CONSTRAINT-BASED COMPOSITION IN REALTIME

Torsten Anders and Eduardo R. Miranda
Interdisciplinary Centre for Computer Music Research
University of Plymouth
{torsten.anders|eduardo.miranda}@plymouth.ac.uk

ABSTRACT

This paper proposes an approach for constraint-based algorithmic composition in realtime. To our knowledge, constraint programming – which performs a search – has not been used for music composition in realtime before.

The main contribution of this paper is a meta-solver with a timeout. We decompose the music creation process into one sub-constraint-problem and solver call per time step. The meta-solver is given a maximum search time: if no solution was found after the maximum search time elapsed, then the search is terminated. The user defines what to do in this case. That way, the system is never busy for too long and always responses to new realtime input.

This research extends our music constraint system Strasheela with realtime capabilities. Strasheela is based on the multi-paradigm programming language Oz. Our meta-solver exploits its paradigms of declarative concurrency (concurrent constraint programming plus first-class procedures) and constraint programming based on computation spaces.

1. INTRODUCTION

Realtime algorithmic composition attracts much interest, as can be seen by the success of systems where users define their own realtime algorithmic composition applications (e.g., Max [13] and SuperCollider [9]), and systems implementing specific composition techniques (e.g., the Real Time Composition Library – an extension for Max – by Karlheinz Essl). One could argue that algorithmic composition applications implement music theories which are then used for music generation or computing musical answers to realtime input. Implementing complex music theories often results in combinatorial problems. A declarative approach greatly simplifies the definition of combinatorial problems. A suitable technique for declaratively defining and efficiently solving combinatorial problems is constraint programming. This article studies the use of constraint programming for realtime algorithmic composition.

Constraint programming makes modelling complex problems simple. A constraint satisfaction problem (CSP) states *constraints* (mathematical relations) between *variables* (unknowns) for which a *domain* (a set of possible values) is defined. Changing the model is straightforward: the user adds, modifies, or deletes constraints.

Constraint programming has been successfully applied for implementing various music theories. A survey of constraint-based harmonisation systems is presented in [11]. Other examples include Fuxian counterpoint [17], Ligeti-like textures [6], and instrument-specific writing [8]. Established composers used constraint programming – for example, Magnus Lindberg (*Engine* for chamber orchestra, 1996) and Hans Tutschku (*Die Süsse unserer traurigen Kindheit*, music theater, 2005). However, in all these cases constraint programming was used in *non*-realtime.

To our knowledge, a general constraint solver which performs a search for solving combinatorial problems has not been used for music composition in realtime. Existing constraint-based systems with realtime support either used propagation algorithms which for efficiency reasons were tailored for the problem at hand, or they applied concurrent constraint programming without search (see below for details). However, computers have become fast enough to search in realtime. Also, the literature documents the demand for realtime constraint programming. For example, [21] and [24] mention constraint-based realtime composition as goals for future research.

The present research aims at a system where users can realise a wide range of musical CSPs in realtime. We extend the music constraint system Strasheela [3] with realtime capabilities, because this system is highly generic. Strasheela is build on top of the multi-paradigm programming language Oz [22].

Constraint systems present a few issues for realtime programming. Existing music constraint systems clearly differ from realtime composition systems. Music constraint systems search until they output a complete (section of a) piece of music, whereas realtime composition systems process and output music incrementally. Also, the search of a constraint system can fail – there may be no solution. Moreover, we cannot know in advance how long the search process will take: it may take too long for realtime output.

This research proposes a pragmatic approach which addresses these issues. Instead of calling a solver once for the full solution, we decompose the music creation process into one sub-CSP and solver call per time step. For example, with each realtime input event we may create and solve a new sub-CSP, or new sub-CSPs are triggered by some internal scheduler. The result is output immediately. Obviously, a musical segment cannot be undone (backtracked) after it is output. Therefore, some musical

CSPs become hard to define (e.g., CSPs with strict relations between past and future output, like a crab canon), but this is a common difference between composition and improvisation.

