Computer Music Journal

Volume 12, Number 3 Fall 1988

Contents

Reviews

PERFORMANCES, EXHIBITIONS, AND CONFERENCES

PUBLICATIONS

Kemal Ebcioğlu
IBM, Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598 USA

An Expert System for Harmonizing Four-part Chorales

Introduction

In this paper, we report on a rule-based expert system called CHORAL, for harmonization and Schenkerian analysis of chorales in the style of J. S. Bach. We first briefly compare our approach with some current trends in algorithmic composition and music analysis, and we then describe the CHORAL system itself.

Overview of Current Approaches to Algorithmic Music

Quite a few trends in algorithmic composition today are based on a streamlined formalism, for example, in the form of random generation of note attributes using elegant statistical distributions (Xenakis 1971), terse and powerful formal grammars (Jones 1981), elegant mathematical models (Kendall 1981; Vaggione 1984), or generalizations of serial composition procedures (Laske 1981). The economy and elegance of the formal representation underlying these musical styles (which are not in the least less respectable than traditional styles of music), may often have an aesthetic appeal in and of themselves. On the other hand, traditional music and most of modern music, which are usually composed without a computer, do not seem to permit such economical representations. In the traditional style, the typical basic training the composer has to go through in harmony, strict counterpoint, fugue, and orchestration already imposes a certain minimal complexity on the amount of knowledge required to describe such a style.

Also, many will agree that a similar complexity can be observed in the works of modern "noncomputer" composers like Karlheinz Stockhausen, Pierre Boulez, György Ligeti, Jan Rychlik, and Steve Reich (his later compositions). It seems that musical composition is a hard mental task that requires a substantial amount of knowledge, and any serious attempt to simulate "noncomputer" music composition on the computer would have to face the task of constructing a formal model of considerable complexity. We have found that even the algorithmic representation of the knowledge underlying the seemingly simple Bach chorale style is a task that already borders the intractable.

As for the music analysis field, the prevailing trends seem to emphasize selective and unobvious properties of music; for example, a golden section in some motets of Dufay (Sandresky 1981), or a surprising log-normal distribution in the dissonances within some chorales of Bach (Knopoff and Hutchinson 1981). Even the analysis approaches that capture a profound structure in tonal music, such as reduction techniques (Schenker 1979; Lerdahl and Jackendoff 1983), are still based on finding a selective property (the property that the piece has a plausible parsing). These properties, although interesting in their own right, do not constitute a satisfactory explanation of the music in question, in the sense that there exist many "pieces" that have these properties, but that have no relationship with the style.

An alternative approach that would perhaps provide a more satisfactory understanding of the music, is to attempt to algorithmically generate pieces in the same approximate style. There have already been some attempts at this analysis by synthesis approach to music in some simpler traditional styles, such as folk melodies (Zaripov 1960), or the

This research was supported by NSF grant DCR-8316665, and a major portion of it was done at the Department of Computer Science, S.U.N.Y. at Buffalo, under the direction of my advisor, the late Prof. John Myhill. A preliminary version of this paper appeared in the *Proceedings of the 1986 International Computer Music Conference* (San Francisco: Computer Music Association).

Computer Music Journal, Vol. 12, No. 3, Fall 1988,

first two phrases of Bach chorale melodies (Baroni and Jacoboni 1976). But there are two problems associated with extending this approach to more substantial traditional styles. First, it may be difficult to prevent the designer of such a resynthesis algorithm from introducing traits that would distort the style in an unscholarly fashion (this unscholarliness may be trivially removed by resynthesizing only the original pieces of the composer in some interesting way—but we feel that the approach of synthesizing new pieces is also worthwhile). The second, and more fundamental problem is that, although there has been good progress in the automated synthesis of sound (Mathews et al. 1969; Roads and Strawn 1985), the automated composition of nontrivial tonal music is to date not sufficiently understood (and is still regarded with some suspicion in the traditional circles). The present research is an attempt to further our understanding of the mechanical generation of music, by extending the analysis by synthesis approach to a more complex style, the style of the Bach chorale harmonizations. To cope with the complexity of the problem, we have developed a rule-based approach inspired from recent research in artificial intelligence, as well as from a heuristic search method that we had used in an earlier, smaller-scale strict counterpoint program (Ebcioğlu 1979, 1980).

