Strasheela: Design and Usage of a Music Composition Environment Based on the Oz Programming Model

Torsten Anders, Christina Anagnostopoulou, and Michael Alcorn

Sonic Arts Research Centre, Queen's University Belfast, Northern Ireland
{t.anders, c.anagnostopoulou, m.alcorn}@qub.ac.uk

Abstract. Strasheela provides a means for the composer to create a symbolic score by formally describing it in a rule-based way. The environment defines a rich music representation for complex polyphonic scores. Strasheela enables the user to define expressive compositional rules and then to apply them to the score. Compositional rules can restrict many aspects of the music – including the rhythmic structure, the melodic structure and the harmonic structure – by constraining the parameters (e.g. duration or pitch) of musical events according to some numerical or logical relation. Strasheela combines this expressivity with efficient search strategies.

Strasheela is implemented in the Oz programming language. The Strasheela user writes an Oz program which applies the Strasheela music representation. The program searches for one or more solution scores which fulfil all compositional rules applied to the score.

1 Introduction

In computer aided composition (CAC), a composer creates music by communicating her or his musical intentions to her 'assistant', the computer. CAC addresses music mainly on the score level and in that way CAC differs from other areas of computer music such as sound synthesis or spatialisation. By using a CAC environment a composer formalises musical ideas or compositional problems and implements them in a computer program which outputs music in a symbolic representation. Diverse strategies exist to generate or transform music; examples include mathematical models (e.g. stochastics), models based on transforming existing data (such as spectral analysis data), or models implementing already existing compositional strategies (e.g. serial composition) [1].

To advance in the compositional process, the composer must not worry too much about low level programming detail. It is therefore highly desirable for the composer to express her intentions on a high level of abstraction. Consequently, CAC environments rely heavily on the expressive power of the underlying programming language and its programming concepts.

Different CAC environments are based on different programming concepts or paradigms. Often, an environment supports a specific paradigm particularly well and encourages the user to employ this paradigm. Other CAC environments support a combination of programming paradigms and the user may separately choose the adequate paradigm for each given sub-problem.

Most CAC strategies and programming paradigms are clearly different compared to the way in which musicians describe a musical style. Using a CAC environment a composer may control aspects of the pitch contour of some voice by deciding for a specific random distribution to generate the pitches. She may further shape the contour by multiplying the resulting pitch sequence with an envelope. Instead of using such a deterministic strategy – in which one process modifies the result of the previous process – musicians tend to describe music by a set of modular rules. A rule is not an algorithm to create a certain result. A rule often states merely a restriction on single notes and their parameters (e.g. duration or pitch) or mutual dependencies between the parameters of multiple notes. Such restrictions do not necessarily result in a single solution score. Instead, the restrictions reduce the domain of all possible scores.

The constraint programming paradigm presents a natural CAC approach in which the composer defines such modular rules restricting a score. In fact, during the last decade constraint programming has become an important strategy for CAC and several environments supporting constraint programming have been developed [2, 3, 4, 5, 6].

Virtually all existing constraint based CAC environments extend already established general CAC environments by constraint programming means. Perhaps surprisingly, most current environments come with their own specifically developed constraint solver. This article proposes a different approach by extending a state-of-the-art constraint programming language into a CAC environment. The programming language Oz [7] offers highly expressive constraint programming means in a multi-paradigm programming context which makes the language very interesting for CAC. The present article proposes Strasheela,[1] a CAC environment implemented in Oz.

The implementation of Strasheela takes advantage of Oz' multi-paradigm programming support. Besides constraint programming, Strasheela applies object-oriented programming, and higher-order functional programming. Strasheela's main data structure, the score representation, is defined by a class hierarchy. Many score object methods are higher-order procedures and expect procedures or method labels as argument. Compositional rules are expressed by constraints on score objects.

---

[1] Strasheela is also the name of a scarecrow in the children's novel *The Wizard of the Emerald City* by Alexandr Volkov (in which the Russian author retells *The Wonderful Wizard of Oz* by L. Frank Baum). Although Strasheela's brain consists only in bran, pins and needles, he is a brilliant thinker who loves to multiply four figure numbers at night. Little is yet known about his interest in music, but Strasheela is reported to sometimes dance and sing with joy.

