EXTENSIBLE CONSTRAINT SYNTAX THROUGH SCORE ACCESSORS

Mikael Laurson
Sibelius Academy, CMT
laurson@siba.fi

Mika Kuuskankare
Sibelius Academy, DocMus
mkuuskan@siba.fi

ABSTRACT

We present in this paper our recent developments dealing with constraint-based programming. Our focus is in a new syntax that extends the pattern-matching part of our rule system. The syntax allows to refer to more high-level entities in a score than before, resulting in compact rules that use only a minimal set of primitives. The system can be used to define a wide range of cases ranging from melodic, harmonic and voice-leading rules. The compiler can be extended to support new score accessor keywords by special compiler methods. The new syntax is explained and demonstrated with the help of a large number of rule examples.

1. INTRODUCTION

When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp, typically the user has to solve a problem in a stepwise manner. In many cases this approach is an adequate one, but for many types of problem it may lead to programs that are difficult to design or understand.
Descriptive languages, such as Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules. It is then up to the language to find solutions that are coherent with the descriptions. This approach is probably more natural for individuals with a musical background. A typical music-theoretical writing offers a discussion on some properties of some pieces of music, not a step-by-step description of how those pieces were made.
PWConstraints [1] can be thought of as a descriptive language. PWConstraints is written in Common Lisp and CLOS. When using it we do not formulate stepwise algorithms, but define a *search-space* and produce systematically potential results from it. Typically we are not interested in all possible results, but filter (or, rather, constrain) these with the help of rules describing an acceptable solution.
There are currently several other constraint-based approaches dealing with musical problems such as Situation [2], Arno [3], OMClouds [4] and the more recent system by Anders based on the OZ programming language [5]. In some respects these systems are similar as they all apply user defined rules to find a solution. They differ, however, considerably in their approach on how the solution is found and how rules are formulated by the user: for instance PWConstraints and Arno are similar as they both normally apply strict rules that must all be satisfied in the result (PWConstraints supports heuristic rules but they belong to a special rule category); OMClouds, by contrast, utilizes systematically a heuristic approach where the system tries to find an approximate solution that fulfils the given criteria as close as possible; rules are formulated in PWConstraints using a pattern-matching language whereas in OMClouds the constraints are expressed in logical forms that are translated into cost functions; finally—as an interesting feature from the point of view of this article—both PWConstraints and Anders' systems support score structures that allow to solve polyphonic search problems (Andres' system is especially interesting in this respect as it allows to solve both the rhythm and the pitch structure simultaneously within in search). A good comparison of different music oriented constraint-based systems can be found in [5].
A specific problem in the musical domain is that the task of describing a musical result is far from trivial. As such a result is typically seen from many different points of view, we need highly flexible data structures to describe musical structures. Each rule should be able to dynamically extract required information out of a data structure, allowing the rule to analyse a result from its own point of view. One of the main problems in formulating such rules is to find a clear formalism with which to point to the required data objects. This article presents a new approach that allows to access in a uniform way various structural entities from a score, such as chords, beats, measures and harmonic formations.
In the following we first introduce and compare the two main components of our constraint-based system: *PMC* and *Score-PMC* (Section 2). We discuss some problems in the current Score-PMC implementation and propose a new syntax that allows to add in the rules score accessor keywords (Section 3). Finally the new syntax is utilized to write melodic, harmonic, voice-leading and special score-sort rules (Sections 4, 5, 6 and 7).

2. BACKGROUND

PMC [1] is currently an integral part of our visual programming environment called PWGL [6]. A search-space is defined as a set of search-variables. Each search-variable has a domain containing a list of values. In a rule a pattern-matching language is used to extract relevant information from a potential solution. This information is given to a Lisp test function, called *Lisp-code part*, that either accepts or rejects the current choice made by the search-engine. The rules are compiled to efficient Lisp functions.