The main contribution of this paper is a meta-solver with a timeout. This solver is given a maximum search time: if no solution was found after the maximum search time elapsed, then the search is terminated. The user defines what to do in this case (e.g., do nothing, or select and play a pre-composed snippet). That way, the system is never busy for too long and always responses to new realtime input. The proposed solver is a meta-solver because it only adds the timeout functionality: the actual solver is given to the meta-solver as argument. This approach allows users to select a search strategy (e.g., a specific variable ordering) which solves their particular CSP efficiently. Oz supports concurrent constraint programming and first-class procedures, which are required for this meta-solver.

Paper Overview

The rest of this paper is organised as follows. Existing approaches which use constraints programming in realtime are discussed in Section 2. Section 3 provides background information on the Oz programming model and constraint programming in Oz. Our meta-solver is presented in Section 4, and demonstrated by an example in Section 5. The paper concludes with a discussion (Section 6).

2. RELATED WORK

Constraint programming has been used in reactive systems before, in particular for graphics-related applications. A survey of this field is provided in [7]. Pioneering systems are Sketchpad [20] and its successor ThingLab [5], which are both interactive drawing systems. The Cassowary constraint solving toolkit [4] has been used for user interface applications such as the Scheme Constraints Window Manager (Scwm). For efficiency, these constraint systems use constraint propagation algorithms which are tailored for the problem at hand. For example, Cassowary quickly solves linear equalities and inequalities incrementally: if the problem changes slightly (e.g., by user interaction), the system reuses as much of the previous solution as possible. On the other hand, these systems are not intended for solving complex combinatorial problems which would require searching, and therefore they are not suitable for implementing music theory models.

Concurrent constraint programming (*cc*, [16]) provides a declarative model for concurrent computations. It features a *store* for accumulating information on variables. The store is monotonic: information can be added, but not removed. Reactive systems, however, must deal with information with changes over time (e.g., new notes input to a realtime composition system). Timed concurrent constraint programming (*tcc*) therefore extends *cc* by an explicit representation of (discrete) time: at each time step, a *cc* program with its own store is executed. Various further extensions have been proposed and implemented with realtime support. For example, timed default concurrent constraint programming [15] adds support for timeouts (what happens if some information is missing at a time step). Its programs are compiled into a finite state automaton for efficient realtime performance. The *ntcc* calculus [12] additionally supports non-deterministic choice. This calculus has been used for multiple musical applications including the modelling of interactive scores [1], and the factor oracle model for music improvisation [14]. A realtime *ntcc* implementation in Lisp is described in [14]. However, these models cannot solve complex combinatorial problems which require a search either. They provide a single monotonic store per time step. Yet search performs speculative computations where information is withdrawn in case of a fail (e.g., with backtracking or multiple stores). [1]

Several constraint-based reactive systems have been proposed in the computer music field besides the *ntcc* calculus. MidiSpace [10] realises music spatialization and mixing in realtime. MidiSpace performs constraint propagation with an algorithm designed for its particular constraint problem. SuperCollider provides a simple constraints extension (in the `CrucialLibrary` by Chris Sattinger, aka crucial felix), for example, it can filter a SuperCollider pattern in realtime. Yet this library is also insufficient for solving combinatorial problems.

3. BACKGROUND

This section introduces some fundamentals of the Oz programming model. The proposed meta-solver relies on these fundamentals.

3.1. The Oz Kernel Language

Oz is based on a simple kernel language, which is a subset of the more convenient full language syntax. The kernel language supports multiple programming paradigms. We introduce a kernel language subset which supports two programming paradigms: concurrent constraint programming and functional programming (i.e., first-class procedures). This combination is called declarative concurrency in Oz terminology.

