performance. Orchids (Esling and Bouchereau 2014) – the successor of Orchideé (discussed above) – meanwhile supports finding dynamically evolving sounds; besides for orchestration it is therefore also interesting for sound design, and electroacoustic composition.

7. References

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

Anders, T. 2009. "A Model of Musical Motifs." In *Mathematics and Computation in Music 2007, MCM 2007*, edited by T. Klouche and T. Noll, 37:52–58. CCIS 37. Berlin: Springer.

Anders, T. 2011. "Variable Orderings for Solving Musical Constraint Satisfaction Problems." In *Constraint Programming in Music*, edited by Gerard Assayag and Charlotte Truchet, 25–54. London: Wiley.

Anders, T. 2014. "Modelling Durational Accents for Computer-Aided Composition." In *Proceedings of the 9th Conference on Interdisciplinary Musicology – CIM14*. Berlin, Germany.

Anders, T., and E. R. Miranda. 2008. "Constraint-Based Composition in Realtime." In *Proceedings of the 2008 International Computer Music Conference*. Belfast, UK.

Anders, T., and E. R. Miranda. 2009. "Interfacing Manual and Machine Composition." *Contemporary Music Review* 28 (2): 133–47.

Anders, T., and E. R. Miranda. 2010a. "A Computational Model for Rule-Based Microtonal Music Theories and Composition." *Perspectives of New Music* 48 (2): 47–77.

Anders, T., and E. R. Miranda. 2010b. "Constraint Application with Higher-Order Programming for Modeling Music Theories." *Computer Music Journal* 34 (2): 25–38.

Anders, T., and E. R. Miranda. 2011. "Constraint Programming Systems for Modeling Music Theories and Composition." *ACM Computing Surveys* 43 (4): 30:1–30:38.

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

Apt, K. R., and M. G. Wallace. 2007. *Constraint Logic Programming Using Eclipse*. Cambridge: Cambridge University Press.

Assayag, G., C. Rueda, M. Laurson, C. Agon, and O. Delerue. 1999. "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic." *Computer Music Journal* 23 (3): 59–72.

Aucouturier, J.-J., and F. Pachet. 2006. "Jamming With Plunderphonics: Interactive Concatenative Synthesis Of Music." *Journal of New Music Research* 35 (1): 35–50.

Barbour, J. M. 2004. *Tuning and Temperament*. Mineola, NY: Dover Publications.

Beek, P. van. 2006. "Backtracking Search Algorithms." In *Handbook of Constraint Programming*, edited by Francesca Rossi, Peter van Beek, and Toby Walsh, 85–134. Amsterdam: Elsevier.

Carlsson, M. 2014. "SICStus Prolog User's Manual 4.3." SICS Swedish ICT AB. https://sicstus.sics.se/documentation.html.

Carpentier, G., G. Assayag, and E. Saint-James. 2010. "Solving the Musical Orchestration Problem Using Multiobjective Constrained Optimization with a Genetic Local Search Approach." *Journal of Heuristics* 16 (5): 681–714.

Carpentier, G., and J. Bresson. 2010. "Interacting with Symbol, Sound, and Feature Spaces in Orchidée, a Computer-Aided Orchestration Environment." *Computer Music Journal* 34 (1): 10–27.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13 (1): 47–56.

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

Ebcioglu, K. 1980. "Computer Counterpoint." In *Proceedings of the International Computer Music Conference 1980*, 534–43. San Francisco: International Computer Music Association.

Ebcioglu, K. 1987. "Report on the CHORAL Project: An Expert System for Harmonizing Four-Part

