We describe our current progress with an expert system project for harmonizing four voice chorales in the style of Johann Sebastian Bach. An attempt has been made to describe the Bach Chorales style via approximately two hundred rules written in a form of first order predicate calculus. The rules are partitioned into groups which observe the chorale from multiple viewpoints, such as the chord skeleton, individual melodic lines, and hierarchical voice leading within the descant and bass. The program generates chorales from left to right, and performs intelligent backtracking until a solution, satisfying all the constraints, is found. A substantial number of heuristics are used for biasing the search toward musical solutions. BSL, a new and efficient logic programming language that compiles into C, was designed to implement the chorale program.

Nous décrivons le stade actuel d'un projet s'inscrivant d'un système expert conçu pour la réalisation des chorales dans le style de Johann Sebastian Bach. Nous avons essayé de décrire le style des chorales de Bach avec deux cent règles, en un programme, écrites en une forme de calcul de predi-
cats de premier ordre. Les règles sont reparties à plusieurs groupes, dont cha-
que observe le chorale d'un point de vue différent, comme la sémantique des accordes, les lignes mélodiques des parties indivi-
duelles, la structure, la hiérarchie du chant et la basse. Le programme construit 
d'une manière distribuée, employant une technique de recherche distribuée 
ed intelligent de backtracking. Il a été décidé de choisir une solution qui satis-
fait toutes les contraintes. Les règles hautes d'un nombre pour conçu pour les 
élaborer et de guider la recherche vers les solutions musicales. BSL, un langage 
qui est complété à la main, est utilisé pour l'élaboration de l'interprétation logique, qui se 
peut programmer. De telles règles ont été changées pour l'écriture du programme de chorales.

1. Introduction

In this paper, we will describe our current progress with a knowledge based expert system for generation and Schen-

kian analysis of chorales in the style of J.S. Bach. 

2. BSL: An Efficient Logic Programming 

Language for Implementing Expert Systems

Lisp, Prolog, and certain elegant software packages built on them, are known to be good languages for designing expert sys-

tems. Unfortunately, the inherent ineffi-
ciency of these languages has some ten-
dency to restrict their domain of applica-
tion to computationally small problems, whereas the problem of generating non-

trivial tonal music requires gigantic com-

putational resources. In order to imple-

ment the Bach chorales program, we designed 

BSL (Backtracking Specification Language), a new and efficient logic programming 

language. BSL programs are based on the 

direct compiled execution of a specially 
bartaborated formula taken from a subset of 

first order predicate calculus. The uni-

verse of interpretation is fixed as the 

set of integers and inductively defined 

formulas and records. A BSL program 

is operationally similar to a typical 

expert system based on production rules 

and the generate-and-test method: Through 

an efficient variant of backtracking, it 

attempts to find instantiations for the 

existentially quantified data structures 

in the formula, that would make the 

formula true in the fixed interpretation. 

Unlike existing logic programming 

languages, such as PROLOG [Kowalski 79], 

or LOOLISP [Robinson and Sibert 80], BSL 

is not a descendant of automatic inference 

methods (Chang and Lee 73). In particu-

lar, there are no concepts of inference or 

unification, instead, a formula is given 

separate non-deterministic program seman-

tics and logical semantics. Moreover, 

for a "pure" subset of BSL, these two are 

shown to be related in a denotational way 

(for formal details are in our forthcoming 

Ph.D. thesis). BSL programs also augment 

logical notation, by a capability of 

specifying a prioritized list of
heuristics, which are themselves formulas, which guide the choices made by the program while it incrementally searches for a solution. BSL programs compile into C for efficient execution.

To provide a feeling as to how BSL looks like, we give here an example of a BSL program solve a tiny puzzle. Place 8 queens on a chess board, so that no queen takes another. Assume that the rows and columns are numbered 0 to 7, and that the array elements p[0]... p[7] represent the position of the queen in row i \[0..7\], respectively.

(options registers \(k \ j \ n\)) (\((p (array \ (0 \ integer)))\)) (\((p \ j \ (< \ (+ \ 1 \ j))\)) (\((p \ j \ (> \ (+ \ k \ 1))\)) (\((k \ (+ \ k \ 1))\)) (\((p \ j \ (< \ (+ \ k \ 1))\)) (\((p \ j \ (> \ (+ \ k \ 1))\)) (\((k \ (+ \ k \ 1))\))

As it can be readily seen, this formula specifies what a solution to the eight queen problem should satisfy (assuming we read an assignment symbol as ordinary equality and translate the quantifiers to a conventional notation). This BSL formula compiles into a backtracking program in C that finds and prints instantiations for the array \(p\) that would make the formula true. The register declarations shown in the option list are passed to C, and the C compiler to place the quantifiers induces \(k\), \(n\) in registers if possible, for faster execution. The BSL compiler is written in Frans Lisp.