BSL: An Efficient Logic Programming Language

Perhaps because of its inherent difficulty, the generation of nontrivial tonal music appears to require large computational resources. In typical computing environments, artificial intelligence (AI) languages such as Prolog or Lisp tend to be too slow for implementing our particular approach to the algorithmic generation of music. More efficient languages such as C or Fortran are not viable alternatives, since they in turn tend to be too low-level for the task of coding a large music expert system. At the outset of our research, we decided to represent musical knowledge using first-order predicate calculus; and in order to cope with the large computational requirements of tonal music generation, we have designed BSL (Backtracking Specification Language) (Ebcioğlu 1986, 1987a, 1987b, 1988), a new and efficient logic programming language that is fundamentally different from Prolog. From the execution viewpoint, BSL is a nondeterministic Algol-class language where variables cannot be assigned more than once except in controlled contexts. However, BSL has a desirable relationship with first-order predicate calculus that makes it a new kind of logic programming language: namely, each BSL program corresponds to an assertion in first-order logic, and executing the BSL program amounts to proving the corresponding assertion. More precisely, the semantics of a BSL program F is defined via a ternary relation Ψ, such that $\Psi(F, \sigma, \tau)$ means program F leads to final state τ when started in initial state σ, where a state is a mapping from variable names to elements of a "computer" universe, consisting of integers, arrays, records, and so on. There is a simple mapping $\lambda u[u']$ that translates a BSL program to a formula of a first-order language, such that *if* a BSL program F terminates in some state σ, *then* the corresponding first order formula F' is true in σ (where the truth of a formula in a given state σ is evaluated in a fixed "computer" interpretation involving integers, arrays, records, and operations on these, after replacing any free variables x in the formula by $\sigma(x)$). Thus, successfully executing a BSL program without free variables amounts to constructively proving that the corresponding first-order sentence is true in the fixed "computer" interpretation, or in all models of a suitably axiomatized theory of integers, arrays, and records.

A formal and rigorous description of BSL, in a style inspired from (de Bakker 1979), can be found in (Ebcioğlu 1987a). To implement BSL on real computers, we wrote a compiler, in Lisp, that translates BSL programs into efficient backtracking programs in C. Except for a few C routines for reading in the melody and for graphics, the CHORAL expert system has been coded entirely in BSL.

A Knowledge Representation Technique for Music

An ambitious music expert system somehow has to deal with the problem of representing a large amount of complex musical knowledge. Even when one has avoided the approach of coding the rules

directly in a conventional language, and one has chosen to represent musical knowledge declaratively, in logic, the complexity of the knowledge base may still be far from being conquered. To represent substantial amounts of musical knowledge in first-order logic, it appears necessary to divide up the logical assertions into groups that observe the music from *multiple viewpoints*. Using a small and nonredundant set of primitive functions and predicates to represent music, although mathematically appealing, does not seem to be suitable for expressing all the required viewpoints of the music in a natural way. For example, a set of primitive functions {p,a,d}, that observe each voice as a linear sequence of notes, such that p(0,v), p(1,v), . . . , a(0,v), a(1,v), . . . , d(0,v), d(1,v), . . . are the pitches, accidentals and durations of the notes of voice v, would certainly be sufficient to describe a simple vocal piece, and would also be suitable for expressing horizontal, melodic relationships in each voice; but the same primitives would be somewhat clumsy for expressing vertical, harmonic relationships between voices, since the ith note of one voice obviously does not in general line up with the ith note of another voice. Similarly, a large-scale work may require several hierarchical viewpoints that may constitute successively refined plans of the piece. So we opted for a knowledge representation that used *multiple sets of logical primitives* to represent the different viewpoints of the music. Each set of primitives was deliberately made richer than required, by incorporating all the musical attributes that we felt we could need while writing rules.