The *pattern-matching part* of a rule uses a fairly typical pattern-matching syntax. It can contain *variables* (symbols starting with a '?'), *anonymous-variables* (plain '?'s), a *wild card* ('*') and *index-variables* (symbols consisting of an 'i' and an index number). A variable extracts single values from a partial solution. By contrast, an anonymous-variable is never bound to a value, i.e. it only acts as a 'place-holder' in the pattern. The wild card matches any continuous part of a partial solution. Finally, an index-variable extracts values from an absolute position. Below we give some pattern-matching examples with their respective bindings (in order to clarify the examples the pattern below the input is formatted with the help of spaces so that it matches its input):

```
input:   ( 1  2  3  4  5)
pattern: (?1  ?  ? ?2  ?)
match:    ?1 = 1, ?2 = 4

input:   (1  2  3  4  5)
pattern: (i1 i2      i5)
match:   i1 = 1, i2 = 2, i5 = 5

input:   (1  2  3  4  5)
pattern: (*          ?1)
match:    * = (1  2  3  4), ?1 = 5

input:   (1  2  3  4)
pattern: (*    ?1 ?2)
match:    * = (1  2), ?1 = 3, ?2 = 4

input:   (1  2  3  4  5  6  7)
pattern: (?1 *          ?2 ?3)
match:    * = (2 3 4 5), ?1 = 1, ?2 = 6, ?3 = 7
```

As an example a PMC rule disallowing two adjacent equal values in a result can be written as follows (this rule uses a wild card and two variables):

```
(* ?1 ?2                    ;;pattern-matching part
  (?if (/= ?1 ?2))          ;;Lisp-code part
  "no equal adjacent values")
```

Besides these basic tools our system contains several extensions. The most important and complex one is used to solve polyphonic search problems. This is accomplished with a function called Score-PMC. Like in a traditional polyphonic score, the user operates with several layers (parts, voices) of events (notes). The rhythmic structure of a search problem is prepared in advance in a standard PWGL score-editor. This input score, which can be arbitrary complex, is given as an argument to the search engine. The search, in turn, aims at filling the input score with pitch information according to the given rules. In a sense Score-PMC can be seen as a multi-layered search problem where each melodic line represents one queue structure similar to the one used by a simple PMC search problem (Figure 1):

```
   (1) PMC:
      v1, v2, v3, ….., vN
   (2) Score-PMC (3-part case):
     v11, v12, v13, ….., v1N
     v21, v22, v23, ….., v2N
     v31, v32, v33, ….., v3N
```

Figure 1. A PMC queue structure (1) compared with a 3-part multilayered Score-PMC example (2).

Score-PMC uses a similar syntax when compared to PMC. The main change is that now the variables in the pattern-matching part always refer to note objects. This scheme is useful as the note objects of the input score contain potential information of the current musical context, such as instrument, part, pitch, metrical position and harmony. This knowledge can be used to define for instance traditional counterpoint rules [1]. Score-PMC has been used to solve several large scale and concrete musical problems. For more details see [1] and [2].

The previous PMC rule example can be translated into Score-PMC syntax as follows:

```
(* ?1 ?2                       ;;pattern-matching part
  (?if (/= (m ?1) (m ?2))) ;;Lisp-code part
  "no equal adjacent melodic pitches")
```

The expressions '`(m ?1)`' and '`(m ?2)`' denote now to the pitch-values of the *notes* referred by the variables '`?1`' and '`?2`' ('`m`' stands for 'midi'). Similarly, harmonic rules can be defined using the functions '`hc`' and '`h-slice`' (short-hands for 'harmonic context' and 'harmonic slice'); rules that refer to the metric position of a note can be defined using the function '`rtm-match?`'; and so on. This rule is applied to all melodic lines found in the score. If we want to apply the rule only to parts 1 and 3 we can modify it by adding the keyword '`:partnum`' in the pattern-matching part:

```
(* ?1 ?2 :partnum (1 3)    ;;pattern-matching part
  (?if (/= (m ?1) (m ?2))) ;;Lisp-code part
  "no equal adjacent melodic pitches in parts 1 and 3")
```