Figure 1 shows the syntax of this Oz kernel language subset. The statement `skip` does nothing. The sequential composition $S_1 S_2$ executes first the statement S_1 and then S_2. By contrast, a statement runs in its own concurrent thread if surrounded by the `thread`-keyword. The keyword `local` introduces fresh variables for the nested statement. Variables in Oz are logic variables, and they are lexically scoped. Information on variables is contained in a store (cf. *cc*). The statement `X=Y` tells Oz that the variables X and Y are equal (unified), while `X` $= v$ binds

[1] Combinatorial problems can be solved with *cc* in principle by specially deviced algorithms (e.g., [16] solves the N-queens problem in *cc*). Yet such algorithms are not as declarative as CSPs are.

X to the value v. The kernel language provides the `if`-conditional. The statement `proc {P X}` S `end` defines the procedure `P` with the argument `X` encapsulating the statement S. Finally, `{P Y}` applies the procedure `P` with `Y` as argument. Procedures are first-class values in Oz. Because procedure arguments are logic variables, they can be used freely both as input and return values.

S ::= `skip`	empty statement
$S_1\ S_2$	sequential composition
`thread` S `end`	thread introduction
`local X Y ... in` S `end`	variable introduction
`X = Y` | `X =` v	imposing equality (tell)
`if B then` S_1 `else` S_2 `end`	conditional
`proc {P X Y ...}` S `end`	procedure abstraction
`{P X Y ...}`	procedure application

Figure 1. The Oz kernel language syntax (subset used in this paper, cf. [22])

Please note that a thread blocks if some logic variable used in a statement of the thread lacks required information. For example `if X then 1 else 2 end` blocks until `X` is determined. Another thread can bind the variable: threads communicate via variables in Oz (dataflow variables).

3.2. Constraint Programming in Oz

This section outlines fundamental aspects of constraint programming in Oz which are necessary to understand the meta-solver proposed in the next section. For a detailed discussion see [19].

Constraint programming in Oz is based on computation spaces. Spaces extend the declarative concurrent programming model introduced above by search and thus make it possible to solve combinatorial problems – in contrast to *cc* and its timed variants. Spaces encapsulate speculative computations in local stores. During the search process, a tree of spaces with local stores is created and traversed.

Spaces make many aspects of the search process programmable at a high level. For solving a CSP, the user specifies the following three components: the actual CSP, a distribution strategy and an exploration strategy. The CSP expresses which constraints must hold between which variables in a solution. Constraints in Oz perform constraint propagation (i.e., narrow variable domains without search). The *distribution strategy* (branching strategy) defines in what order variables are visited during the search process, and how their domain is reduced – this defines a dynamic variable and value ordering, in other words, the shape of the search tree. The *exploration strategy* specifies how the search tree is traversed.

The CSP and distribution strategy are both wrapped in a *script*, which is a unary procedure whose argument is the solution. The exploration strategy is defined by a *constraint solver*, also a procedure, which expects the script as an argument.

Various predefined distribution strategies and exploration strategies are available. For example, a typical distribution strategy implements the first-fail variable ordering (i.e., visit the variables with smallest domains or most constraints applied first). An often used exploration strategy performs depth-first search. Special exploration strategies help finding a good solution with respect to some user-defined criterion, without traversing the full search tree. Others allow the user to visually guide the search process (e.g., with the Oz Explorer [18]), or support a parallel search on multiple processors or networked computers.

Moreover, users can define their own distribution and exploration strategies, and Oz provides high-level means for such definitions. The distribution strategy has great influence on the efficiency of the search process, and a suitable distribution strategy is problem-dependent (see [3] for a discussion of suitable variables ordering for various musical CSPs).

The fact that spaces encapsulate the search process has an important benefit for realtime music constraint programming: the system can solve a CSP and concurrently receive input and send output. Also, multiple search processes can run in parallel.

4. A META-SOLVER WITH A TIMEOUT

4.1. Using the Meta-Solver

This research proposes a meta-solver with support for a timeout as a basis for realtime constraint programming. This section explains its use.

