HIGHER-ORDER CONSTRAINT APPLICATORS FOR MUSIC CONSTRAINT PROGRAMMING

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 studies how constraints are applied to the score in a musical constraint satisfaction problem (CSP). How can we control which variable sets in the score are affected by a given constraint? Our overall objective is to produce a highly generic music constraint system, where users can define a wide range of musical CSPs, including rhythmic, harmonic, melodic and contrapuntal problems. Existing systems provide constraint application mechanisms which are convenient for specific cases, but lack generality and cannot be extended by users. As a result, complex sets of variables are hard to constrain in these systems. For example, constraining notes from different voices in a polyphonic setting (e.g., with harmonic constraints) is inconvenient or even impossible in most systems.

We propose an approach which combines convenience with full user control: higher-order constraint applicators. A constraint is a first-class function, while a constraint applicator is a higher-order function which traverses the score in order to apply a given constraint to variable sets. This text presents constraint applicators suitable for a many musical CSPs, and reproduces important mechanisms of existing systems. Most importantly, users can define their own constraint applicators with this approach.

1. INTRODUCTION

Constraint programming has often been used for making computational models of music theories and composition. Many compositional tasks have been addressed by constraint programming successfully. Constraint-based harmonisation systems are surveyed in [14]. Other examples include purely rhythmical tasks [18], Fuxian counterpoint [19], Ligeti-like textures [10, 5], and instrument-specific writing [10].

A *constraint satisfaction problem* (CSP) consists of a set of *variables* and mathematical relations between these variables which are called *constraints*. Variables have a *domain*, that is a set of values they may take in a solution. In most systems, the domain is a finite set (e.g., a finite set of integers). A CSP usually presents a combinatorial problem. A *constraint solver* finds one or more solutions for the problem. A *solution* determines each variable to a domain value which is consistent with all its constraints.

We aim at producing a highly generic music constraint system where users can define a wide range of musical CSPs, including rhythmic, harmonic, melodic and contrapuntal problems. Several music constraint systems have been proposed which allow users to define their own musical CSP. However, these systems are often designed for specific problem classes (e.g., harmony) to the detriment of others. The bias of these systems is also reflected by their constraint application mechanisms, which are convenient but restrictive. We therefore propose an constraint application approach which is both convenient and generic. It is implemented in our system Strasheela [3].

2. EXISTING CONSTRAINT APPLICATORS AND THEIR LIMITATIONS

Many music constraint systems have been proposed. Two seminal systems are PWConstraints [9] and Situation [17]. Carla [6] is a pioneering system. Further examples include the aggregation of the music representation MusES with the constraint system BackTalk [13], OMRC (defined on top of PWConstraints) [18], Arno [2], and OMClouds [20].

Each system provides three components, which are essential for a music constraint system:

i) a music representation, where some aspects (e.g., note durations or pitches) can be variables (unknowns),

ii) a mechanism for defining constraints and for applying them to variables in the music representation,

iii) a constraint solver which finds a solution for all the variables in the music representation.

Nevertheless, these systems differ widely in their music representation, their constraint definition and application, and their constraint solvers. As a result, different systems are suitable for different CSPs. For example, the system Situation is well suited for creating Messiaen-like chord progressions, and the PWConstraints subsystem Score-PMC supports complex polyphonic CSPs.

The present paper studies one of these three components in more detail, namely how constraints are defined and applied to the variables. Basically, in many existing systems users define constraints as Boolean expressions. Such constraint definitions are convenient and fully generic: arbitrarily complex constraints can be defined as Boolean expressions.

However, the application of constraints to the variables in the score is problematic. Users want to constrain diverse musical aspects (e.g., note durations and pitches, pitch intervals, the metric positions of notes, the pitch classes of chords and their roots etc.), possibly even in a single constraint. Music constraint systems aim for making CSPs convenient to define for the user, but at the same time their constraint applications must somehow access all the music representation variables expressing these diverse aspects.

Moreover, a musical constraint is often applied to several score object sets at the same time. For example, a classical counterpoint rule restricts which intervals can occur between two melody notes: the corresponding constraint would be applied to all pairs of consecutive notes in all voices. The user of MusES plus BackTalk addresses this case by a loop whose running index i is used to access neighbouring note pairs in a melody as follows (cf. [16]).

$$myConstraint(nth(MyMelody, i), \\ nth(MyMelody, i+1))$$

For complex rules, however, it becomes tedious to explicitly access all the variables involved. Existing systems therefore often provide convenient mechanisms to apply a constraint to several score object sets at the same time. Yet these mechanisms are not generic: some object sets are supported and others are not.