While this system has proven to be powerful and efficient, Score-PMC rules can occasionally be quite hard to define and understand. One reason for this difficulty is the fact that the variables always refer to the most low-level entity of the score (i.e. note objects), even if the rule works with more high-level concepts like chords, beats, measures or harmonies. This feature—i.e. lack of structural accessors—has been criticized in [5]. Another difficulty with Score-PMC rules is that the system strongly prefers melodic formations (see Figure 1 and the previous Score-PMC rule) and the pattern-matching part is useful mainly when writing melodic rules. If the user wants to write for instance a harmonic rule then the required structural information has to be accessed in the Lisp-code part by using special Lisp help functions like 'hc', 'h-slice', 'hc-midis', 'prev-item', etc. Thus when writing non-melodic rules the user has to have knowledge of the implementation of the system and master a large library of help functions. Finally, as one has to be conscious of implementation details the coding of new rules or Lisp help functions can become a complex and error prone process.
This paper presents a new Score-PMC pattern-matching syntax that allows the user to specify special *score accessors*, which provide the user with a more high-level and intuitive approach when working with Score-PMC rules. This syntax is compatible with the old one and rules using the old syntax are fully functional in the

new system. The variables given in the pattern-matching part of a rule can now refer to the structural entity the user is interested in, such as note, chord, beat, measure and harmony. The aim is to deal with the most complex part of the system automatically in the compilation process (i.e. when rules are translated into Lisp functions). The novel compiler is modular and new score accessors can be added incrementally.

The new syntax has many benefits: rules tend to be more compact and simple; the pattern matching language can be used systematically for all kinds of valid accessors; the user typically needs to know only a handful primitives in order to write new rules; potential bugs can be localized more easily as the most complex part of the system is localized in the compiler and not in user code.

3. SCORE ACCESSOR SYNTAX

In the new syntax the compiler accepts an optional score accessor keyword that defines the type of the variables that are declared in the pattern-matching part. If the rule has no accessor keyword then the compiler assumes that the variables refer to note objects. For instance in the "no equal adjacent melodic pitches" rule the variables '?1' and '?2' are notes. Thus all melodic rules written in the old syntax will work as before. Valid accessor keywords are: ':chord', ':beat', ':measure', ':harmony', and ':score-sort'. This list can be extended by adding appropriate compiler methods for the new accessor keyword. Extending the compiler can be quite complex and will not be discussed any further in this article. In the following we will only consider the existing list of keywords and describe how they can be used to write Score-PMC rules.
Another important change in our syntax is the function 'm'—which was already mentioned in the previous rule examples—that used to return the pitch-value (i.e. the MIDI key-number) of a note. The 'm' function is of primary importance as a Score-PMC search problem ultimately deals with pitch information. Now 'm' is defined as a method where the receiver or the first argument can be any variable type defined by the score accessor keyword. This means that the behaviour of 'm' depends on its first argument. For instance if a variable is a note then 'm' returns a single numeric value; if the variable is ':chord', ':beat', ':measure', or ':harmony' then 'm' returns a list of pitch-values (':score-sort' is somewhat special and will be discussed later). The 'm' method accepts also a number of optional keyword arguments that greatly enhances the functionality of this method. Keyword arguments can be used either to modify or filter the result or they can return a flag (i.e. either true or false). Currently the following keyword arguments are supported: ':int', ':harm-int', ':min', ':max', ':part', ':complete-case?' and ':prev-item'. For instance the ':int' keyword allows to convert a list of pitch-values into a list of intervals; ':min' results in the minimum and ':max' in the maximum value of a pitch-value list; ':part' filters pitch-values so that 'm' returns only a pitch-value belonging to a given part. Several of these keywords will be discussed in more detail in conjunction with rule examples.
The ':complete-case?' keyword is more special and needs some further explanation. When working with compound structures consisting of several notes such as chords, beats, measures and harmonic formations, the search-engine calls the rules each time it encounters a new note in the structure. Thus the rules have most of the time to deal with *partial solutions* (i.e. cases where only some of the notes belonging to the current structure have a solution). Normally this is not a problem as the 'm' method by default returns only values that have a solution. The 'complete-case?' keyword is typically used only in rules where it is required that the structure is fully solved (Section 6 gives such an example) .