The viewpoints used by our CHORAL system include one that observes the chord skeleton of the chorale, one that observes the individual melodic lines of the different voices, and another one that observes the Schenkerian voice leading within the descant and bass. The need for using multiple viewpoints of the "solution object" has arisen in expert systems in other domains as well: For example, the Hearsay-II speech understanding system (Erman et al. 1980) observed the input utterance as mutually consistent streams of syllables, words, and word sequences, and the *constraints* system of (Sussman and Steele 1980) used equivalent circuits for observing a given fragment of an electrical circuit from more than one viewpoint.

Implementing Multiple Points of View

We now describe one possible way of implementing multiple viewpoints of the music in BSL. In this method, which was used in the CHORAL system, each viewpoint is represented by a different data structure, typically an array of records (called the *solution array* of that viewpoint), which serves as a rich set of primitive pseudofunctions and predicates for that view. This multiple view paradigm has the following procedural aspect: It is convenient to visualize a separate process for each viewpoint, which incrementally constructs (assigns to) its solution array, in close interaction with other processes constructing their respective solution arrays. Each process executes a series of *generate-and-test steps*. At the nth generate-and-test step of a process, $n = 0, 1, \ldots$, a value is selected and assigned to the nth element of the solution array of the viewpoint, depending on the elements $0, \ldots, n-1$ of the same solution array, the currently assigned elements of the solution arrays of other viewpoints, and the program input. The processes behave somewhat like the multiple processes that are scheduled on a single hardware processor in a timesharing operating system. They are arranged in a round-robin scheduling chain. Each process (implemented as a BSL predicate), whenever it becomes active (is given the CPU), first attempts to execute zero or more generate-and-test steps until all of its inputs are exhausted, and then gets blocked, relinquishing the CPU to the next process in the chain. Among the processes, there is a specially designated *clock* process, which executes exactly one step when it is scheduled; all other processes depend on inputs produced by this process, and thus become blocked whenever they need an input that has not yet been produced by the clock process. Thus, each step of the clock process determines the total amount of work done in one complete trip around the round-robin scheduling chain. The process scheduling paradigm described here is *backtrackable*, as explained later.

Knowledge Base of the Viewpoints

The knowledge base of each viewpoint is expressed in three groups of logic assertions (BSL subformu-

las), which determine the way in which the nth generate-and-test step is executed. (1) *Production rules:* These are the formal analogs of the production rules that would be found in a production system for a generate-and-test application, such as Stefik's GA1 system, which solves a search problem in molecular genetics (Stefik 1978). The informal meaning of a production rule is "IF certain conditions are true about the partial solution (elements $0, \ldots, n-1$, and the already assigned attributes of element n) and external data structures, THEN a certain value can be added to the partial solution (assigned to a group of attributes of element n)." Their procedural effect is to generate the possible assignments to element n of the solution array. (2) *Constraints:* These side-effect-free subformulas assert absolute rules about elements $0, \ldots, n$ of the solution array, and external data structures. They have the procedural effect of rejecting certain assignments to element n of the solution array (this effect is also called *early pruning* in AI literature [Hayes-Roth, Waterman, and Lenat 1983], since it removes certain paths from the search tree that are guaranteed not to lead to any solution). (3) *Heuristics:* These side-effect-free subformulas assert desirable properties of the solution elements $0, \ldots, n$ and external data structures. They have the procedural effect of having certain assignments to element n of the solution array tried before others are. The purpose of the heuristics is to guide the search so that the solution first found is a good solution. Here is how the nth generate-and-test step of a process is executed: First, all possible assignments to the nth element of the solution array are generated via the production rules. If a candidate assignment does not comply with the constraints, it is thrown away, otherwise its *worth* is computed by summing the weights of the heuristics that it makes true (the heuristics are weighted by different powers of two, with the most important heuristic having the greatest weight, etc.). Assuming that at least one choice was found for the nth element, the generate-and-test step is then completed by assigning the best choice to the nth element of the solution array (with ties being resolved randomly). But at the same time, the current state of all the processes, and the list of remaining choices for this generate-and-test step, are pushed down; so that the current state of all the processes can later be recovered and the execution of the current process can be restarted, by choosing the next best alternative for this generate-and-test step. Later on, if an impasse is encountered, that is, some process fails to find any acceptable values for an element of its solution array, control returns to the most recent step among the history of the steps of all the processes, which is estimated to be responsible for the failure. This is not necessarily the immediately preceding step, which could be irrelevant to the failure; BSL uses an *intelligent backtracking* algorithm. Execution then continues with the next best choice at the step where the return has been made to (assuming that there is a remaining choice at this step. If there is none, further backtracking occurs).