The difficulties of constraint application are discussed below by outlining the application mechanisms of the two seminal systems PWConstraints and Situation. Both systems where originally developed for PatchWork [9]. They are meanwhile available in the PatchWork successor systems OpenMusic [4] or PWGL [11].

2.1. PWConstraints

PWConstraints [9, 17] proposes a pattern matching mechanism to apply constraints to the score. This mechanism plays such an important role for the system that subsystems were named accordingly (e.g., the basic subsystem is PMC, which stands for *pattern matching constraints*). A constraint consists of a pattern matching part and a body (a Boolean expression). The body is applied to any set of score objects which matches the corresponding pattern. For example, the pattern $[*, X, Y]$ matches every two neighbours of a sequence (the star matches zero or more objects).

Figure 1 shows this pattern in a melody constraint: the interval between any two consecutive pitches must not exceed a fifth (the interval is measured in semitones, a fifth corresponds to 7). [1] Pattern matching variables are implicitly bound to the matching values (e.g., the first two pitches, then the second two pitches etc.). The free variables X and Y in the body (Figure 1, second line) use this binding. Pattern matching is a convenient application mechanism for sequential score object sets (e.g., a sequence of melody notes).

$[*,\ X,\ Y]$	pattern matching expression
$\lvert X - Y \rvert \leq 7$	constraint body

Figure 1. Melodic PMC constraint: the interval between any two consecutive pitches must not exceed a fifth

However, polyphonic CSPs often include non-sequential score object sets (e.g., simultaneous notes which stem from different voices). PWConstraints' polyphonic subsystem Score-PMC therefore defines a polyphonic music representation, whose accessor functions can be called within a constraint definition. Score-PMC provides accessors to the following score contexts: the sequence of melodic notes, simultaneous notes, and the metric position of a note. An extended pattern matching language simplifies access to these contexts [12]. Instead of matching only individual notes of a single melody, new language keywords support the matching of simultaneous note sets and note sets at specific metric positions. Highly complex polyphonic CSPs can be expressed with Score-PMC.

Nevertheless, the designer of a composition system cannot foresee all potential needs of its users, and therefore composition systems tend to be highly user-extendable. By contrast, Score-PMC users cannot introduce new pattern matching language keywords to access further score contexts (nor can they extend the music representation). Moreover, pattern matching is axiomatically limited to what can be expressed by a pattern: PWConstraints' pattern matching language cannot express every possible combination of elements in a sequence. For example, a single PWConstraints pattern cannot express 'any pair of neighbouring variables in a sequence such that pairs do not overlap': for the sequence $[a, b, c, d]$, there is no pattern that only matches $X = a \wedge Y = b$ and $X = c \wedge Y = d$.

2.2. Situation

Situation [17] was originally conceived in collaboration between the composer Antoine Bonnet and the computer scientist Camilo Rueda as a constraint system for solving a range of harmonic CSPs. Its music representation still reflects the history of the system: music is represented as a sequence of composite score objects (e.g., chords, or rhythmic motifs).

The constraint application mechanism of Situation is tailored for this sequential music representation: constraints are applied to sets of objects which are identified by their numeric index in the sequence. For example, Situation makes it easy to constrain the first and the fifths chord in a chord sequence to contain specific pitches.

Any possible score object set can be expressed by an index-based constraint application mechanism. Nevertheless, such mechanism is only convenient for a sequence

[1] In PWConstraints, rules are defined in Lisp – this text uses a mathematical notation instead.

of score objects, but it is less suitable for constraining complex score topologies (e.g., hierarchically structured scores). For example, when constraining simultaneous notes the user would need to know the sets of numeric indices of simultaneous note sets.

3. INTRODUCING OUR APPROACH

As mentioned earlier, our objective is to produce a highly generic music constraint system, in which the user can define a wide range of musical CSPs, including rhythmic, harmonic, melodic and contrapuntal problems. For such a generic system, the existing constraint application mechanisms are limiting. We therefore propose an approach which makes constraint application mechanisms freely programmable.