4. MELODIC RULES

In the following sections we will demonstrate how score accessors can be used to write Score-PMC rules. Section 2 gave already two simple melodic rule examples without a score accessor keyword (i.e. in this case we assume that all variables in the pattern-matching part refer to notes). In this section we discuss the score accessor keywords ':beat' and ':measure' which can be used to define melodic rules where variables denote either beats or measures.

Our first example uses the accessor ':beat' and thus the '?1' variable refers to a beat. This rule is applied to all beats in the input score and therefore all beats in the final result should fulfil this rule. The 'm' method returns all pitch-values for all notes of the current beat that have a solution. Finally the function 'setp' checks that these pitches do not contain modulo 12 duplicates:

```
(* ?1  :beat
   (?if  (setp (m ?1) :key 'mod12))
  "no mod12 pitch dups in beat")
```

Our next example is very similar: the only change is the accessor which is now ':measure' (i.e. all *measures* should fulfil the rule):

```
(* ?1  :measure
   (?if  (setp (m ?1) :key 'mod12))
  "no mod12 pitch dups in measure")
```

The next example contains two main changes when compared to the previous one. First, the pattern-matching part has two variables instead of one. Second, the rule contains the keyword ':partnum'. This means that this rule is applied for each *measure pair* only for part number 3. The Lisp-code part checks that the pitches in the two adjacent measures are never equal:

```
(* ?1 ?2 :measure :partnum 3
   (?if  (not (equal (m ?1) (m ?2)))
  "no equal pitch contents in adjacent measures in part
  3")
```

Our last example in this section demonstrates the use of the ':int' keyword when accessing pitches from score objects. Now we convert the pitch-value information to intervals and the rule forbids all cases where two adjacent measures have equal interval lists:

```
(* ?1 ?2 :measure :partnum (1 3)
   (?if
    (let ((ints1 (m ?1 :int t)) (ints2 (m ?2 :int t)))
      (if (and ints1 ints2)
       (not (equal ints1 ints2))
        t)))
  "no equal pitch interval contents in adjacent measures
for parts 1 and 3")
```

5. HARMONIC RULES

The score accessor keyword ':harmony' is especially interesting for us in this article as it allows to formulate both harmonic and voice-leading rules in a very compact manner. In this section we focus in examples that deal with harmony. With the term harmony we mean here vertical pitch formations in the score where one, two or more notes are sounding together (in a typical case the size of the smallest harmonic formation is equal to two). Let us start with a simple rule that forbids any harmonic modulo 12 duplicates (i.e. the harmonic formations should not contain pitch duplicates or octaves):

```
(* ?1  :harmony
  (?if (setp (m ?1) :key 'mod12))
 "harmonic duplicate rule")
```

What is interesting in our new syntax is that this rule is very close to the first rule in the previous section. The only change is the score accessor keyword which is now ':harmony'.
Our next rule demonstrates the use of the ':harm-int' keyword in conjunction with the 'm' method. In this case 'm' first sorts all pitch-values in ascending order and after this calculates a difference list (thus the resulting interval list contains only positive values). The Lisp-code part states that all harmonic modulo 12 intervals should be unique.

```
 (* ?1  :harmony
   (?if (setp (m ?1 :harm-int t) :key 'mod12))
  "harmonic interval duplicate rule")
```