The generate-and-test method described here is based on the idea of producing the solution incrementally, and backing up where necessary. An alternative search technique in the field of algorithmic music is to repetitively generate (with a non-backtracking algorithm) a new random solution and test it, until an acceptable solution is found (e.g., Baroni and Jacoboni 1976); but this latter technique is difficult to use when the acceptable solutions are extremely few in comparison to the generable solutions, which we feel is a common situation in complex styles of music. Generate-and-test is a basic search technique used in expert system design (Hayes-Roth, Waterman, and Lenat 1983). Studies on other relevant search techniques of artificial intelligence can be found (e.g., Nilsson 1971; Pearl 1983). This style of incorporating heuristics in a generate-and-test method for producing music, was used in our earlier strict counterpoint program (Ebcioğlu 1979, 1980) and was also independently used by B. Schottstaedt in another, larger-scale strict counterpoint program (Schottstaedt 1984). (Ames 1983) used a similar technique for generating music in a more contemporary contrapuntal style.

In certain cases a viewpoint may be completely dependent on another, that is, it may not introduce new choices on its own. In the case of such redundant views, it is possible to maintain several views (solution arrays) in a single process, provided that one master view is chosen to execute the process

step and comply with the paradigm. This can be done as follows: As soon as a possible assignment to the nth element of the solution array of the master viewpoint is generated via the production rules of the master viewpoint, the subordinate views are tentatively updated according to this new element of the master viewpoint. The subordinate views can advance by zero, one, or more than one steps, even though the master view advances by one step. Then the constraints and heuristics of the subordinate views are used in conjunction with the constraints and heuristics of the master view in order to determine the acceptability of the current tentative assignment to the nth element of the master view, and to compute how desirable this assignment is. Note that the subordinate views do not have production rules, since the new addition to the master view completely determines all attributes of the new element(s) of the subordinate views.

In the generate-and-test technique described here, the heuristics constitute the most crucial ingredient for obtaining musical results. It is known that absolute rules, such as those found in a treatise, are not sufficient for producing beautiful music. Composers use much additional knowledge (roughly termed as "talent") for choosing among the "correct" extensions of a partial composition at each stage of the compositional process. While our limited powers of introspection prevent us from replicating the thought process of such choices in an algorithm, we conjecture that a good algorithmic approximation can be obtained by using a large number of prioritized heuristics, or recommendations, based on style-specific musical knowledge. The heuristics used in our music generation algorithm help to prevent the search process from taking "correct" but unmusical paths (these paths could easily be followed if absolute rules and random search were used alone), and they guide the music in the preferred direction.

The Viewpoints of the CHORAL System

We are now in a position to discuss the knowledge models, or viewpoints of the CHORAL system. The CHORAL system knowledge base, which was developed over a period of several years, is based on our study of the Bach chorales (Terry 1964), our personal intuitions, and traditional theoretical treatises such as (Louis and Thuille 1906; Koechlin 1928). We will give here only a brief overview of the knowledge base of CHORAL, which is in reality very long and complex. The CHORAL system uses the backtrackable process scheduling technique described above to implement the following viewpoints of the chorale.

The *chord skeleton* view observes the chorale as a sequence of rhythmless chords and fermatas, with some unconventional symbols underneath them, indicating key and degree within key. This is the clock process, and produces one chord per step. The primitives of this view allow referencing attributes such as the pitch and accidental of a voice v of any chord n in the sequence of skeletal chords. Although some harmony treatises tend to omit rules about degree transitions (e.g., Dubois 1921), keeping track of the key and degree within key, and imposing careful rules for the transitions between the different degrees, were found to be necessary for maintaining a solid sense of tonality early during our research. Without key and degree information, progressions tend to sound "Gregorian." Similarly, implementing modulations turned out to be no simple matter, and a literal implementation of the definition of modulation as in a treatise was found to be too permissive, yielding unacceptable results. Instead, we have implemented a complex set of production rules for generating a set of style-specific modulating progressions, constraints for filtering out the unacceptable modulating progressions, and heuristics for choosing the desirable modulating progressions. In the chord skeleton view we place, for example, the production rules that enumerate the possible ways of modulating to a new key, constraints about the preparation and resolution of a seventh in a seventh chord, and heuristics that prefer "Bachian" cadences.