The procedure `SearchWithTimeout` (the meta-solver) expects a script (the CSP for the current time step), a solver, and a maximum search time. It then returns the solution for the current time step. The following example allows for 10 msec search time (Figure 2).

```
local MaxSearchTime=10 in
  {SearchWithTimeout MyScript
   MySolver MaxSearchTime Solution}
end
```

Figure 2. Calling the meta-solver `SearchWithTimeout` with 10 mses search time

`SearchWithTimeout` returns the solution in a singleton list. If no solution is found after the maximum search time (or if the search failed), then `nil` is returned.

The script `MyScript` (Figure 3) defines a very simple CSP (it uses the syntax of full Oz). The solution is a list of two finite domain (FD) integers with the initial domain $\{1, \ldots, 10\}$ (line 2). The sum of both list elements is constrained to 7, and the first element must be smaller (lines 3–4). The distribution strategy visits the variables in the order of their sequence and selects their minimum domain value (naïve distribution, line 5). The first solution found be this distribution strategy is the list `[1 6]`.

```
proc {MyScript [X Y]}
   [X Y] = {FD.list 2 1#10}
   X + Y =: 7
   X <: Y
   {FD.distribute naive [X Y]}
end
```

Figure 3. An simple script example

The example solver `MySolver` performs a depth-first search. Oz users normally call a constraint solver explicitly, and different solvers often differ in their interface. `SearchWithTimeout`, however, calls the solver implicitly, and the solver must always have the following interface. The solver expects the script as input and returns two values, `KillP` and the solution. `KillP` is a null-ary procedure: calling `KillP` terminates the search of `MySolver`.

```
{MySolver MyScript KillP Solution}
```

Many build-in solvers in Oz already provide such a termination-procedure as argument. It can be implemented as follows. The kill-procedure solely determines a variable, but this variable is accessible by the solver. Whenever the next constraint distribution step is performed by the solver, the solver first checks whether its kill-procedure's variable is determined. In that case, the solver immediately returns `nil`, otherwise it continues searching. Consequently, only an actual search process is terminated by calling the kill-procedure, but a script which keeps computing forever without search (e.g., because it contains an infinite loop) cannot be terminated. The full implementation of a solver is beyond the scope of this paper (cf. [19]).

4.2. The Meta-Solver Definition

This section defines the meta-solver. `SearchWithTimeout` (Figure 4) creates two parallel threads. The first thread calls the given solver `MySolver` and waits for the solution (`Wait` blocks until `Solution` is determined). The second thread waits for `MaxSearchTime` milliseconds (`Delay` blocks for `MaxSearchTime`).

Both threads communicate via the variable `DoneFlag`. The first thread who finishes waiting finds `DoneFlag` still unbound (free). It then binds it and performs some additional action. The later thread finds `DoneFlag` determined (not free) and does nothing (`skip`). In case the solver-thread finishes waiting first, it binds the `Result` to its `Solution`. In case the other thread finishes waiting first, it terminates the solver (calls `KillSearchP`) and binds the `Result` to `nil`. The actual implementation differs slightly. For example, a lock makes the if-statements atomic and that way ensures that only one of the two threads can succeed in setting the `DoneFlag` and doing its action.

```
proc {SearchWithTimeout MyScript
      MySolver MaxSearchTime Result}
   local
      Solution KillSearchP DoneFlag
   in
      thread
         {MySolver MyScript
          KillSearchP Solution}
         {Wait Solution}
         if {IsFree DoneFlag}
         then DoneFlag = unit
              Result = Solution
         else skip
         end
      end
      thread
         {Delay MaxSearchTime}
         if {IsFree DoneFlag}
         then DoneFlag = unit
              {KillSearchP}
              Result = nil
         else skip
         end
      end
   end
end
```

Figure 4. Definition of the meta-solver with a timeout

Strasheela complements `SearchWithTimeout` by a class `ScoreSearcherWithTimeout` which adds further functionality to searching for Strasheela score objects. In particular, this class provides convenient access to realtime input, and previous solutions.

