COMPOzE — Intention-based Music Composition through Constraint Programming

Martin Henz	Stefan Lauer	Detlev Zimmermann
Programming Systems Lab	Computer Science Department	Graduate School for Cognitive Science

University of Saarland
Im Stadtwald
D-66041 Saarbrücken, Germany
E-mail: {henz,lauer,detlev}@cs.uni-sb.de

Abstract

The goal of this work is to derive four-voice music pieces from given musical plans, which describe the harmonic flow and the intentions of a desired composition. We developed the experimentation platform COMPOzE for intention-based composition. COMPOzE is based on constraint programming over finite domains of integers. We argue that constraint programming provides a suitable technology for this task and that the libraries and tools available for the constraint programming system Oz effectively support the implementation of COMPOzE.

This work links the research areas of automatic music composition on one hand and finite domain constraint programming on the other, and contributes the tool COMPOzE, which practically demonstrates the potential of constraint programming to open up new areas of application for automatic music composition.

1 Introduction

The aim of this project is to build a system for the automatic composition of music.

Music experts are often skeptical about music which is autonomously composed by computers. We share this skepticism and therefore chose—instead of completely autonomous composition—the task of composition to accompany multimedia presentations as our application domain. In this context, music serves as acoustic background, and supports the intentions of the presentation using the appropriate musical effects.

The system developed in the project consists of two main modules: the *arrangement system* AARON [12] and the *composition system* COMPOzE. AARON derives a *musical plan* from the intentional and metric structure of a given presentation by proceeding in two steps. In the first step, it generates a vector of musical parameters describing in musical terms how the given intentions ought to be realized. In the second step, it generates from these musical parameters an harmonic progression. The harmonic progression fixes the metric, rhythmic and harmonic structure, but leaves open the harmonic elaboration and the melody.

COMPOzE derives concrete audible music (i.e. MIDI data) from a musical plan and an harmonic progression. In this presentation, we focus on the COMPOzE subsystem. COMPOzE produces a progression of four voice chords (soprano, alto, tenor, bass), which implements the musical plan generated by AARON and is in accordance with standard musical laws. To accomodate different musical tastes and to allow for easy tuning of the system, we want to open up the composition process to the user by giving her maximal flexibility in the choice of the musical laws. COMPOzE's graphical user interface allows the user to choose musical laws by direct manipulation. The composition process is visualized including its solutions. The user can listen to and compare the solutions by mouse click.

It turns out that this task of open intention-based composition can be elegantly described as a constraint satisfaction problem and efficiently implemented using the constraint programming (CP) system Oz. The main contribution of COMPOzE is to demonstrate that CP in general and Oz in particular provide an adequate computational framework for open intention-based music composition.

2 Musical Plan

The given musical plan consists on the one hand of an harmonic progression. An harmonic progression is a sequence of harmonic functions. As an example, consider:

$$T^{9}\ S^{6}\ D^{\substack{7\\4}}\ D^{7}\ \underset{7}{T}\ \underset{3}{s}\ D^{T}\ D^{7}\ T$$

Additionally, a basic tonal key is given in which the score is to be set. Throughout the paper, we assume that the basic tonal key is C major. Every harmonic function corresponds to a certain tonal scale and limits the pitches of the corresponding chord to three tones. The harmonic function *T* selects the *major tonic* scale of the basic key, and fixes the pitches of the chord to the first, third and fifth tone of this scale, in our case the tones c, e or g. Similarly, the harmonic functions *S* and *D* select the major *subdominant* and *dominant* scales, and the harmonic functions *t*, *s*, and *d* select the *minor* instead of the major tonal scales. An introduction to these basic notions of harmony is given in [1].

Additional attributes of the harmonic function may force disharmonic tones into the chord as indicated by numeric upper indices to the function symbols, mix harmonic functions as indicated by function symbols as upper indices, or fix the bass voice to a certain tone as indicated by the numerals below the symbols.