3. The Knowledge Representation Techniques of the Chess Program

Representing knowledge about multiple viewpoints of a solution-object is a need that often arises in the design of complex expert systems [Erman et al., 86, Usman and Stolze 88]. The present expert system was no exception.

The chess program observes and constructs the chess program from several viewpoints at once. Each viewpoint is represented by a different data structure, that serves as a rich set of primitive pseudo functions and predicates for that view. Multiple viewpoints are implemented in BSL via the following paradigm: it is convenient to visualize a separate process for each viewpoint, which constructs that particular view of the situation. In close interaction with other processes constructing their respective views. A process typically operates in units of "step". The purpose of each step, is to assign acceptable values to the \(n\)th element of an array of records, depending on the values of the array elements \(\ldots \ n-1\), and external inputs, e.g. elements of external arrays of records whose values have been assigned by other processes. The processes, implemented as BSL predicates, which execute the process first attempts to execute zero or more steps until one of its registrants exhausted, and then schedules (calls) the next process in the chain with parameters that indicate how far each process has progressed in assigning values to its output arrays. The specially designated clock process attempts to execute exactly one step when it is scheduled, all other processes adjust their timing to this process. In case of extremely redundant views, it is possible to maintain several views in a single process and share heuristics and constraints, provided that the master view is chosen to execute the process step.

The Bach chorale program uses this backtracking process scheduling technique to implement the following viewpoints of the chorale. The chord skeleton view, which corresponds to the clock process, observes the chorale as a sequence of rhythmized chords and fermatas, with some unconventional symbols underneath them, indicating key and degree within key. The fill-in view observes the chorale as four interacting voices that change states in a stepwise fashion, generating the basic notes of the chorale in the form of suspensions, passing tones, etc. The chord pitch view is the underlying chorale skeleton. The melodic view observes the chorale as a succession of verticle time-aligned pitch each of which has a duration of a small unit (an eighth note). It imparts the harmonic constraints. The Schenkerian analysis view is based on our formal writing rules inspired first [Schenker 78, Lerdahl and Jackendoff 83]. The descent and bass views are parsed separately according to these rules. The Schenkerian analysis views observes the chorale as a sequence of steps of two non-deterministic bottom-up parsers for the descent and bass. This analysis view will be further discussed later in this paper.

The knowledge embodied in each viewpoint is implemented via three groups of subformulas that are listed below. Formal analogs of production rules: these subformulas have the informal meaning "if certain condition then certain values can be added as
the n'th element.\footnote{Constraints: these are subformulas that assert absolute rules about elements $\ldots n$ of the partial solution and external inputs, they have the procedural effect of rejecting certain assignments to element $n$. Heuristics: these subformulas assert what is desirable about elements $\ldots n$ of the partial solution and external inputs, they have the procedural effect of having certain assignments to element $n$ tried before others. A simple power-of-two weighting scheme is imposed on the heuristics, that determines the ordering of the candidates for element $n$.}

The chorale program attempts to alleviate the overhead associated with backtracking by a special compilation technique that associates a tag with every variable. At run time, the tag contains the stack level to backtrack to in order to get a different choice for the value of the corresponding variable. When a step of a process cannot be executed, the tags of the variables that occur in the failing tests allow the program to backtrack to the most recent step suspected of being responsible for the failure, which is not necessarily the immediately preceding step. Similar techniques in AI are called dependency-directed backtracking or intelligent backtracking [Stalman and Susman 77].

There are currently a total number of approximately 200 rules and heuristics in the chorale program. The rules and heuristics were found mainly from empirical observation of the chorales and personal experience. For example, we used a number of traditional treatises (such as [Charles Rosen 71]) as an anachronistic, but not necessarily the immediately preceding step. Similar techniques in AI are called dependency-directed backtracking or intelligent backtracking [Stalman and Susman 77].

As a concrete example as to what type of knowledge is embodied in the program, and how such musical knowledge is expressed in logic, we take a constraint from the chorale skeleton view. The following subformula asserts a familiar constraint about false relations: "When two notes which have the same pitch number but different accidentals occur in two consecutive chords, but not in the same voice, then the second chord must be a diminished seventh, or the first inversion of a dominant seventh, and the bass must sound the sharpened note of the false relation, or the soprano must sound the flattened note of the false relation." (The exception where the bass sounds the sharpened note of the false relation is common-place, the less usual case where the soprano sounds the flattened note, can be seen in the chorale "Ave Maris Stella" no. 165 in [Terry 61].) We see the logical translation of this rule below.