Chorales.” Report 12628. IBM, Thomas J. Watson Research Center.
Erlich, P. 1998. “Tuning, Tonality, and Twenty-Two-Tone Temperament.” *Xenharmonikôn* 17. http://lumma.org/tuning/erlich/erlich-decatonic.pdf.
Esling, P., and A. Bouchereau. 2014. “ORCHIDS : Abstract and Temporal Orchestration Software.” Paris: IRCAM. http://repmus.ircam.fr/_media/esling/orchids-documentation.pdf.
Fernández, J. D., and F. Vico. 2013. “AI Methods in Algorithmic Composition: A Comprehensive Survey.” *Journal of Artificial Intelligence Research* 48: 513–82.
Fokker, A. D. 1955. “Equal Temperament and the Thirty-One-Keyed Organ.” *The Scientific Monthly* 81 (4): 161–66.
Fux, J. J. 1965. *The Study of Counterpoint. from Johann Joseph Fux’s Gradus Ad Parnassum*. London: W.W. Norton & Company.
Hiller, L., and L. Isaacson. 1993. “Musical Composition with a High-Speed Digital Computer.” In *Machine Models of Music*, edited by Stephan M. Schwanauer and David A. Lewitt, 9–21. Cambridge, MA: MIT press.
Kretz, J. 2006. “Navigation of Structured Material in Second Horizon for Piano and Orchestra.” In *The OM Composer’s Book – Vol. 1*, edited by Carlos Agon, Gérard Assayag, and Jean Bresson, 97–114. Editions Delatour France/Ircam-Centre Pompidou.
Laurson, M. 1996. “PATCHWORK: A Visual Programming Language and Some Musical Applications.” PhD thesis, Helsinki: Sibelius Academy.
Laurson, M., and M. Kuuskankare. 2001. “A Constraint Based Approach to Musical Textures and Instrumental Writing.” In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. Paphos, Cyprus.
Laurson, M., and M. Kuuskankare. 2005. “Extensible Constraint Syntax Through Score Accessors.” In *Journ’ees d’Informatique Musicale*. Paris.
Laurson, M., and M. Kuuskankare. 2009. “Two Computer-Assisted Composition Case Studies.” *Contemporary Music Review* 28 (2): 193–203.
Laurson, M., M. Kuuskankare, and V. Norilo. 2009. “An Overview of PWGL, a Visual Programming Environment for Music.” *Computer Music Journal* 33 (1): 19–31.
Meseguer, P., F. Rossi, and T. Schiex. 2006. “Soft Constraints.” In *Handbook of Constraint Programming*, edited by Francesca Rossi, Peter van Beek, and Toby Walsh, 281–328. Amsterdam: Elsevier.
Nethercote, N., P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. 2007. “Minizinc: Towards a Standard CP Modelling Language.” In *Principles and Practice of Constraint Programming–CP 2007*, 529–43. Berlin: Springer.
Nouno, G., A. Cont, G. Carpentier, J. Harvey, and others. 2009. “Making an Orchestra Speak.” In *Proceedings of the 6th Sound and Music Computing Conference*. Porto, Portugal.
Pachet, F., and O. Delerue. 1998. “MidiSpace: A Temporal Constraint-Based Music Spatializer.” In *ECAI 98 Workshop on Constraints for Artistic Applications*. Brighton, UK.
Pachet, F., and P. Roy. 2001. “Musical Harmonization with Constraints: A Survey.” *Constraints Journal* 6 (1): 7–19.
Pope, S. T., ed. 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press.
Rendl, A., G. Tack, and P. J. Stuckey. 2014. “Stochastic MiniZinc.” In *Principles and Practice of Constraint Programming*, edited by Barry O’Sullivan, 636–45. LNCS 8656. Cham, Switzerland: Springer.
Rothstein, J. 1995. *MIDI: A Comprehensive Introduction*. 2nd ed. Madison, WI: A-R Editions, Inc.
Roy, P. van, and S. Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA: MIT Press.
Sandred, Ö. 2000. *OMRC 1.1. A Library for Controlling Rhythm by Constraints*. 2nd ed. Paris: IRCAM.

Sandred, Ö. 2003. "Searching for a Rhythmical Language." In *PRISMA 01*. Milano: EuresisEdizioni.
Sandred, Ö. 2006. "'Kalejdoskop' for Clarinet, Viola and Piano." In *The OM Composer's Book – Vol. 1*, 223–35. Editions Delatour France/Ircam-Centre Pompidou.
Sandred, Ö. 2010. "PWMC, a Constraint-Solving System for Generating Music Scores." *Computer Music Journal* 34 (2): 8–24.
Sandred, Ö., M. Laurson, and M. Kuuskankare. 2009. "Revisiting the Illiac Suite—a Rule-Based Approach to Stochastic Processes." *Sonic Ideas/Ideas Sonicas* 2: 42–46.
Schilingi, J. B. 2009. "Local and Global Control in Computer-Aided Composition." *Contemporary Music Review* 28 (2): 181–91.
Schulte, C., G. Tack, and M. Z. Lagerkvist. 2015. "Modeling and Programming with Gecode." Tutorial. http://www.gecode.org/documentation.html.
Tack, G. 2009. "Constraint Propagation: Models, Techniques, Implementation." PhD thesis, Saarland University, Germany.
Taube, H. 2004. *Notes from the Metalevel*. London and New York: Taylor & Francis.
Tymoczko, D. 2011. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford: Oxford University Press.