5. AN EXAMPLE

This section demonstrates the meta-solver in a simple example. The example implements a variant of first species counterpoint. Strasheela receives the notes of a melody from SuperCollider via Open Sound Control (OSC, [23]), for example, played on a keyboard or generated algorithmically. Strasheela considers this voice the cantus firmus and creates an appropriate counterpoint for it in realtime. Strasheela sends the counterpoint back to Supercollider (again via OSC), and Supercollider synthesizes both voices synchronously with a small latency (all notes are marked with an OSC timestamp).

The counterpoint is homophonic to the cantus firmus. However, in this example the cantus firmus is rhythmically free. The CSP implements the following rules for the counterpoint.

i) Harmonic rule: simultaneous notes must be consonant (perfect or imperfect consonance up to an octave plus a major third, but no unison).

ii) Melodic rule: only specific melodic intervals are allowed (anything between a minor second and a fourth,

or a fifth, or an octave). Pitch repetition is not permitted.

iii) Melodic rule: the generated melody is diatonic.

The implementation uses the class `ScoreSearcherWithTimeout`, because it simplifies the transfer of parameters such as realtime input and previous solutions to the sub-CSP of the current time step. The example defines a script – similar to a standard CSP – but this script expects additional arguments, for example, the realtime input and previous solutions. In addition, the example defines an OSC responder procedure which is called whenever Strasheela receives a new note (cf. OSC responders in SuperCollider). The responder procedure calls the meta-solver object (a `ScoreSearcherWithTimeout` instance) with arguments such as the script, the current input note, the distribution strategy, and the actual solver (arguments like the distribution strategy and the solver are optional for convenience). The responder then waits for the solution (or the timeout), and finally sends the solution via OSC. In the background, the meta-solver object creates the actual script for the sub-CSP of the current time step (which incorporates all its arguments) and calls the solver with that script. Figure 5 displays the structure of the example in a block diagram.

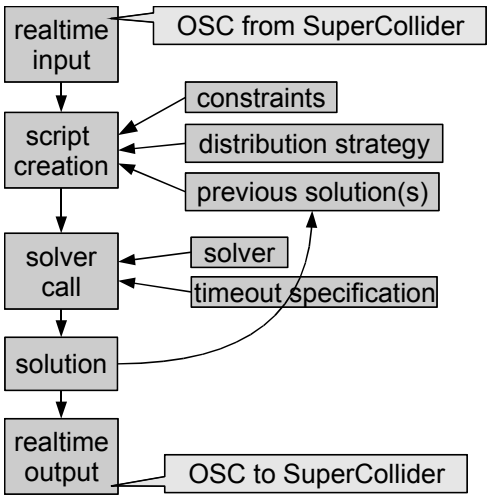

Figure 5. Realtime counterpoint example (block diagram)

This example is discussed in detail (including full source code and a resulting sound except) on the Strasheela website [2]. As the sub-CSPs only search for a single variable (the next note pitch), special distribution or exploration strategies are of little relevance. No note was dropped (no timeout) nor was any note late in this example with 10 msec maximum search time and 35 msec latency (for OSC network traffic) on a Macbook Pro 2.2GHz. 10 msec is the shortest sensible search time possible, because the Oz implementation Mozart uses hardware timer for thread preemption, and the time slice is 10 msec long. However, note that 10 msec are also sufficient to solve, for example, an all-interval series. Creating a full-scale first-species Fuxian counterpoint of 11 notes took always less then 60 msec (40 msec average).

For simplicity, this example has been implemented in a purely declarative way (i.e., without any stateful operation): it continuously generates lots of data which must be garbage collected. Nevertheless, given an appropriate latency, the output of the example is perfectly timed.