Our approach is based on a well-understood formalism: a constraint is a first-class function [1], and the application of a constraint to the score is simply a function application. In contrast to existing systems, this formalism fully decouples the definition and the application of a constraint. That way, the user can freely apply each constraint to arbitrary sets of score objects by any control structure. Besides defining constraints, the user can also define convenient constraint application mechanisms. Such mechanisms are implemented as a higher-order functions which expects a constraint (i.e., another function) as argument.

In contrast to previous systems which provide a single and limiting constraint application mechanism, the Strasheela user can freely select from a range of predefined constraint application mechanisms, including the index-based application mechanism of Situation and the pattern-matching-based mechanism of PWConstraints. In addition, Strasheela provides several mechanisms not supported by previous systems (e.g., the application of a constraint to any score object which meets some user-defined condition). Most importantly, the user can define new constraint application mechanisms.

Music constraint programming always involves some form of a music representation, and existing systems differ widely in their representation. The proposed constraint application mechanism abstracts away from the actual music representation format. In principle it can be used for any representation format (e.g., a simple event list or a MIDI-like representation). Nevertheless, a hierarchic representation format is better suited for expressing more complex music CSPs. For example, one may use a variant of CHARM [7], or Smoke [15] for polyphonic CSPs. We use the Strasheela music representation, which provides a rich interface for accessing score information [3].

4. CONSTRAINTS ARE FIRST-CLASS FUNCTIONS

For the rest of this paper, a *constraint* is a function which returns a Boolean value. All arguments of a constraint are either variables or score data which contain variables (e.g., a note whose pitch is a variable). A variable has a domain, that is a set of values it may take in a solution. The Boolean value returned by a constraint is also a variable, and its value can be constrained. If and only if a constraint returns $true$, then do the constraint's arguments satisfy the restriction of this constraint.

4.1. Direct Constraint Application

A constraint can be applied directly to score objects. The following example defines and applies the constraint *constrainClimax*. This constraint expects a list of melody pitches Ps and a numeric index I and ensures that the I-th element of Ps forms the peak of the melody. *constrainClimax* is applied directly to the list $MelodyPitches$, and constrains its 7th element to the melodic peak.

$$\begin{array}{ll}\textbf{let} & constrainClimax(Ps, I) := \max Ps = nth(Ps, I) \\ \textbf{in} & constrainClimax(MelodyPitches, 7)\end{array}$$

The example returns the Boolean variable which is returned by the constraint. This text adopts the convention that a Boolean variable returned by a full example is implicitly constrained to be $true$.

4.2. Applying a Constraint to Every Element in a List

Often we want to apply a constraint multiple times to different score object sets. Any control structure can be used for this. For example, a constraint can be applied directly to all notes in a voice by iterating through all these notes in a loop. Instead, this text proposes the use of higher-order functions for this purpose, because higher-order functions can encapsulate any traversal of score objects, even if it is far more complex than iteration. A constraint is a first-class function; a constraint applicator is a higher-order function which expects a constraint (i.e., a function) as argument.

A programming technique very similar to iteration is mapping. The higher-order function map applies a function to every element in a list, and returns a list with the collected results. In Figure 2, map applies the unary constraint *restrictPitch* to every note of the alto voice. *restrictPitch* accesses the pitch of its note-argument, and constrains its vocal range to g3–e5 (i.e., the MIDI key-numbers 55–74).

$$\begin{array}{ll}\textbf{let} & restrictPitch(N) := getPitch(N) \in \{55, \ldots, 76\} \\ \textbf{in} & \bigwedge map(getNotes(AltoVoice), restrictPitch)\end{array}$$

Figure 2. Restrict the pitch domain of every note in the alto voice to g3–d5

The function map returns a list of Boolean variables and the operator $\bigwedge$ returns the conjunction of all these values. Again, the Boolean returned by the example is implicitly constrained to be true, thus all $restrictPitch$ applications in the example are constrained to return true.