Plan of the Paper. The following section presents an overview of Strasheela from a user's point of view. The Strasheela score representation is discussed in Sec. 3. Strasheela suggests expressive strategies to define compositional rules and to apply them to the score (Sec. 4). Strasheela predefines distribution strategies – in effect search orders – which are optimised for scores (Sec. 5). Many aspects of Strasheela are explained throughout the text by a single canon example which is finished in Sec. 6. Section 7 presents related work. The article concludes with a discussion of Strasheela's limitations (Sec. 8).

2   Strasheela Overview

Strasheela offers a means to create a symbolic score by formally describing it in a rule-based way. The resulting score is later performed by human musicians or a sound synthesis language to create the actual sound. The main objective of Strasheela is the creation of original music and not to replicate traditional musical styles. Having said that, a conventional example based on well-known textbook rules is more easy to communicate in a paper focusing on software design. All compositional rules discussed here are hence inspired by traditional counterpoint [8].

As an example, we assume that a composer wishes to use Strasheela to create a canon, a musical form in which several voices imitate each other in a rather literal way. The Strasheela user first instantiates a score data object and in doing so she predetermines certain aspects of the score. For the canon, the composer predefines the number of voices in the score and the number of notes in each voice. However, the composer leaves other aspects of the score undetermined. She may leave undetermined all durations and pitches of the notes, because she wants these parameters to satisfy a set of compositional rules she has in mind.

Possible rules include restrictions on the pitch succession in each voice (melodic rules), rules restricting the simultaneous pitch combinations (harmonic rules), rules on the sequence of durations in a voice (rhythmic rules), and a rule restricting the different voices to be similar such that they form a canon.

Each rule imposes some constraints on some score objects. For example, a melodic rule may restrict the pitch interval between two successive notes. However, a melodic rule will usually not only affect a single note pair but, for instance, all successive note pairs in all voices. Rules are therefore defined in a modular way: the rule definition is abstracted from its application to multiple score objects.

By instantiating the score data object, defining the compositional rules, and by applying the rules to the score, the composer states the constraint problem. A solution of the problem is a score which fulfils all the rules applied to it.

In Oz, a constraint problem is implemented by a search script, a procedure with the solution as its only argument [9]. The constraint solver of Oz finds one or more solutions for the script.

Strasheela outputs the solution score into multiple formats including the score format of several sound synthesis languages and common music notation.

The Strasheela user interface is the Oz programming language: a Strasheela user writes an Oz program which applies Strasheela's contributions to Oz – most of all its score representation.

3   The Score Representation

A general and powerful music representation is vital for the expressivity of Strasheela, because both the solution score and the problem definition are expressed using this representation. Much research has been done in the domain of music representation [10, 11, 12, 13, 14, 15]. The score representation of Strasheela combines ideas presented in the literature and in existing implementations of CAC environments.

3.1   Class Hierarchy

A musical score contains many different object types. Examples in conventional music notation include notes marking pitch and timing information, articulation signs, and staffs of five lines to organise notes in voices. Different musical styles may use different type sets. During the compositional process the composer may even introduce further types (e.g. roman numbers to sketch a harmonic progression).

The Strasheela score representation attempts to generalise this broad width of possible score information. Instead of implementing an enormous set of different types in an unrelated way, the representation defines the score data types as classes in a class hierarchy in the object-oriented programming sense. Figure 1 presents an example excerpt of the class hierarchy. Depicted are the relations between the classes used to represent timing information. Many of these classes are explained in subsequent sections. The user can extend the class hierarchy by her own classes if so desired.

Object-oriented programming in Oz is often stateful. Nevertheless, the Strasheela score representation is stateless.

3.2   Hierarchic Score Structure

Most existing score representations support the notion of score *events*. The instances of the event class produce sound when the score is played.

Many event attributes (such as start time or pitch) are specified by *parameters*. Strasheela defines parameters in their own class to allow the addition of information to the actual parameter value. For instance, parameters allow the composer to specify their unit of measurement (such as key-number or cent-value for pitch) which subsequently affects the score when it is transformed into an output format. Parameter objects are also important for the definition of specific search strategies (Sec. 5). Parameter values are the only predefined score data which the composer can constrain.