(A v1) base (= v1 soprano) (= v1 v1) (A v2 base (= v2 soprano) (= v2 v1) (imp (and > n) (= (mod (p1 v1) 7) (mod (p2 v2) 7)) (or (and (p1 v1) (p2 v2)) (= v1 v2)) (and (member chordtype7 (=seventh domseventh)) (= v2 bass))) (= v2 soprano))}

Here, $n$ is the sequence number of the current chorale. (p1) and (p2) are the pitch of voice 1 or of chord $n-1$, encoded as 7-octave number-pitch name. (a v1), (a v2), $\ldots$ is the accidental of voice $v$ in chord $n-1$. chordtype1, $\ldots$ is the pitch configuration of chord $n$. The notation $p_1, p_2$, etc. is an abbreviation system, obtained by an encoding RSL "with" statement, that allows convenient and fast access to the most recent element of the array of records representing the chord skeleton view.

It is a known fact that absolute constraints are not by themselves sufficient for musical results: composers normally use much additional knowledge to guide their choices among the possible solutions. Our limited powers of introspection prevent us from exactly replicating the thought process of such choices in an algorithm; however, there exist various algorithmic approximations, based on large amounts of precise domain-specific heuristics, or preference rules that tend to give good results in practice [Lenat 76]. The chorale program's extensive body of heuristics, which are used for selecting the preferred choices among the list of possibilities at each step of the program. As an example, let us consider the possibility of possibilities at each step of the program. As an example, let us consider the possibility of possibilities at each step of the program.

Here the construct $(\text{Em q (p1 q2) (F q)}):$ is a macro which expands into a series of possibilities, such as $(\text{Em q (p1 q2) (F q)})$.

imp (and > n) $(\text{Em q (p1 q2) (F q)})$)

We would have liked to have absolute rules that would accept every chorale. However, attempts to do so, in the unwieldy proliferation of allowable, conditional violations, have failed. Moreover, there are cases where the attenuating condition for the violation is hard to
find (consider the 5ths by contrary motion in m. 8 of no. 18 in [Terry 64]). In certain cases, we therefore used our own judgement in deciding where to put the list of conditional violations. We remark, in passing, that there have been precedents in describing real styles with treatise rules (e.g. [Jeppesen 39] for real 16th century counterpoint).

4. A Formal Theory of Voice Leading

The Schenkerian analysis viewpoint of the chorale program is based on our formal theory of voice leading, inspired mostly from 'Free composition (der freie Satz)' of Schenker and partly from the theory of Lerdahl and Jackendoff. The core of our theory consists of a set of rewriting rule schemata. In our theory, unlike Lerdahl and Jackendoff, the descant and bass are analyzed separately, because we feel that there is no other way to capture their independent deep linear progressions. The parse trees obtained by repeated applications of these rules to a starting pattern until they can no longer be applied contain the sequence of pitches of the soprano (or bass) at its terminal nodes.

The separate trees for the descant and bass, plus a set of ordered pairs connecting the terminal nodes of these trees (analogous to Schenker's diagonal lines) constitute the analysis of the chorale. We give here a subset of the formal grammar. In these rewriting rules the construct (note x) is the only terminal symbol schema, and indicates an actual notehead of the final piece. The variable y,x range over diatonic pitches (piccaspaced descending octave no. + pitch name). The construct (x y) corresponds to an analytic slur between the noteheads for pitches x and y. The construct (lp x y) indicates an analytic slur over a linear progression leading from pitch x to pitch y. The starting pattern for the descant is seen to resemble the fundamental line of Schenker. In these rules, *x indicates one or more occurrences of x.

\[
\begin{align*}
(e \ x \ y) & \to \\
& (note \ y) \\
& (note \ x) \\
& (note \ x) \ x \ z \ \ (lp \ x \ y) \\
& (lp \ x \ z) \ (lp \ y \ z)
\end{align*}
\]

Constraints:

- second = abs(x-z) ≤ octave
- second = abs(y-x) ≤ octave

\[
\begin{align*}
(lp \ x \ y) & \to \\
& (s \ x \ z) \ x \ z \ x \ z \ \ (s \ y \ y) \ (s \ x \ y) \ \ (s \ y \ x) \\
& (s \ x \ z) \ x \ z \ x \ z \ \ (s \ y \ y) \ (s \ x \ y) \ \ (s \ y \ x)
\end{align*}
\]

Similarly, a pre-function (s d5) = (note a4) (lp d4 d5) stands for the elaboration of a slur connecting d5 and d5 with what Schenker would call a motion from an inner voice at leading to d5, as seen below.

Our voice leading theory consists of a few typical middleground elaborations of linear progression, motions from inner voice, neighbor notes, apogeeizations, and tonic-dominated-tonic patterns, which are nevertheless surprisingly sufficient for parsing the middleground of many chorales. But a complete and local formalization of Der freie Satz remains an open problem. In particular our theory does not accommodate the unidirectional nesting of analytic slurs in any given sample parsing. We also omitted any discussion that we thought would not lead to a more interesting parsing of the chorale, such as those background reductions that occur on a level earlier than the level in which the soprano and bass become independent.