On the other hand, COMPOzE has access to the musical parameters generated by AARON (Section 1). This is necessary in order to deduce conditions for the movement of single voices. The following vector of musical parameters describes for instance the musical restrictions derived when preparing the music for the ending of a presentation context:

Rhetoric:	finish	Melody:	downwards
Tempo:	slow	Rhythm:	calm
Metric:	metrical	Harmony:	soothing

A typical parameter used by COMPOzE describes the course of the melody voice, in our case the soprano: "Melody: downwards" forces the last pitch of the soprano to be well below the first one, while all others must lie between those two.

3 Composition Rules

In addition to the musical plan, the generated sequence of chords needs to obey the rules of composition, most of which we adapted from standard textbooks on harmony [6, 4]. To explain the implementation of these rules, we are using the following two examples:

Crossing Prohibition: The voices within one chord may not cross, in a sense that a lower voice may not play a higher note than a higher voice. For example, the bass may not play a higher note than the tenor within a chord.

Jump Law: A jump of a voice from a chord to its neighbor that exceeds a given distance must be soothed in the following chord by a jump of one or two steps in the opposite direction.

4 Related Work

An interesting approach to the simulation of Jazz improvisations is described in [5]. The system takes as input a chord progression to which a melody is automatically improvised. Besides aesthetic criteria, there are no restrictions such as intentions given. The algorithm generates in the first step a certain *contour*. This contour represents the main shape of the melody and is derived by a regular grammar. In the second step the concrete pitches are chosen using constraints.

The expert system CHORAL [2] is a system for harmonizing chorals in the style of J.S. Bach. The input is a fixed soprano (melody) voice of a certain choral. The system's task is to generate a four voice score according to the harmonic rules of Bach's epoche (17th and 18th century) in traditional musical notation. The system uses a rule-based heuristic search with backtracking. The knowledge base contains 350 rules, which may be absolute or heuristic. The absolute rules represent conditions which must be strictly obeyed. The heuristic rules do not have to be strictly satisfied, but they are necessary to support composition decisions based on knowledge about the asthetic appearance of chorals.

In [7], a system is described that automatically derives a four voice score from a given melody, i.e. it composes the missing bass and middle voices. The basic musical knowledge needed for realizing the scores is represented in an object-oriented framework. The composition task is represented as a constraint system. The algorithm has two main steps. In the first step, constraints are applied which represent relations of intervals between notes of a single voice in neighboring chords. The second step concerns constraints which represent relations between notes of the four voices in each single chord. The search algorithm is based on backtracking.

5 The Composition Task as a Constraint Satisfaction Problem

The most suitable framework in which to formalize the composition task is provided by the theory of constraint satisfaction [11]. Cast into a constraint satisfaction problem (`csp`), the problem looks as follows:

- In general, there are $n * v$ *variables* where n is the number of chords in the sequence and v the number of tones in each chord. In our case, we have 4 voices in each chord, namely bass, tenor, alto and soprano. We name the variables as follows: B_i, T_i, A_i, S_i, where $i \in \{1 \dots n\}$.

- The *domain* for these variables is given by a range of playable pitches.

- The harmonic functions and the composition rules can be formulated as *constraints* ruling between one or several variables. For example, the crossing prohibition in Section 3 can be expressed by the following constraint:

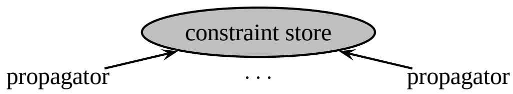
$$\forall_{i \in \{1 \ldots n\}} (B_i \leq T_i \leq A_i \leq S_i)$$

To solve this `csp` it turned out that approaches from Operations Research are not able to account for the variety of present constraints. Thus, we used the framework of CP [3]. Recent developments in CP culminated, among others, in the system Oz [10] that we use as implementation platform.

6 Constraint Programming

The goal of CP is to progressively restrict the set of possible values for variables using the given constraints, until finally, a unique value has been found for each variable.

The set of possible values is kept in the *constraint store*. For example, the fact that the base pitch of the first chord must be taken from the first 25 pitches of the scale is expressed by the constraint `B1` $\in \{0 \ldots 24\}$ in the constraint store. More complex constraints are expressed by *propagators* that observe the constraint store and amplify it if possible as depicted below.