The class *note* is an event subclass. Besides other event parameters (e.g. start time) a note defines the additional parameter pitch. The class *element* is
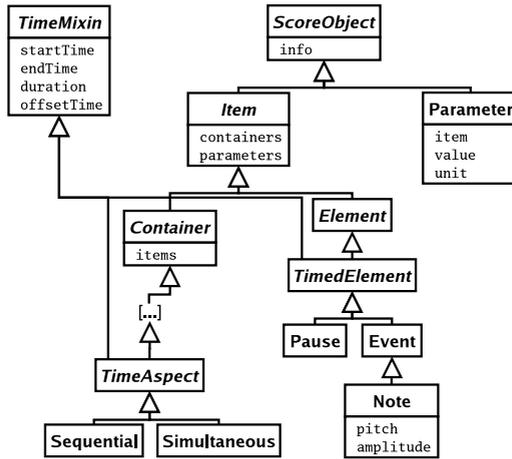
Fig. 1. Score class hierarchy. The excerpt shows timing related classes. The figure omits some classes for brevity, making some class names appear arbitrary (such as `TimeAspect` instead of `TimedContainer`)

a superclass of event. Instances of element subclasses (besides event) are silent when the score is played. Examples include the predefined class *pause* or a class representing an initialisation statement for some sound synthesis language.

Musicians rarely talk about single score events when talking about music. They talk about event groups such as motives, voices, rhythmic patterns, or chords. To express such concepts, Strasheela defines the class *container*. The superclass of both container and element is called an *item* in Strasheela. A container contains other items and so can represent groups of score objects. Data can be recursively nested to form a tree (e.g. to express a note in a motive in a melody, or a note in a chord in a staff).

Strasheela supports different hierarchies of different container types to express, for example, timing structure, grouping, harmonic information, or the bar structure. Multiple hierarchies can be combined in a graph in which different hierarchies share the same elements as leaves of their trees. As most of these container types depend closely on the music the user wants to represent, Strasheela predefines only abstract classes from which the user may derive her own classes according needs. Nevertheless, containers expressing a timing hierarchy are already predefined.

3.3   Hierarchic Timing Structure

Some score items have timing related parameters. For these objects, Strasheela explicitly represents the *start time*, *offset time*, *duration*, and *end time*. For all timed items, Strasheela implicitly constrains start time, duration, and end time (1). The offset time is an alternative means to express a pause in front of an item.

$$end_{\text{item}} - start_{\text{item}} = duration_{\text{item}} \tag{1}$$

Strasheela defines container classes whose instances constrain the timing of their contained items (Fig. 2). The items contained in a *simultaneous* object run in parallel with each other. The offset time of a contained item denotes how much the start time of the item is delayed. Equations (2) and (3) show the implicit constraints between a simultaneous object and all its contained items, n denotes the number of items in the container.
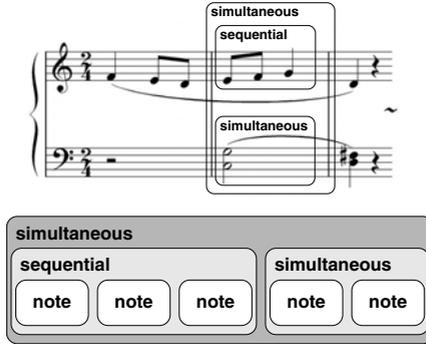

Fig. 2. The timing structure forms a tree with events as leaves, parameters are omitted (Béla Bartók. Mikrokosmos, No. 87)

$$\forall\, i \in \{1, \ldots, n\} : start_{\text{simItem}_i} = start_{\text{sim}} + offset_{\text{simItem}_i} \tag{2}$$

$$end_{\text{sim}} = \max(end_{\text{simItem}_1}, \ldots, end_{\text{simItem}_n}) \tag{3}$$

The items contained in a *sequential* object follow each other sequentially in time. The offset times of contained items specify pauses between the items (Equations (4) to (6)). Only the constraints (1) to (6) are implicitly applied to every score; further constraints are applied by the user.