Garbage collection is indeed performed repeatedly while the example is running, and unlike Supercollider, Oz does *not* have a realtime garbage collector. In Oz, the program execution must be shortly interrupted for garbage collection. Nevertheless, Oz' garbage collection algorithm (copying dual-space algorithm) is fast for programs which require little active memory size. Moreover, larger applications can be split into multiple processes (using Oz' excellent support for distributed programming), where each process has its own local garbage collection. If the time-critical parts of an Oz application run in small processes, then their local garbage collection will be fast.

6. DISCUSSION

This research introduced an approach to music constraint programming in realtime which is useful for various applications areas. The presented example implemented a conventional music theory. However, the meta-solver is also applicable, for example, in generative systems processing input data (e.g., for sound installations), in accompaniment programs like band-in-a-box, for the dynamic tuning of user input (e.g., for just intonation), or constraint-based sound synthesis control as in MidiSpace.

The example constrained the relation between the current input event, the previous output, and the next output event by a set of constraints. Other applications can add further techniques. For example, buffering input data allows the user to play several notes before the system outputs an answer (e.g., triggered by a special event). Some aspects of the resulting musical form may be fixed beforehand (e.g., the underlying harmonic progression). The user may change the set of constraints in realtime: different sub-CSPs apply different constraint sets. Constraints encapsulated in first-class procedures can be script arguments like the realtime note input in the presented example. Applications may 'plan ahead' (e.g., create additional musical segments which are variations of the next output but played later). Also, applications may combine constraint programming with other algorithmic composition techniques. Finally, Strasheela already provides a rich toolbox for music constraint programming (e.g., an expressive music representation, constraint-based models of musical aspects such as harmony, motifs and meter, and various pattern constraints).

The main idea presented by this paper – calling constraint solvers with a timeout – might appear an obvious approach for constraint programming in realtime. However, implementing this idea calls for a programming model which combines concurrent programming and a constraint programming model which encapsulates the search process, like the space-based constraint model. Realtime constraint programming needs concurrent programming, because the search process and the timeout check must run

in parallel. Also, input/output for one time step may run concurrently with the search of the preceding/next time step. In addition, the search process must be encapsulated so it can run concurrently. By contrast, a solver which runs on the top-level such as in Prolog is unsuitable, because a top-level solver is incompatible with concurrent programming.

Moreover, the presented approach requires first-class procedures for flexibility. Non-trivial applications which feature user input or depend on previous output (as the counterpoint example) create the CSP script for each time step dynamically at runtime. This requires a first-class abstraction. Note that only few programming systems make it possible to combine first-class procedures, concurrent programming and constraint programming. The systems usually used for music constraint programming do not support concurrent programming. This might be the reason why realtime constraint programming has not been proposed before for music.

It can be unsatisfying that the meta-solver returns no result in case of a fail or a timeout. However, this research applies a complete search method and a fail only occurs if there exists no solution. A failed search is caused either by an over-constrained problem or a programming error. An over-constrained CSP can be weakened, for example, by applying less constraints or using soft constraints. Oz supports soft constraints, for example, via reified constraints (i.e., constraints whose validity can be constrained).

In case of a timeout, we do not output the current partial solution because it can be wrong (i.e., could lead to a fail). Timeouts can be avoided by optimising the search process. For example, finding a good distribution strategy can dramatically reduce the search tree. Also, redundant constraints may improve propagation and thus efficiency. Alternatively, the maximum search time for the solver may be increased, or simply a faster computer must be used.

Nevertheless, a timeout or a fail cannot always be avoided. The most musically appropriate strategy what to do in this case cannot be foreseen by a system designer as it depends on the CSP at hand. Therefore, the user can freely define what to do in this event. In many cases, it may be suitable to output nothing for that time slice (a rest). Depending on the CSP, it may be appropriate to repeat a previous result or to play some pre-composed snippet, possibly varied (e.g. transposed). Also, it might be fitting to create an "emergency CSP" with a minimal set of constraints: the timeout for the main CSP would be chosen such that it would still leave some time of the "emergency CSP".