The *fill-in* view observes the chorale as four interacting automata that change states in lockstep, generating the actual notes of the chorale in the form of suspensions, passing tones, and similar ornamentations, depending on the underlying chord skeleton. For each voice v at fill-in step n,

the primitives allow referencing attributes of voice v at a weak eighth beat and an immediately following strong eighth beat, and the new state that voice v enters at fill-in step n (states are suspension, descending passing tone, and normal). At each of its steps, the fill-in view generates the cross product of all possible inessential notes (passing tones, neighbor notes, suspensions, other chorale-specific ornamentations) in all the voices, discards the unacceptable combinations of inessential notes, and selects the desirable combinations, via a complex set of production rules, constraints, and heuristics.

We felt that bold clashes of simultaneous inessential notes were indispensable for achieving the effect of a "Bachian" harmonic-melodic flow. The harmonization task would have been greatly simplified if we had avoided simultaneous inessential notes, but we felt that we then probably would not obtain music. Note that precise rules about simultaneous inessential notes were not at all readily available. The typical treatise on school harmony gives precise rules on severely restricted forms of simultaneous inessential notes (e.g., Bitsch 1957). In other traditional studies on passing notes in Bach, (e.g., Koechlin 1922; McHose 1947) authors tend to merely give examples of clashes of simultaneous inessential notes from Bach, which the talented music student will nevertheless digest in an unconscious way, but which are not of suitable precision for programming.

The fill-in view reads the chord skeleton output. In this view we place, for example, the production rules for enumerating the long list of possible inessential note patterns that enable the desirable bold clashes of passing tones, a constraint about not sounding the resolution of a suspension above the suspension, and a heuristic on following a suspension by another in the same voice (a Bachian cliché).

The *time-slice* view observes the chorale as a sequence of vertical time-slices, each of which has a duration of an eighth-note, and imposes harmonic constraints. This view is redundant with and subordinate to fill-in. The primitives of this view allow referencing attributes such as the pitch and accidental of a voice v at any time-slice i, and whether a new note of voice v is struck at that time-slice. In this view we place, for example, a constraint about consecutive octaves and fifths.

The *melodic string* view observes the sequence of individual notes of the different voices from a purely melodic standpoint. The primitives of this view allow referencing attributes such as the pitch and accidental of any note i of a voice v. The *merged melodic string* view is similar to the melodic string view, except that the repeated adjacent pitches are merged into a single note. The merged melodic string view was necessary for recognizing and advising against bad melodic patterns which are not alleviated even when there are repeating notes in the pattern. These views are also redundant with, and subordinate to fill-in. In these views we place, for example, a constraint about sevenths or ninths spanned in three notes, a heuristic about continuing an ongoing linear progression in a given voice, and some other highly difficult constraints for enforcing "melodic interest" in the inner voices. (The melodic interest constraint of the merged melodic string view indicates that in any voice, when a note has occurred as a *high corner* (is sandwiched between two notes of lower pitch), then it cannot occur as a high corner for at least two measures. Notes that occur in high-corner positions are perceived to be more salient than notes which are, e.g., in the middle of a linear progression. This rule prevents the monotony arising from the repetition of the same pitch in the salient high-corner positions.)

The *Schenkerian analysis* view is based on our formal theory of hierarchical voice leading, inspired from (Schenker 1979) and also from (Lerdahl and Jackendoff 1983). The core of this theory consists of a set of rewriting rules which, when repeatedly applied to a starting pattern, can generate the bass and descant lines of a chorale. The Schenkerian analysis view uses our rewriting rules to find separate parse trees for the bass and descant lines of the chorale, employing a bottom up parsing method, and using many heuristics for choosing (among the alternative possible actions at each parser step) the action that would hopefully lead to the musically most plausible parsing. Unlike Lerdahl and Jacken-

doff's theory, which is based on a hierarchy of individual musical events (e.g., chords, noteheads), our theory is based on a hierarchy of slurs, and is more in line with Schenker's theory. The discussion of our voice-leading theory is beyond the scope of this paper, and the details can be found in (Ebcioğlu 1987a).