$$start_{\text{seqItem}_1} = start_{\text{seq}} + offset_{\text{seqItem}_1} \tag{4}$$

$$\forall\, i \in \{1, \ldots, n-1\} : start_{\text{seqItem}_{i+1}} = end_{\text{seqItem}_i} + offset_{\text{seqItem}_{i+1}} \tag{5}$$

$$end_{\text{seq}} = end_{\text{seqItem}_n} \tag{6}$$

3.4   Application Programming Interface and Patterns

The application programming interface (API) for the score classes includes convenient constructors for complex scores, expressive score accessors as well as score transformers. For instance, the standard score constructor expects a shorthand representation of a score which consists of all score object initialisation methods nested according to the score hierarchy. Examples for typical accessors include a method which returns the item preceding some item in a container or

a method which returns all items in the whole score which are simultaneous to some item. Many accessors and transformers are higher-order procedures. Such accessors include, for instance, a method which maps a user specified function to all objects in the score graph which fulfil some test function. With the help of this method, the user can, for example, collect the pitches of every second note in some voice to constrain this pitch list to follow some user defined pattern.

Strasheela predefines many pattern constraints which either constrain the order of list elements by unification, impose numeric constraints on list elements, or combine multiple sublists into an other list. For example, a simple order pattern repeats the first n list elements throughout the list in a circular manner; a more complex example unifies list elements according to some Lindenmayer system defined by the user. Numeric patterns constrain, for example, each list predecessor to be smaller then its successor, the maximum number in a list to occurs only once, or n list elements to be pairwise distinct.

4   Compositional Rules and Their Application

Oz predefines a broad width of constraints, for instance, for finite domain (FD) integers [9] and for finite sets of integers [16]. The Strasheela user applies constraints to score parameter values – which are usually FD integers but may be other constrainable data as well – to express restrictions on these parameters. For instance, a composer may express a melodic restriction which constraints the distance between the pitches of two consecutive notes to not exceed the interval of a fifth (7). The interval is measured in semitones, 7 denotes a fifth.

$$7 \geq |pitch_1 - pitch_2| \tag{7}$$

Yet, a *compositional rule* is usually more general as it holds more than only once. The Strasheela user therefore often encapsulates the constraints expressing a compositional rule into an Oz procedure. The user freely controls the *rule scope* by defining a control structure which accesses sets of score objects and applies the rule to them. Often – but not necessarily – the rule scope has a relation to the hierarchic nesting of the score. For example, a rule restricting a melodic interval may be applied to any consecutive note pair in any sequential container of a score.

Each application of this rule constraints a set of score objects which are inter-related in a uniform way: the pitches of a consecutive note pair in a sequential. Another rule may constrain sets of score objects which are inter-related in another uniform way, for example, the duration of some note and the durations of all its simultaneous notes. A *context* is the way how a set of objects is inter-related. Strasheela's score API predefines various context accessor methods which return, for example, all items in the score which are simultaneous to some given item. The user can also define her own context accessors. Using these accessors, the context for a rule as well as the control structure for the rule scope is usually defined in a convenient way.

Figure 3 illustrates the terms rule, context and scope graphically and shows how each rule imposes one ore more constraints between several score object sets. The example rule `RestrictMelodicInterval` (Fig. 3, a – Fig. 4 shows the Oz code) implements (7) as a procedure with the argument $note_1$. The preceding $note_2$ is accessed within the procedure. The rule is applied to all notes in two different voices which have a predecessor (Fig. 3, b).
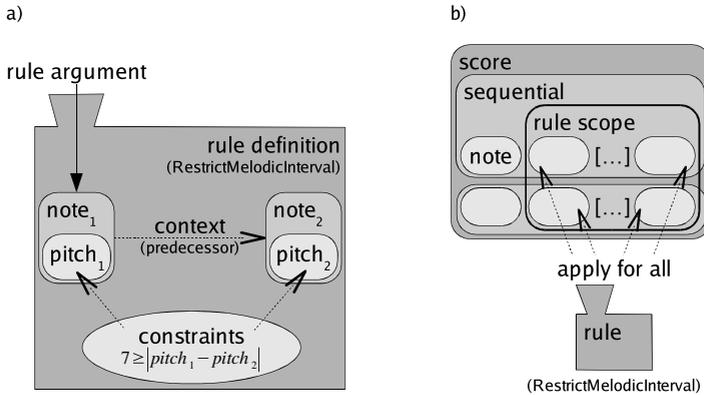