4.3. Applying a Constraint to Neighbours in a List

The next example constrains multiple pairs of score objects. In a sequence of chords $MyChords$ every two consecutive chords are constrained to share common pitches. More specifically, the intersection of the pitch classes of these chords must not be empty. The constraint applicator is the function $map2Neighbours$, which expects a list and a binary function as arguments. It applies the function to every pair of neighbouring elements in the list. [2] Note that this applicator is also often used for melodic rules, for example, to constrain the interval between neighbouring notes.

$$map2Neighbours(MyChords, \\ f : f(C_1, C_2) := (C_1 \cap C_2) \neq \emptyset)$$

Figure 3. Constrains that neighbouring chords in $MyChords$ share common pitches

5. USER-DEFINED CONSTRAINT APPLICATORS

The examples above demonstrate how higher-order functions are used as constraint applicators. This section shows how such applicators are defined by the user. This ability marks an important distinction between Strasheela and existing systems.

The higher-order function map (applied in Figure 2) is widely known in functional programming, and its definition can be studied in many textbooks on functional programming languages, such as [1]. Figure 4 defines the higher-order function $map2Neighbours$ which was used in the example above (Figure 3). The definition is very brief and consists of only a call to the function *zip*, a relative of the function map.

$$map2Neighbours(Xs, fn) := \\ zip(butLast(Xs), tail(Xs), fn)$$

Figure 4. Constraint application functions can be defined by the user: the definition of the higher-order function $map2Neighbours$

The function *zip* expects two lists Xs and Ys and a binary function *fn*, and collects the results of all calls $fn(x_i, y_i)$, where x_i and y_i are the i-th element of Xs and Ys. For instance $zip([1,2,3],[4,3,2],max)$ returns $[4,3,3]$. The Common Lisp equivalent of *zip* is $mapcar$, which supports mapping over any number of lists. The function $map2Neighbours$ calls *zip* with three arguments, namely two sublists of its list argument Xs – one containing all but the last element and the other all but the first element of Xs – plus the binary function argument fn.

Whereas $map2Neighbours$ applies a binary constraint to every pair of two neighbouring list elements, the generalised function $mapNeighbours$ applies an n-ary function to every sublist of n neighbours in a list. The actual definition of $mapNeighbours$ is not shown for brevity. $mapNeighbours$ is useful, e.g., for applying melodic constraints which affect more than two neighbouring notes (e.g., counterpoint rules like: "after a large skip follows a step in the opposite direction").

6. MODELLING EXISTING CONSTRAINT APPLICATION MECHANISMS

This text argues that using higher-order constraint applicators is more generic than the constraint application mechanisms of existing systems. To substantiate this claim, this section reproduces the constraint application mechanisms of Situation and PWConstraints as higher-order functions. Please note that these systems do not support the application mechanisms of each other.

6.1. Index-Based Constraint Application

Situation offers index-based constraint application mechanisms. The system introduces various constraint-specific mini-languages for controlling the constraint application to score objects. The present section describes two examples, which are then reproduced as higher-order functions.

One Situation mechanism applies a constraint to single objects at specific positions in a sequence. The following example (Figure 5) constrains the 1st, 6th, 11th, and 21st note of $MyVoice$ to start a new phrase. The example applies the constraint $beginPhrase$ to these notes, and $beginPhrase$ constrains a note to follow a rest and additionally constrains the note's duration to at least a half note. [3] The constraint applicator $mapIndex$ reproduces the Situation mechanism which applies a constraint to every list element whose position is specified. This function expects three arguments: a list (in Figure 5 the notes of $MyVoice$), a specification of indices (1, 6, 11, and 21), and a unary constraint ($beginPhrase$). $mapIndex$ is defined in [3].

The next example is particularly typical for Situation, which has been developed originally for solving harmonic CSPs. Figure 6 constrains chords at specific positions in a chord sequence to contain pitches specified for these positions. The constraint $requirePitches$ expects a chord C and a list of pitches Ps: all elements of Ps must be in $getPitches(C)$. The constraint applicator ***mapIndexArgs*** is similar to $mapIndex$, but the index declaration is now a mini-language. The specification $1\#[61,65]$ denotes that the binary constraint $requirePitches$ is called

[2] This paper uses the *where-notation* [8] as syntactic sugar for an anonymous function. For example, the function f serves the purpose of an anonymous function in $g(f \textbf{ where } f(x) := x^2)$ or shorter $g(f : f(x) := x^2)$.