The Schenkerian analysis view observes the chorale as the sequence of steps of two nondeterministic bottom-up parsers for the descant and bass. This view reads the fill-in output. The primitives of this view allow referencing the output symbols of a parser step *n*, the new state that is entered after executing step *n*, and the action on the stack at parser step *n*. The rules and heuristics of this view belong to a new paradigm of automated hierarchical music analysis, and do not correspond to any rules that would be found in a traditional music theory treatise. In this view we place, for example, the production rules that enumerate the possible parser actions that can be done in a given state, a constraint about the agreement between the fundamental line accidentals and the key of the chorale, and a heuristic for proper recognition of shallow occurrences of the Schenkerian D-C-B-C ending pattern.

The fill-in, time-slice, and melodic string views are embedded in the same process, with fill-in as the master view among them.

The order or scheduling of processes is cyclically: chord skeleton, fill-in, Schenker-bass, Schenker-descant. Each time chord skeleton is scheduled, it adds a new chord to the chorale, each time fill-in is scheduled, it fills in the available chords, and produces quarterbeats of the actual music until no more chords are available. Each time a Schenker process is scheduled, it executes parser steps until the parser input pointer is less than a lookahead window away from the end of the currently available notes for the descant or bass. (The lookahead window gradually grew bigger as our ideas evolved, and in the recent versions, for the sake of reducing module sizes, we have found it expedient to place the Schenker processes in a separate postprocessing program that reads its input from a file produced by the other views.) When a process does not have any available inputs to enable it to execute any steps when it is scheduled, it simply schedules the next process in the chain without doing anything. The chorale melody is given as input to the program.

Results and Conclusions

BSL and the CHORAL system are presently running on the IBM 3081-3090 computers at the IBM Thomas J. Watson Research Center, under CMS and Lisp/VM and the AT&T C compiler. The program takes as input an alphanumeric encoding of the chorale melody, and outputs the harmonization in conventional music notation, and the hierarchical voice-leading analysis in slur-and-notehead notation. The output can be viewed on a graphics screen or can be printed. The inputs typically take 3–30 min of IBM 3081 CPU time to get harmonized, but some chorales have taken several hours. Figures 1 and 2 show two output examples, harmonizations of chorales no. 286 and no. 75 (Terry 1964). In these harmonizations, the voices are not in the proper ranges; but the program writes the harmonizations in such a way that there exists a transposition interval that will bring them to the proper ranges. Note that the parallel fifths between the soprano and tenor accompanying the anticipation pattern in the soprano at the end of the second phrase of no. 75, are allowable in the Bach chorale style (see, e.g., no. 383 [Terry 1964]). Many more output examples, and the complete list of rules of the program, can be found in (Ebcioğlu 1987a). The program has presently been tested on over 70 chorale melodies, and has reached an acceptable level of competence in its harmonization capability; we can say that its competence approaches that of a talented student of music who has studied the Bach chorales. While the heuristically guided generate-and-test method described in this paper is not necessarily an accurate cognitive model of the human compositional process, it seems to work, and it seems to be capable of producing musical results. We hope that our techniques will be of use to researchers in algorithmic composition who may be seeking alternative approaches.

Fig. 1. Output example of the harmonization of J. S. Bach's Chorale no. 286.