Fig. 3. Definition and application of a compositional rule. (a) A rule is a procedure which imposes constraints between the procedure arguments and often their contexts as well. (b) The rule scope is a set of score object sets to which the user applies the rule

```
proc {RestrictMelodicInterval Note1}
    Note2 = {Note1 getTimeAspectPredecessor($)}
in
    7 >=: {FD.distance {Note1 getPitch($)} {Note2 getPitch($)}}
end
```

Fig. 4. A melodic rule, defined as procedure

In Oz, a procedure is a first-class value which can be used as an argument to other procedures. Figure 5 shows how the scope of the rule `RestrictMelodic-Interval` (i.e. all notes which have a predecessor) is controlled. The method `forAll` applies the rule recursively to all score objects contained in `MyScore` for which the specified test function returns `true` – regardless of nesting depth.

There are situations in which a particular context of a score item is undetermined before search. For instance, simultaneous items are undetermined for most score items in case timing parameters (e.g. note durations) are found only during the search process. In such cases, standard accessors are unsuitable as they will suspend until the context is determined. Nonetheless, Oz supports the notion of constraining the validity of constraints and we can use this ability to constrain the context of an item even if we can not directly access this context.

```
{MyScore
 forAll(RestrictMelodicInterval
        test:fun {$ X}
                {X isNote($)} andthen
                {X hasTimeAspectPredecessor($)}
              end)}
```

Fig. 5. Application of the melodic rule

In Oz, the validity of a constraint is reflected into a truth value by a reified constraint [9]. A 0/1-integer – a FD integer with the domain $\{0, 1\}$ – represents the truth values *false* or *true*. Reified constraints can be used to state logical connectives. For example, the Strasheela user can express: the fact that two notes are simultaneous implies that the pitches of these notes must form a consonant interval (8). As 'isSimultaneous' and 'isConsonant' are both reified constraints, the user can express this implication even when simultaneous notes are undetermined before the search. The scope of the rule implementing (8) are all note pairs which are possibly simultaneous in the solution.

$$(\text{isSimultaneous}(note_1, note_2) \rightarrow \text{isConsonant}(note_1, note_2)) \leftrightarrow true \quad (8)$$

Whether two items are simultaneous or not is formalised by reified constraints on their respective start and end time (9). In an implementation of (9), the validity b is a 0/1-integer. Whether two notes are consonant is formalised in a similar way by reified constraints stating whether the interval between the pitches of two notes is in {minor third, major third, fifth, ..., octave + major third} (10).

$$((start_1 < end_2) \wedge (start_2 < end_1)) \leftrightarrow b \quad (9)$$

$$(|pitch_1 - pitch_2| \in \{3, 4, 7, 8, 9, 12, 15, 16\}) \leftrightarrow b \quad (10)$$

The rules discussed so far restrict local relations between score objects. However, to specify aspects of the musical form such as motifs and their relations, a rule context may also range over a longer time span. A simple example of this kind is a rule which constrains the musical form to a canon by a pair-wise unification of the note durations and pitches of two voices.

5 Score Distribution Strategies

Constraint problems in Strasheela involve often hundreds or more constrained variables resulting in a huge search space. An efficient search strategy is therefore crucial to make Strasheela useful for a composer.

Oz employs a complete search strategy which is often referred to as *propagate and distribute* [9]. Constraint propagation reduces the variables' domains by removing the values that cannot satisfy the constraints. When no further

propagation happens, distribution decides for either some additional constraint on some variable or the complement of that constraint. That way, distribution restarts propagation. An important advantage of the Oz constraint programming model lies in the fact that this decision making process is fully programmable on a high level of abstraction: Oz allows to customise the search strategy according to the constraint problem.