7 Composition Rules as Propagators

A propagator inspects the store with respect to a fixed set of variables. When values are ruled out from the domain of one of these variables, it may add more information on others to the store, i.e., it may *amplify* the store by adding constraints to it. As an example consider the crossing prohibition in Section 3. It can be expressed for the first chord by installing the following three propagators:

```
B1 =<: T1   T1 =<: A1   A1 =<: S1
```

To explain how they can amplify the constraint store, let us assume that `A1` is constrained to $\{30 \ldots 45\}$, and `S1` to $\{25 \ldots 60\}$. Then the third propagator will exclude the values $25, \ldots, 29$ from the domain of `S1`, reducing its domain to $\{30 \ldots 60\}$. Vice versa, if later on it becomes known that `S1` $\in \{30 \ldots 40\}$, then `A1` will also be constrained by `A1` $\in \{30 \ldots 40\}$. Note that this propagator remains active, waiting for more information on either `A1` or `S1` to arrive. It only ceases to exist, when it becomes clear that it will never amplify the store again. Since it is not known in advance when the propagators will be able to perform their computation, they should be viewed as concurrent entities that observe the constraint store and amplify it whenever possible.

Similarly, we implement the Jump Law in Section 3 by a propagator that observes the distance between two neighboring tones of a voice. For example, the Jump Law is inforced on the bass voices of the first three chords by the following Oz program:

```
thread
   if {FD.distance B1 B2 ´>:´ JumpDistance}
   then {FD.distance B2 B3 ´=<´ 2}
      if B1 >: B2
      then B2 <: B3
      else B2 >: B3 end
   else true end
end
```

The conditional is moved to a concurrent thread of computation. As soon as the condition between **`if`** and **`then`** becomes logically implied by the constraint store, the conditional will reduce to the **`then`** part, which will emit a propagator for `FD.distance` and another conditional. If the negation of the condition becomes logically implied, the conditional just disappears, as indicated by **`true`** in the **`else`** part.

8 Search

Constraint propagation typically does not suffice to determine the values for all variables of the `csp`. Thus, after exhaustive propagation, a non-determined variable is speculatively constrained to one of its remaining values. This decision typically enables some propagators to exclude values for other variables. Thus the search space is continuously pruned while it is being explored. For a more detailed treatment of search and finite domain programming in Oz consider [9].

9 The Experimentation Platform COMPOzE

COMPOzE takes as input a musical plan, as given by the user or generated by AARON. The output of COMPOzE is one or several compositions that implement the given musical plan and fulfill user defined criteria. Figure 1 shows a snapshot of the current COMPOzE interface after configuration by a user. COMPOzE allows a user to decide for each musical law, if it should be be ignored (`off`), strictly obeyed (`hard`), or preferably obeyed (`soft`) with a user given weight from 0 through 100. The implementation uses a branch-and-bound technique to minimize the violation of the soft laws. If there is more than one soft law, their weight is used to determine their relative importance. The Oz Explorer [8] is used to visualize the search tree as shown in Figure 2. The user can interactively listen to a solution (realized by generating MIDI output) by clicking on the so-