[3] The definition of $followsRest$ depends on the music representation. Strasheela provides the note parameter $offsetTime$ for expressing a preceding rest.

$$
\begin{array}{ll}
\textbf{let} & \mathit{beginPhrase}(N) := \\
& \quad \mathit{getDuration}(N) \geq \mathit{halfNote} \\
& \quad \wedge\, \mathit{followsRest}(N) \\
& \mathit{indices} := [1, 6, 11, 21] \\
\textbf{in} & \bigwedge \mathit{mapIndex}(\mathit{getNotes}(\mathit{MyVoice}), \\
& \qquad \mathit{indices}, \mathit{beginPhrase})
\end{array}
$$

Figure 5. Constrain the notes at position 1, 6, 11, and 21 in a given voice to start a new phrase: these notes have a relatively long duration and follow a rest

$$
\begin{array}{ll}
\textbf{let} & \mathit{requirePitches}(C, Ps) := \\
& \quad \bigwedge \mathit{map}(Ps, f : f(P) := P \in \mathit{getPitches}(C)) \\
& \mathit{decl} := [1\#[61, 65], 3\#5\#[60, 66], 6\#[61, 65]] \\
\textbf{in} & \bigwedge \mathit{mapIndexArgs}(\mathit{MyChordSeq}, \\
& \qquad \mathit{decl}, \mathit{requirePitches})
\end{array}
$$

Figure 6. Constrains chords at specific positions to contain specific pitches

with the 1st chord and the list of pitches $[61, 65]$. The notation $3\#5\#[60, 66]$ means all chords from position 3 to 5 are constrained with the pitch list $[60, 66]$.

Our reproductions of Situation's application mechanisms are even more general then the original. In Situation, these mechanisms are hard-wired to a specific constraint. As higher-order functions, however, they can be used for applying any constraint. Also, the original mechanisms are restricted to the sequential score topology of Situation. Instead, the mechanisms of this section can be used on any score object set which can be represented as a sequence. For example, Figure 5 applies a constraint to the notes of a specific voice – which might be only one of several voices in a polyphonic score. The score object set 'notes of voice $MyVoice$' has no equivalent in Situation.

6.2. Constraint Application with a Pattern Matching Language

This section reproduces the constraint application mechanism of PWConstraints. In PWConstraints, a constraint is applied to all object sets which match the pattern matching expression of the constraint header (see above).

The present section defines a pattern matching language which is similar to PWConstraints', and which defines three symbols. There are two place-holder symbols: ? (question mark) matches exactly one sequence element, and $*$ (star) matches zero or more elements. Instead of PWConstraints' pattern-matching variables, this language provides the symbol x: every occurrence of a pattern-matching variable in PWConstraints is substituted by the unvarying symbol x here. The following example shows an expression which matches any but the first pair of two successive elements.

$$
\begin{array}{ll}
\textbf{let} & \mathit{restrictIntervals}([P_1, P_2]) := |P_1 - P_2| \leq 7 \\
\textbf{in} & \bigwedge \mathit{mapPM}(\mathit{mapItems}(\mathit{MyVoice}, \mathit{getPitch}), \\
& \qquad [*, x, x], \mathit{restrictIntervals})
\end{array}
$$

Figure 7. Constrains the interval between two consecutive note pitches in $MyVoice$ not to exceed a fifth

$$[?, *,\ x,\ x]$$

PWConstraints' constraint application mechanism is reproduced by the higher-order function $mapPM$. It expects three arguments: a list Xs (e.g., score objects), a pattern matching expression $pattern$ (using the syntax above), and a unary constraint f which expects a list. The function $mapPM$ applies f to every sublist of Xs which matches $pattern$. The example below results in the following two function calls: $[f([a, c]), f([a, d])]$.

$$
\begin{array}{ll}
\textbf{let} & \mathit{pattern} := [x, ?, *,\ x,] \\
\textbf{in} & \mathit{mapPM}([a, b, c, d], \mathit{pattern}, f)
\end{array}
$$

Figure 7 uses $mapPM$ to apply a melodic constraint to all pairs of consecutive note pitches in $MyVoice$. The function $mapPM$ is defined in [3].

Note that the approach proposed here is more general than PWConstraints' constraint application mechanism. The function $mapPM$ can be used on any data sequence. For example, $mapPM$ can apply constraints to any score object sequence extracted from a highly nested music representation. This is similar to the effect of the new pattern matching keywords introduced by [12] discussed above, but here the user is not limited to a set of predefined keywords.