Strasheela adapts this high-level means to define distribution strategies provided by Oz; the Strasheela user can easily define strategies to distribute score parameters. Such a distribution strategy has access to the whole score via each parameter, because the relations between an item and its parameters as well as a container and its contained items are bidirectional linked in the score representation. A distribution strategy aims to help constraint propagators to reduce the search space. To this end, a score distribution strategy often addresses with special care undetermined rule contexts. For instance, the constraints of a harmonic rule can only propagate and reduce the domain of note pitches after it is known which notes are simultaneous.

A few score distribution strategies are already predefined. A typical strategy first determines timing parameters, or determines parameters 'from left to right', that is in increasing order of the start times of their respective items. The latter strategy is explained in more detail in [17].

A distribution strategy not only effects efficiency. Also heuristics can be defined by distribution strategies, as the distribution strategy affects the order in which solutions are found. For instance, particularly useful for musical purposes are heuristics in which the distribution randomly decides in favour of a particular domain value.

6   The Canon Example

The above-mentioned rules established the starting point for a composer who extended the canon (Fig. 6) description to about 15 rules, many of which are inspired by [8].[2] The conjunction of all rules results in a complex search problem; the solution shown below is found in about 20 seconds (first solution found with a distribution strategy involving random on a 2GHz PC). However, a solution is found in only a single second in case some rule is excluded. Strasheela solutions can be output into several formats. Currently, the sound synthesis languages

---

[2] The composer controlled the rhythm (the canon starts and ends with long notes and note durations may change only slowly across a voice). She adjusted the melodic rules (only notes in c-major are allowed, the first and last pitch of the lower voice must be the fundamental, only jumps up a minor third are permitted) and extended the harmonic rule by voice-leading rules (passing notes are allowed, open parallels are not). The canon is changed into a canon in the fifth of the first n notes ($n = 10$ in Fig. 6). Perhaps the most important extension are rules which control the melodic contour, for example, which force the maximum and minimum pitch of each voice to occur only once.

Csound [18], SuperCollider [19], Common Lisp Music (CLM) [20], and the music notation software LilyPond [21] are supported.

Fig. 6. A canon example which applies about 15 rules

7   Related Work

Many constraint based CAC environments have been proposed [2,3,4,5,6]. This section discusses the Oz application COMPOzE and the environment PWConstraints.

7.1   COMPOzE

The composition system COMPOzE [22] generates a sequence of four-note chords to accompany multimedia presentations. The system expects as input a symbolic musical plan which consists of a harmonic progression and additional information. The harmonic progression is represented by harmonic functions in the tradition of the music theorist Hugo Riemann (e.g. $T s_3 D^7 T$). Additional information is used to restrict movements of single voices (e.g. "soprano melody shall move downward"). Besides this musical plan, COMPOzE's chord sequence output follows further compositional rules which are defined by the system and implement standard textbook rules on harmony.

COMPOzE represents music as a sequence of chords. Each chord consists of four notes and each note is represented by a FD integer denoting the pitch. The harmonic functions, voice movement restrictions and compositional rules are formulated as constraints on these pitches.

COMPOzE and Strasheela clearly have different goals. COMPOzE, on the one hand, formalises a certain sub-task of traditional music composition. COMPOzE solely constrains note pitches. The COMPOzE user adjusts the arguments of a predefined set of compositional rules applicable to four-voiced music.

Strasheela, on the other hand, does not predefine any general musical laws. Instead, Strasheela aims to provide the composer with a general tool to describe her own music by programming compositional rules from scratch. Strasheela offers means to represent and constrain music that is far more complex than a four-voiced chord progression. In particular, Strasheela supports polyphonic music where voices containing items such as notes, or chords run in parallel. More complex music is represented by further nesting of sequential and simultaneous containers. Besides note pitches, the whole timing structure and arbitrary additional parameters are constrainable.

7.2   PWConstraints and Score-PMC

PWConstraints [2, 6] is a library of the graphical programming language and
CAC environment PatchWork [2]. PWConstraints consists of two main layers:
a general constraint programming language (PMC) and an extension with spe-
cial support for polyphonic music (Score-PMC). The Score-PMC user prepares
in advance an arbitrary complex score to determine the rhythmic structure of
the final result. She defines compositional rules which constrain score parameter
relations (e.g. $7 \geq |\text{pitch}(note_1) - \text{pitch}(note_2)|$). The user states the scope of
each rule with a pattern matching expression (e.g. a pattern like $[* \, note_1 \, note_2]$
applies a rule to all consecutive note pairs in the score). Within the rule defini-
tion, the user often accesses some score context (e.g. the pitches of simultaneous
notes). PatchWork and PWConstraints are implemented in Common Lisp.