Chorale no. 286

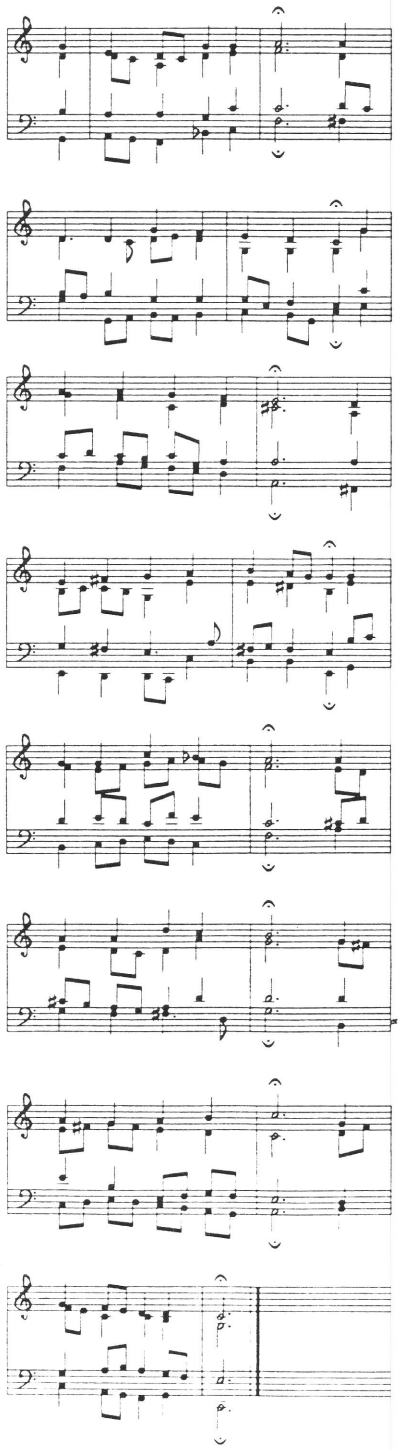
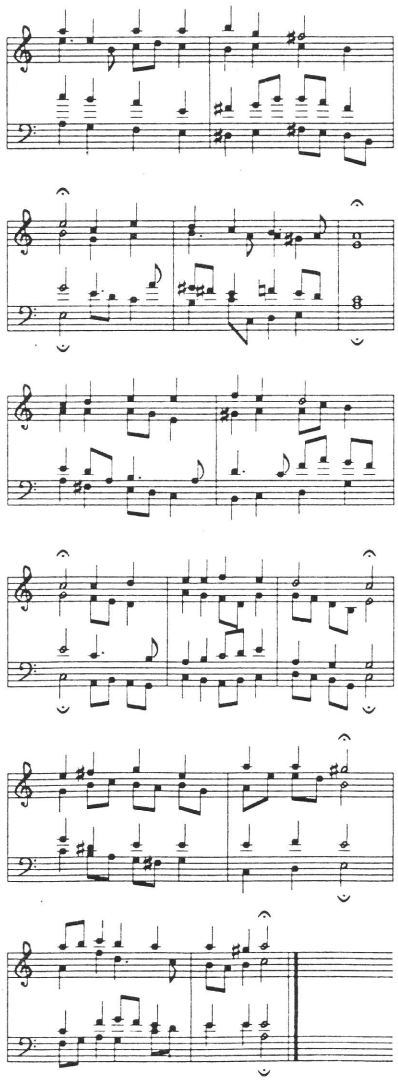


Fig. 2. Output example of the harmonization of J. S. Bach's Chorale no. 75.

Chorale no. 75



Acknowledgments

I am grateful to Prof. Lejaren Hiller for encouraging me to study computer music. I am also grateful to the late Prof. John Myhill, for getting me interested in the mechanization of Schenkerian analysis.

References

Ames, C. 1983. "Stylistic Automata in *Gradient*." *Computer Music Journal* 7(4):45–56.

Baroni, M., and C. Jacoboni. 1976. *Verso una Grammatica della Melodia*. Università Studi di Bologna.

Bitsch, M. 1957. *Précis d'Harmonie Tonale*. Paris: Alphonse Leduc.

de Bakker, J. 1979. *Mathematical Theory of Program Correctness*. Amsterdam: North Holland.

Dubois, T. 1921. *Traité d'Harmonie Théorique et Pratique*. Paris: Heugel.

Ebcioğlu, K. 1979. "Strict Counterpoint: A Case Study in Musical Composition by Computers." M. S. thesis (in English), Department of Computer Engineering. Ankara: Middle East Technical University.

Ebcioğlu, K. 1980. "Computer Counterpoint." In H. Howe, ed. *Proceedings of the 1980 International Computer Music Conference*. San Francisco: Computer Music Association.