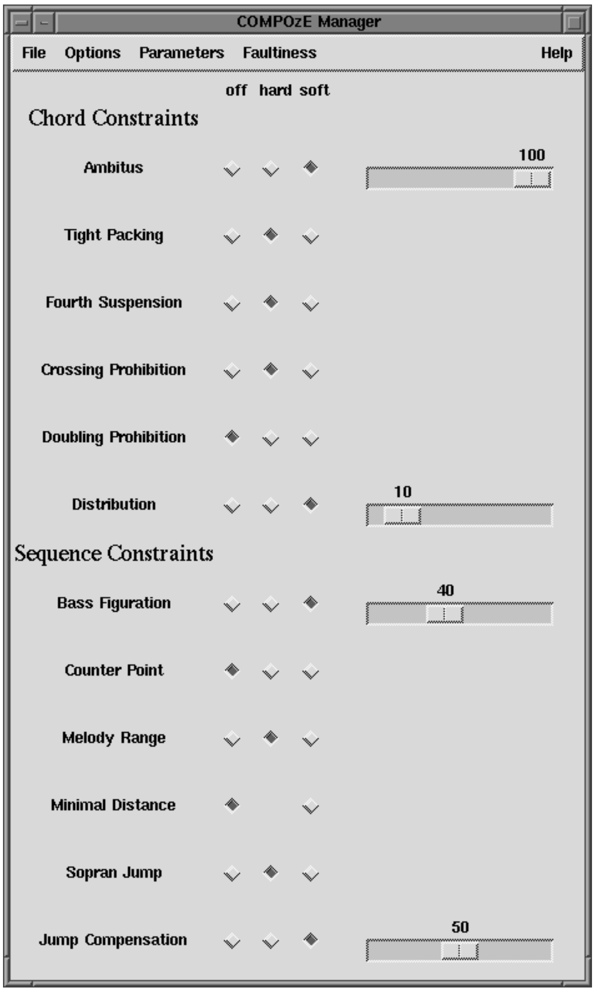


Figure 1. The COMPOzE Manager Window

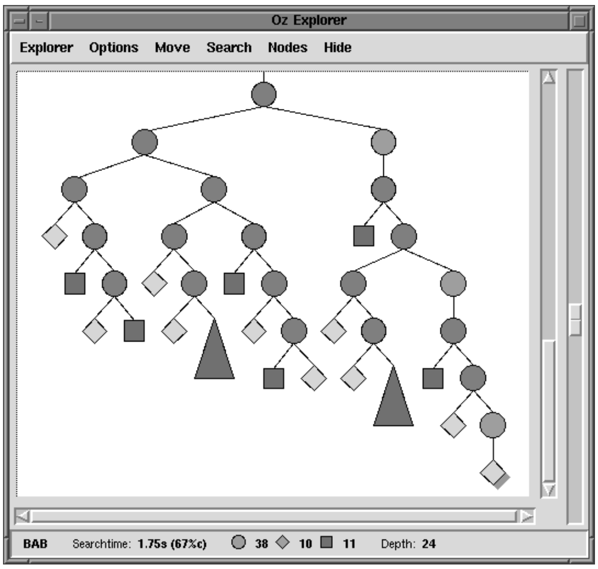


Figure 2. The Oz Explorer

lution nodes represented by diamonds in the Explorer and compare different solutions.

The current performance results are encouraging; with most user parameters and harmonic progressions of length 20 to 30, the system either shows that there is no solution or finds a first solution within one second of computation using a PC (Pentium 133Mhz). If there are soft constraints and the first solution was not optimal, the system usually finds several better solutions within another second. The resulting compositions are simplistic in style due to the rigid dynamic structure, but very pleasing from an harmonic and melodic point of view. Dynamics, instrumentation and percussion are subjects for further research.

References

[1] C. Dahlhaus. Harmony. In *The New Grove Dictionary of Music and Musicians*. Groves Dictionaries of Music, Washington D.C., 1980.

[2] K. Ebcioğlu. An expert system for harmonizing chorales in the style of J.S. Bach. In M. Balaban, K. Ebcioğlu, and O. Laske, editors, *Understanding Music with AI: Perspectives on Music Cognition*, The AAAI Press, pages 3–28. The MIT Press, Cambridge, Menlo Park, London, 1992.

[3] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[4] K. Jeppesen. *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*. Dover, 1992.

[5] P. Johnson-Laird. Jazz improvisation: A theory at the computational level. In P. Howell, R. West, and J. Cross, editors, *Representing Musical Structure*, pages 291–325. Academic Press, London, 1991.

[6] T. Krämer. *Harmonielehre im Selbststudium*. Breitkopf & Härtel, Wiesbaden, Germany, 1991.

[7] F. Pachet and P. Roy. Mixing constraints and objects: A case study in automatic harmonization. In *Proceedings of TOOLS Europe*, Versailles, 1995.

[8] C. Schulte. Oz Explorer: A visual constraint programming tool. ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/explorer96.ps.Z, 1996.

[9] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, LNCS, vol. 874, pages 134–150, Berlin, 1994. Springer-Verlag.

[10] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, LNCS, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[11] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, San Diego, CA, 1993.

[12] D. Zimmermann. Exploiting models of musical structure for automatic intention-based composition of background music. In G. Widmer, editor, *Proceedings of the IJCAI'95 Workshop on Artificial Intelligence and Music*. AAAI Press, Menlo Park, CA, 1995.