When I designed Strasheela, Score-PMC was one of the models I had in
mind: Strasheela aims at being more general than Score-PMC without loosing
efficiency. Important differences between the two environments are due to the
differences of their underlying constraint solvers. PWConstraints, on the one
hand, applies backtracking (with optional refinements such as forward check-
ing or backjumping): a constraint checks the validity of constrained variables
only *after* they are determined.[3] During search, the variables are determined in
an order which was fixed before the search started. In Oz, on the other hand,
constraint propagators prune the domains of constrained variables *before* their
values are determined. During search, the distributor decides which variable to
visit next only when it actually happens.

The Score-PMC user must fully predetermine the rhythmic structure of a
score before the search starts. The program needs this information to deduce its
static search order. Strasheela is more general: parameters which determine the
rhythmic structure are constrainable like all other parameters. The Strasheela
user may freely mix rhythmic rules with rules on, for instance, pitches, and rules
which interrelate timing parameters and pitches. Score distribution strategies
still allow an efficient search.

In the general language PMC, the domains of constrained variables consist
of arbitrary data (e.g. ratios representing microtonal frequency proportions or
nested lists representing whole musical sections). As constraints are only applied
to determined variables, any Lisp function returning a boolean can serve as
a constraint. In this respect, Oz is less expressive for the sake of efficiency.
Constrained variables are quasi typed (e.g. FD integers or finite sets) and only
specially defined propagators can constrain variables.

Score-PMC predefines several context accessors, but its design does not allow
the user to define her own accessors. For example, to create a canon the user
would wish to define an accessor for note sets which hold the same position in
different voices, as this context is not predefined by Score-PMC. The pattern
matching mechanism of Score-PMC to define the scope of a rule is convenient

---

[3] Forward checking rules complicate the situation, but most PWConstraints programs
   use plain backtracking.

mainly for melodic rules where notes occur in a sequential order. The Score-PMC user can not extend or change this mechanism, non-melodic rules are only expressible with the help of context accessors. Strasheela, however, allows the user to freely define new score accessors. The Strasheela user defines the scope of a rule by an arbitrary control structure. She could, for example, define her own pattern matching mechanism.

The polyphonic music representation of Score-PMC has a fixed hierarchic structure and a fixed set of score object types. In Strasheela, the hierarchic nesting is user defined and the class hierarchy is user extendable.

8   Discussion

The present paper argued that the Oz programming language is a highly suitable foundation for a computer aided composition (CAC) environment. The text introduced Strasheela, a composition environment implemented in Oz. Strasheela's design was outlined and the usage was shown in an application example.

Nonetheless, Strasheela is limited in some ways. Strasheela does not support arbitrary compositional rules, only score parameters are constrainable. In particular, the musical form is not freely constrainable as the hierarchic nesting of score containers and events must be fully determined before search. However, Strasheela allows to constrain the number of elements in a container by a 'trick': events with the duration 0 may be considered as non-existing.

Complex rhythms (e.g. nested tuplets) or complex microtonal music is best represented using fractions or real numbers for parameter values. The extendable Oz constraint model does support real-interval constraints. However, much more constraints are predefined for finite domain (FD) non-negative integers in Oz. Therefore, the predefined timing constraints (1) to (6) as well as related score API methods such as 'isSimultaneous' (9) are defined for FD integers and consequently the values of all timing parameters (i.e. all offset times, start times, durations and end times) are restricted to non-negative integers. As offset times are non-negative, they can only express pauses before items and not the overlapping of items.