7. FURTHER EXAMPLES

7.1. Constraint Application to Selected Objects in a Score Hierarchy

The constraint applicators introduced so far apply a constraint to specific elements (or element sets) in a sequence. These mechanisms were implemented by higher-order functions which traverse a sequence for applying a constraint. Yet higher-order functions can define control structures which traverse arbitrary data structures. The present section proposes a technique which applies a constraint to all score objects in a hierarchic music representation which meet a user-defined condition.

Strasheela supports the hierarchic nesting of score objects in order to express, for example, which objects form a voice, a motif or other object sets. Score objects which hold other objects are called containers. Strasheela containers understand a method map which generalises the map-function discussed above. This method traverses a score hierarchy instead of a list. The method also supports optional arguments. For example, the argument $test$

expects a Boolean function: only objects for which the test returns *true* are processed; other objects are skipped.

Figure 8 uses this method *map* for constraining any instance of a specific motif a to have a single melodic peak. These motifs can be nested arbitrarily deep in the score. *map* applies the constraint *hasUniquePeak* to all objects which are a container marked with the tag $motif_a$. This condition is defined by the test function f. The function *hasThisInfo* returns a Boolean whether a given object is tagged with a specific symbol. The constraint *hasUniquePeak* (Figure 9) constrains that the melodic peak (i.e., the maximum pitch) occurs exactly once in all the notes contained in its argument motif M. The constraint *once* forces that $MaxP$ occurs only once in Ps.

$$\bigwedge map(MyScore, hasUniquePeak,\ test{:}\ f : f(X) := isContainer(X) \wedge hasThisInfo(X, motif_a))$$

Figure 8. The constraint *hasUniquePeak* is applied to every container marked as $motif_a$

$$\begin{array}{l} hasUniquePeak(M) := \\ \quad \textbf{let}\ \ Ps := map(M, getPitch, test{:}\ isNote) \\ \qquad\ \ MaxP = max(Ps) \\ \quad \textbf{in}\ \ \ once(MaxP, Ps) \end{array}$$

Figure 9. In the list of all note pitches in the motif M, the maximum pitch occurs exactly once

7.2. Nested Constraint Application

This section shows how more complex compositional rules are defined by nesting constraint applicators. Note that the constraint application mechanisms of Situation and PWConstraints do not support nesting. For example, in PWConstraints a pattern-matching expression can only occur in the header of a constraint but not in the constraint body, nor can a constraint call other constraints.

The following example applies a harmonic constraint (Figure 10). All pairs of simultaneous notes in a polyphonic score are constrained to be consonant. The rhythmical structure of the music can be arbitrarily complex.

$$\bigwedge mapSimNotePairs(collect(MyScore, test{:}\ isNote), isConsonant)$$

Figure 10. All pairs of simultaneous notes are constrained to be consonant

The constraint applicator *mapSimNotePairs* traverses all notes in the score and applies the constraint *isConsonant* to all pairs of simultaneous notes. The method *collect* is a relative of the method *map* discussed above: *collect* traverses the score hierarchy below a given container and selects all score objects for which a given Boolean function returns *true*. In this example, *collect* returns all notes in $MyScore$.

The constraint applicator *mapSimNotePairs* is realised with two nested mappings (Figure 11). The outer mapping traverses $Notes$, a given list of note objects. For every note N_1, the function *getHigherSimNotes* returns all notes simultaneous with N_1 but which are situated in a higher voice (discussed below). The inner mapping applies the given function *myConstraint* to every N_1 in $Notes$ and any of its simultaneous notes.

$$\begin{array}{l} mapSimNotePairs(Notes, myConstraint) := \\ \quad map(Notes, \\ \qquad f : f(N_1) := \\ \qquad\quad \bigwedge map(getHigherSimNotes(N_1), \\ \qquad\qquad g : g(N_2) := \\ \qquad\qquad\quad myConstraint(N_1, N_2))) \end{array}$$

Figure 11. The applicator *mapSimNotePairs* applies a given constraint to all pairs of simultaneous notes

Figure 12 defines the function *getHigherSimNotes*. This function exploits the fact that score objects in Strasheela's hierarchic music representation are bidirectionally linked: every container has access to its contained score objects (e.g., notes or other containers) and vice versa. That way, any note object N_1 can access the top-level container of the score *Top*. The container *Top* then collects all score objects which meet a given condition using the method *collect* just introduced. The method collects all objects in *Top* for which the Boolean function f returns *true*, namely all notes which are situated in a higher voice than N_1 (the highest voice has the lowest voice position) and which additionally are simultaneous with N_1. N_1 itself is excluded from the result.