Ebcioğlu, K. 1986. "An Expert System for Chorale Harmonization." *Proceedings of the Association for the Advancement of Artificial Intelligence*. Los Altos: Morgan-Kaufmann.

Ebcioğlu, K. 1987a. "Report on the CHORAL Project: An Expert System for Chorale Harmonization." Research report no. RC 12628, IBM, Thomas J. Watson Research Center, Yorktown Heights, March 1987. This is a revised version of the author's Ph.D. thesis (1986), "An Expert System for Harmonization of Chorales in the Style of J. S. Bach," Technical Report no. 86-09, Department of Computer Science, S.U.N.Y. at Buffalo.

Ebcioğlu, K. 1987b. "An Efficient Logic Programming Language and Its Application to Music." *Proceedings of the 4th International Conference on Logic Programming*. Cambridge, Mass.: MIT Press.

Ebcioğlu, K. 1988. "An Expert System for Chorale Harmonization in the Style of J. S. Bach." *Journal of Logic Programming*. To appear in the special issue on Applications of Logic Programming to Knowledge-based Systems.

Erman, L. D., et al. 1980. "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty." *Computing Surveys* 12(2):213–253.

Hayes-Roth, F., D. Waterman, and D. Lenat, eds. 1983. *Building Expert Systems*. Reading, Mass.: Addison-Wesley.

Jones, K. 1981. "Compositional Applications of Stochastic Processes." *Computer Music Journal* 5(2):45–61.

Kendall, G. S. 1981. "Composing from a Geometric Model: *Five-leaf Rose*." *Computer Music Journal* 5(4):66–73.

Knopoff, L., and L. Hutchinson. 1981. "Information Theory for Musical Continua." *Journal of Music Theory* 25(1).

Koechlin, C. 1922. *Étude sur les Notes de Passage*. Paris: Éditions Max Eschig.

Koechlin, C. 1928. *Traité de l'Harmonie*. Volumes I, II, III. Paris: Éditions Max Eschig.

Laske, O. 1981. "Composition Theory in Koenig's Project One and Project Two." *Computer Music Journal* 5(4):54–65.

Lerdahl, F., and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge, Mass.: MIT Press.

Louis, R., and L. Thuille. 1906. *Harmonielehre*. Stuttgart: C. Grüninger.

Mathews, M. V., et al. 1969. *The Technology of Computer Music*. Cambridge, Mass.: MIT Press.

McHose, A. I. 1947. *The Contrapuntal Harmonic Technique of the 18th Century*. Englewood Cliffs: Prentice-Hall.

Nilsson, N. J. 1971. *Problem Solving Methods in Artificial Intelligence*. New York: McGraw-Hill.

Pearl, J., ed. 1983. *Artificial Intelligence* 21. Special Issue on Search and Heuristics.

Roads, C., and J. Strawn, eds. 1985. *Foundations of Computer Music*. Cambridge, Mass.: MIT Press.

Sandresky, M. V. 1981. "The Golden Section in Three Byzantine Motets of Dufay." *Journal of Music Theory* 25(2).

Schenker, H. 1979. *Free Composition (Der freie Satz)*. Trans. and ed. Ernst Oster. New York: Longman.

Schottstaedt, B. 1984. "*Automatic Species Counterpoint*." CCRMA, Report no. STAN-M-19. Stanford: Department of Music, Stanford University.

Stefik, M. 1978. "Inferring DNA Structures from Segmentation Data." *Artificial Intelligence* 11:85–114.

Sussman, G. J., and G. Steele. 1980. "Constraints—A Language For Expressing Almost-Hierarchical Descriptions." *Artificial Intelligence* 14:1–39.

Terry, C. S., ed. 1964. *The Four-voice Chorales of J. S. Bach*. Oxford: Oxford University Press.

Vaggione, H. 1984. "Fractal C." *Program notes*, International Computer Music Conference, Paris, October 1984.

Xenakis, I. 1974. *Formalized Music*. Bloomington: Indiana University Press.

Zaripov, R. K. 1960. "On Algorithmic Expression of Musical Compositions." Doklady, Akademii Nauk, 132(6).