Composers often want to formulate merely a preference instead of defining a strict rule. For instance, a composer might prefer small melodic intervals but still allow larger intervals. Also, composers wish to grade the importance of compositional rules such that less important rules might be neglected in an over-constraint situation. The Strasheela user may specify rule sets which allow the violation of rules a certain number of times or in certain situations using reified constraints. Also preferences (optionally graded in importance) can be expressed using best solution search: after a solution is found, further solutions are constrained to be better according to some user defined criterion. However, best solution search is often less efficient than searching for a single strict solution.

The score representation of Strasheela is rich and explicit. For instance, for every timed score item Strasheela introduces variables and propagators for four timing parameters. On the one hand, such an explicit representation makes a

score description very convenient. For instance, both the definition of a rhythmical rule constraining item durations and a relation such as 'isSimultaneous' which constrains start and end times are straightforward. On the other hand, this rich representation causes the search script to consume much memory during search. Nevertheless, the Strasheela user may use recomputation – a technique which substitutes computer memory for computation time – to solve problems which would not fit into the available memory otherwise.

Despite these shortcomings, Strasheela realises a highly expressive CAC environment. The present paper explained how Strasheela represents a score, how the composer defines compositional rules and how she applies them to the score. Compositional rules can restrict many aspects of the music including the rhythmic structure, the melodic structure and the harmonic structure. Strasheela combines this expressivity with an efficient search strategy.

Acknowledgements

References

1. Roads, C.: The Computer Music Tutorial. MIT press (1996)
2. Laurson, M.: Patchwork. A Visual Programming Language and Some Musical Applications. PhD thesis, Sibelius Academy (1996)
3. Anders, T.: Arno: Constraints Programming in Common Music. In: Proceedings of the 2000 International Computer Music Conference. (2000)
4. Truchet, C., Assayag, G., Codognet, P.: OMClouds, a heuristic solver for musical constraints. In: MIC2003: The Fifth Metaheuristics International Conference. (2003)
5. Sandred, O.: OpenMusic. RC library Tutorial. (2000)
6. Rueda, C., Lindberg, M., Laurson, M., Block, G., Assayag, G.: Integrating Constraint Programming in Visual Musical Composition Languages. In: ECAI 98 Workshop on Constraints for Artistic Applications, Brighton (1998)
7. van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
8. Motte, D.d.l.: Kontrapunkt. Bärenreiter-Verlag (1981)
9. Schulte, C., Smolka, G.: Finite Domain Constraint Programming in Oz. A Tutorial. (2004)
10. Selfridge-Field, E., ed.: Beyond MIDI. The Handbook of Musical Codes. MIT press (1997)
11. Dannenberg, R.B.: Music Representation Issues, Techniques, and Systems. Computer Music Journal **17(3)** (1993)

12. Wiggins, G., Miranda, E., Smaill, A., Harris, M.: Surveying Musical Representation Systems: A Framework for Evaluation. Computer Music Journal **17(3)** (1993)
13. Desain, P., Honing, H.: CLOSe to the edge? Advanced object oriented techniques in the representation of musical knowledge. Journal of New Music Research **2** (1997)
14. Dannenberg, R.B.: The Canon Score Language. Computer Music Journal (1989)
15. Dannenberg, R.B., Desain, P., Honing, H.: Programming language design for music. In Poli, G.D., Picialli, A., Pope, S.T., Roads, C., eds.: Musical Signal Processing. Lisse: Swets & Zeitlinger (1997)
16. Müller, T.: Problem Solving with Finite Set Constraints in Oz. A Tutorial. (2004)
17. Anders, T.: A wizard's aid: efficient music constraint programming with Oz. In: Proceedings of the 2002 International Computer Music Conference. (2002)
18. Boulanger, R., ed.: The Csound Book. Perspectives in Software Synthesis, Sound Desing, Signal Processing, and Programming. The MIT Press (2000)
19. McCartney, J.: Rethinking the Computer Music Language: SuperCollider. Computer Music Journal **26(4)** (2002)
20. Schottstaedt, B.: CLM. (`http://ccrma-www.stanford.edu/software/clm/`)
21. Nienhuys, H.W., Nieuwenhuizen, J.: LilyPond . . . music notation for everyone. (`http://lilypond.org/`)
22. Henz, M., Lauer, S., Zimmermann, D.: COMPOzE — intention-based music composition through constraint programming. In: Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence. (1996)