$$\begin{array}{l} getHigherSimNotes(N_1) := \\ \quad \textbf{let}\ \ Top := getTopLevelContainer(N_1) \\ \qquad f(N_2) := \ \ N_1 \neq N_2 \\ \qquad\qquad\qquad \wedge\ isNote(N_2) \\ \qquad\qquad\qquad \wedge\ getVoicePos(N_1) > getVoicePos(N_2) \\ \qquad\qquad\qquad \wedge\ isSimultaneous(N_1, N_2) \\ \quad \textbf{in}\ \ collect(Top, test{:}\ f) \end{array}$$

Figure 12. *getHigherSimNotes* returns all notes which are simultaneous with a given note but which are from a higher voice (with lower voice position)

The constraints *isSimultaneous* and *isConsonant* are defined below. *isSimultaneous* returns *true* if two given

score objects overlap in score time. *isConsonant* constrains the interval between two notes N_1 and N_2 to either a prime, a minor third, a major third, and so on up to a fifth plus an octave.

$$
\begin{aligned}
&isSimultaneous(X, Y) := \\
&\quad getStartTime(X) < getEndTime(Y) \\
&\wedge\ getStartTime(Y) < getEndTime(X)
\end{aligned}
$$

$$
\begin{aligned}
&isConsonant(N_1, N_2) := \\
&\quad \textbf{let}\ \ Interval \in \{0, 3, 4, 7, 8, 9, 12, 15, 16, 19\} \\
&\quad \textbf{in}\ \ \ Interval = |getPitch(N_1) - getPitch(N_2)|
\end{aligned}
$$

This example can be refined easily to follow conventional counterpoint rules where the interval between a bass note and a higher note must not form a fourth (so $^6_4$-chord inversions are avoided), but a fourth can occur between non-bass notes. Instead of traversing all notes at once with $mapSimNotePairs$, the bass notes and the non-bass notes would be addressed individually. The present constraint $isConsonant$ would be applied to pairs with a bass note, and a relaxed version (which additionally allows the fourth) to non-bass note pairs.

This example accessed and constrained simultaneous notes. However, the presented approach is much more general. Any sets of score objects can be accessed and constrained in this manner – as long as the music representation provides enough information to isolate these score object sets.

Note that complex constraint applications as presented in this example are impossible in most existing systems. Score-PMC allows for similar constraint applications in principle, but only by using its music representation API directly – its extended pattern matching language does not cover this case. As a higher-order function, on the other hand, users can define constraint applicators like $mapSimNotePairs$ themselves. Using the applicator is convenient: users only specify a list of score objects and the constraint – the applicators hides all the details.

8. RELATION TO THE STRASHEELA IMPLEMENTATION

Strasheela provides higher-order constraint applicators as proposed in this text, but the Strasheela implementation differs from the approach described here. This text is based on the notion of functions, because first-class functions are well known in the computer music community, and we can use the common mathematical notation for them.

Strasheela, however, is based on the concurrent constraint programming model of its underlying programming language Oz [21]. Oz provides first-class procedures for abstraction (functions are a special case), and procedures can run concurrently. For example, constraints are concurrent agents in Oz (constraint propagation happens concurrently).

So, most conjunction constraints notated explicitly in this text (e.g., the conjunction of all Booleans returned by a map) are not required in Strasheela. The set of all applied constraints (i.e., all concurrent propagators) are implicitly conjunct. Still, constraints like conjunction or implication are supported as well, as constraints can be reified (the validity of a constraint can be constrained).

Several examples in this text perform complex operations when accessing score objects for constraint application (e.g., a traversal of all score objects in the score). Nevertheless, such constraint applications are not computationally expensive in Strasheela, because this access is performed only once, and then the applied constraints run concurrently.

The concurrent programming model of Oz blocks if a thread misses required information. For example, the function $getHigherSimNotes$ (Figure 12) blocks until it is know whether notes are simultaneous: the constraint application is delayed until enough information is available. Consequently, the constraint applicators can exploit information even if this information is not specified in the CSP, but only found during the search process. For example, Strasheela can constrain simultaneous notes to be consonant even if the rhythmical structure of the music is undetermined in the CSP definition. For an efficient search process, however, the user must take care that the constraint is not applied too late: in this case, the temporal parameters of a note should be determined before its pitch so that the constraint $isConsonant$ is applied and can perform constraint propagation before the note's pitch is searched for.