A completely different approach would use an incomplete search, for example, a local search algorithm as in the music constraint system OMClouds [21]. This approach has the advantage that there is always a current solution available when the search must be terminated after the search time elapsed. However, local search finds usually only approximated solutions, because all its constraints are effectively soft constraints. Having soft constraints only is insufficient for many music theory models. For example, an approximated consonance or an approximated all-interval series are of limited interest.

One can argue that our sub-CSPs have some similarity with the time steps in tcc and its variants. In both approaches, constraint-based computations occur at particular moments in time on fresh variables contained in a store. However, in our approach each sub-CSP constitutes a combinatorial problem, which is solved by search. The solver creates a tree of stores (encapsulated in computation spaces) – which allows for speculative computations per time step. By contrast, tcc creates only one store per time step.

For efficiency, the constraint solver Cassowary resolves similar problems incrementally (see above). Incremental solving is not suitable for our purposes, though. We usually don't want successive musical outputs to be as similar as possible – in contrast to Cassowary's graphics-related applications such as positioning the windows on a screen. Also, successive sub-CSPs are not necessarily related in music constraint programming, in particular when users may interactively change the set of constraints between sub-CSPs.

Acknowledgements

We thank Graham Percival for his comments on this text. This research was supported by the EPSRC project 'Learning the Structure of Music' (LeStruM), EPD063612-1.

7. REFERENCES

[1] A. Allombert, G. Assayag, M. Desainte-Catherine, and C. Rueda. Concurrent Constraints Models for Interactive Scores. In *Sound and Music Computing (SMC) 2006*, Marseille, 2006.

[2] T. Anders. Strasheela. `http://strasheela.sourceforge.net/` (accessed 19 May 2008).

[3] T. Anders. *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System*. PhD thesis, School of Music & Sonic Arts, Queen's University Belfast, 2007.

[4] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4), 2001.

[5] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), 1981.

[6] M. Chemillier and C. Truchet. Two Musical CSPs. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001.

[7] W. Hower and W. H. Graf. A bibliographical survey of constraint-based approaches to CAD, graphics, layout, visualization, and related topics. *Knowledge-Based Systems*, 9(7), 1996.

[8] M. Laurson and M. Kuuskankare. 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, 2001.

[9] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4), 2002.

[10] F. Pachet and O. Delerue. MidiSpace: a Temporal Constraint-Based Music Spatializer. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

[11] F. Pachet and P. Roy. Musical Harmonization with Constraints: A Survey. *Constraints Journal*, 6(1), 2001.

[12] C. Palamidessi and F. D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *Principles and Practice of Constraint Programming – CP 2001*. Springer, 2001.

[13] M. Puckette. Max at Seventeen. *Computer Music Journal*, 26(4), 2002.

[14] C. Rueda, G. Assayag, and S. Dubnov. A Concurrent Constraints Factor Oracle Model for Music Improvisation. In *Proceedings of Latin American Informatics Conference CLEI 2006*, Santiago de Chile, 2006.

[15] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6), 1996.

[16] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[17] W. Schottstaedt. Automatic Counterpoint. In M. V. Mathews and J. R. Pierce, editors, *Current Directions in Computer Music Research*. The MIT Press, 1989.

[18] C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, 1997. The MIT Press.

[19] C. Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[20] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6.329–6.346. ACM, 1964.

[21] C. Truchet, G. Assayag, and P. Codognet. Visual and Adaptive Constraint Programming in Music. In *Proceedings of International Computer Music Conference 2001*, Havana, Cuba, 2001.

[22] P. van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[23] M. Wright and A. Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, Hellas, 1997.

[24] A. Zils and F. Pachet. Musical Mosaicing. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, Limerick, Ireland, 2001.