The program's approach to Schenkerian 'synthesis' is to generate the chorale via local constraints and test it bottom-up for paratability, in parallel with generation. The alternate approach of top-down synthesis was deemed to be impractical because it involves making commitments at an early program stage without knowing what these commitments will exactly lead to, which can cause problems in meeting local constraints later on.

ICMC '84 Proceedings
The chorale parser, like the parser for computer languages, maintains a stack and sequences itself through a set of states. The purpose of the parsing algorithm for the case of the descent, is to reduce the descent line to a linear progression. Linear progressions can be shallow, as in a scalar motion, or they can be deep, with other notes getting in between the notes of the linear progression. The chorale parser operation can be explained by the following example: when the current input pitch fails to continue the the current linear progression (e.g. if it jumps), the parser may push down the current state, and enter a different state. When the expected continuation of the interrupted linear progression later appears, the stack may be popped, restoring the state that existed when the linear progression was interrupted, after drawing slurs (i.e. outputting nodes of the parse tree) close any linear progression that was in progress before the expected continuation was seen. At a given step there is usually more than one action to perform, each of which would potentially yield a different parse tree. The most plausible action is algorithmically chosen by means of prioritized heuristics, as in the rest of the chorale program view. These heuristics, unlike the grammar itself, do take regard of the melodic, rhythmic and harmonic context of the pitches. For example, one heuristic declares that if the note following the current note is an expectation of a linear progression that was pushed down, and if it is a high carrier in low voice, then it is endeavoring to reduce to a pop the stack in the current state. The implication of this heuristic to the melody of Jesus (Ich dehne dir), the line progression that started at g as has been interrupted. Above the arrow, d is encountered, which is a possible continuation of that interrupted progression. The heuristic says that it would be possible to really consider that d as the continuation and draw a slur from a to d, since the next note is a better continuation. The better parsing is shown here.

Certain absolute constraints on the parse tree are used for ruling out absurd analyses, e.g. the main linear progression must always align with the key of the piece. This would rule out analyzing a chorale such as Jesus mein Fels with a devastating octave progression, since the octave progression would be dissonant, not minor.

The parser has a method of dealing with the initial ascent and the unsupported stretch. In a way that does not require special treatment. The parser assumes that the descent is preceded by an imaginary pitch equal to a guess for the first structural pitch. Thus an initial ascent is like a motion from an inner voice. An initial guess that is too high, however, could mean the entire descending line to be parsed as part of an initial ascent, leading to backtracking later on.

5. Current State of the Project

The current version of the chorale program aims only to harmonise existing chorale melody, and assign an analysis to it. It currently takes 15-18 minutes of VM 17/88 cpu time to generate a chorale. Despite the efficient implementation that compiles the knowledge base into C. The outputs of the chorale program, including the parse trees in Schenkerian slur-and-notehead notation, are drawn on a graphics screen or are saved in a file for later plotting on a laser printer. A simple interactive interface is compiled into the program that can explain the choices at a given step of a view. The project is not yet complete. In particular we are still having problems in obtaining good base parseings. Nevertheless, at the end of this paper we present some recent output examples that can indicate how far we have got. The examples show an harmonisation of Chorale no. 22 from Terry 64), and the analysis of its descant line. The melody was compiled from critical sources. We presently feel that the harmonisation, although currently acceptable, may be improved toward a more loyal style. The figures in the analysis indicate the internal depth of the progressions in the melody line. The pitch is seen and all actions related to that pitch are completed. In Schenkerian analysis, the numbers might be taken to mean the lowest level that the pitch belongs to, where level numbers increase as we go from the background to the foreground. We can see an ending line, a fifth progression in this case, at those notes where the index is excepted for the final note, whose level is 2.

We stress to the music researcher that we do not seek to train a program that produces only all the beautiful chorales. The primary musical goal of the present research is to gain a more precise understanding of the chorales, and Schenkerian analysis as applied to them. Our approach is fundamentally different from the traditional, reductionist, approaches to analysis (that could search and find a golden section Landes's 81), or an elegant statistical distribution (Kapoff and Hutchinson 81) in music theory and our music generation methods are quite unlike the ones in the vogue, such as the
random selection techniques. Although the traditional passive approach to music analysis does have the scholarly safety of not "interfering with a master's music," we nevertheless feel that the knowledge acquired during an effort toward machine generation of a musical style could contribute new ideas to the analytic understanding of that style. It is thus hoped that the present research will inspire some new directions in analysis.

Acknowledgement: This specific project is currently being supported by National Science Foundation grant no. MCS-8316665, under the direction of Prof. John Myhill.

References:
Kowaleki, R. "Logic for Problem Solving" North Holland 1979.