9. DISCUSSION

Strasheela provides direct constraint application as well as convenient constraint application mechanisms. The fact that a Strasheela constraint is a first-class function allows the user to program complex constraint application mechanisms as higher-order functions.

Experience with Strasheela shows that a CSP which makes use of suitable higher-order functions is often more concise than an equivalent CSP which applies constraints directly (e.g., by explicit nested loops). Higher-order functions can abstract away complex control structures which implement mathematical concepts (e.g., the Cartesian product), or knowledge about the music representation (e.g., implicit traversing of a score hierarchy).

Arbitrary control structures can be defined in terms of higher-order functions, including the constraint application mechanisms of previous systems. As a demonstration, index-based constraint application mechanisms of Situation and the pattern-matching based mechanism of PWConstraints have been reproduced in Strasheela.

Note that first-class functions can also express constraint application mechanisms which cannot be adequately reproduced by the mechanisms of Situation and PWConstraints. The mechanisms of both systems are highly suited to applying a constraint to elements in a sequence, be-

cause these mechanisms rely on positional relations, such as neighbouring elements. First-class functions, on the other hand, can process arbitrary data structures, such as trees or graphs, besides sequences. For example, a constraint can be applied to all score objects in a hierarchic score representation for which some test function returns true. Hence, a constraint application mechanism based on the notion of higher-order functions is more generic than the mechanisms of Situation and PWConstraints.

Because higher-order constraint applicators allow for a concise and expressive CSP definition, other music constraint systems may be interested in adopting these mechanisms. Higher-order constraint applicators can be easily reproduced in the combination of MusES and BackTalk. This system is implemented in Smalltalk which provides code blocks, and these are essentially first-class functions. Realising higher-order constraint applicators in systems which do not support direct access to their music representation and its variables (i.e., Situation, PWConstraints and OMClouds) would require internal changes to these systems.

Acknowledgements

We thank Graham Percival for his comments on this text.

10. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] T. Anders. Arno: Constraints Programming in Common Music. In *Proceedings of the 2000 International Computer Music Conference*, Berlin, Germany, 2000.

[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. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue. Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal*, 23(3), 1999.

[5] 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.

[6] F. Courtot. A Constraint Based Logic Program for Generating Polyphonies. In *Proceedings of the International Computer Music Conference*, Glasgow, 1990.

[7] M. Harris, A. Smaill, and G. Wiggins. Representing Music Symbolically. In *IX Colloquio di Informatica Musicale*, Genoa, Italy, 1991.

[8] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3), 1966.

[9] M. Laurson. *PATCHWORK: A Visual Programming Language and some Musical Applications*. PhD thesis, Sibelius Academy, Helsinki, 1996.

[10] 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.

[11] M. Laurson and M. Kuuskankare. PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In *Proceedings of International Computer Music Conference*, Göteborg, Sweden, 2002.

[12] M. Laurson and M. Kuuskankare. Extensible Constraint Syntax Through Score Accessors. In *Journées d'Informatique Musicale*, Paris, 2005.

[13] F. Pachet and P. Roy. Mixing Constraints and Objects: a Case Study in Automatic Harmonization. In I. Graham, B. Magnusson, and J.-M. Nerson, editors, *Proceedings of TOOLS-Europe'95, Versailles, France*. Prentice-Hall, 1995.

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

[15] S. T. Pope. The Smoke Music Representation, Description Language, and Interchange Format. In *Proceedings of the International Computer Music Conference*, San Jose, 1992.

[16] P. Roy and F. Pachet. Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming*, 10(4), 1997.

[17] C. Rueda, M. Lindberg, M. Laurson, G. Block, and G. Assayag. Integrating Constraint Programming in Visual Musical Composition Languages. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

[18] O. Sandred. Searching for a Rhythmical Language. In *PRISMA 01*. EuresisEdizioni, Milano, 2003.

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

[20] C. Truchet, G. Assayag, and P. Codognet. OMClouds, a heuristic solver for musical constraints. In *MIC2003: The Fifth Metaheuristics International Conference*, Kyoto, Japan, 2003.

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