The following rule is similar as it uses the ':harm-int' keyword. The difference is however that now the pattern-matching part contains three variables. Thus this rule is applied to all sequences that consists of three adjacent harmonic formations. The rule disallows any harmonic interval duplicates within such a sequence.

```
(* ?1 ?2 ?3 :harmony
 (?if
    (let ((ch-int1 (m ?1 :harm-int t))
          (ch-int2 (m ?2 :harm-int t))
          (ch-int3 (m ?3 :harm-int t)))
   (if ch-int3
       (and  (not (equal ch-int1 ch-int2))
             (not (equal ch-int2 ch-int3))
             (not (equal ch-int1 ch-int3)))
      t)))
     "3 harmonic succession duplicate interval rule")
```

6. VOICE-LEADING RULES

Voice-leading rules tend to be harder to formulate than melodic or harmonic ones as they often deal both with melodic and harmonic formations in the same rule. The required musical context can be spread among several parts in the score. This can lead to very difficult situations especially if the rhythmic structure of the input score contains complex poly-rhythms.

In this section we continue to use the ':harmony' score accessor as it often allows to capture these cases in a compact manner.

We start with a rule that disallows any voice crossings (thus for example all pitches in part 2 should be lower that in part 1). This rule is quite complex as it operates with pitch-values *and* part numbers. First we need to access the current part number. This is done with the expression '(partnum ?csv)' where '?csv' is a reserved symbol that is always bound to the current search-variable. Thus 'p1' contains the current part number and we calculate a part number 'p2' that is below the current one by adding 1 to the current one (we assume here that the highest or the top-most part has the part number 1). Next we need to access the pitch-values for 'p1' and 'p2'. This is done by giving the 'm' method the keyword ':part' which returns the pitch-value for this specific part. It is important to note that 'm' can also return (), which means that the requested part is not existing in the current harmonic formation. Thus we need to check that both pitch-values are not (), i.e. that both parts are found. (This is done in the expression '(and m1 m2)'). If this is the case then we check that 'm1' is greater than 'm2'. Otherwise we return t as it makes no sense applying the rule if both parts are not present. This rule is general and works with a score with arbitrarily many parts.

```
(* ?1 :harmony
  (?if (let* ((p1 (partnum ?csv)) (p2 (+ p1 1))
              (m1 (m ?1 :part p1)) (m2 (m ?1 :part p2)))
        (if (and m1 m2)
          (> m1 m2)
          t)))
 "no voice-crossings")
```

Often it is interesting to use a rule that is applied only to some of the parts. In the next rule, which is almost identical with the previous one, we disallow voice crossings only for two parts which are given in 'p1' and 'p2' (in our example 'p1' is 1 and 'p2' is 2).

```
(* ?1 :harmony
  (?if (let* ((p1 1) (p2 2)
              (m1 (m ?1 :part p1)) (m2 (m ?1 :part p2)))
        (if (and m1 m2)
          (> m1 m2)
          t)))
 "no part p1+p2 voice-crossings")
```

The next example demonstrates how to write a rule that disallows any modulo 12 *cross-relations* between the highest and lowest pitches in a sequence of two adjacent harmonic formations (see Figure 2).

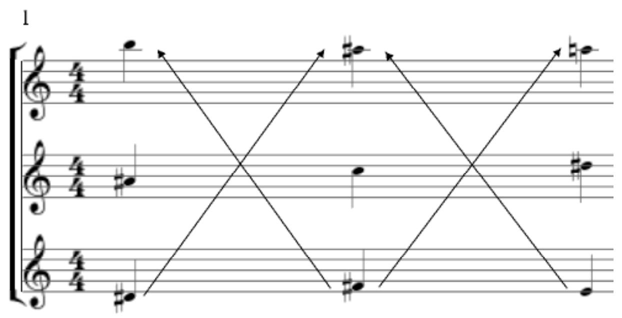

Figure 2. A 3-part score where cross-relations between soprano and bass parts are denoted with arrows.

The pattern-matching part of the rule thus contains two variables (‘?1’ and ‘?2’). As we can run this rule only if both harmonic formations are fully solved we must first check that the latter one (‘?2’) is finished using the expression ‘(m ?2 :complete-case?)’. If this is the case we extract with the keywords ‘:min’ and ‘:max’ the minimum and maximum pitch-values from both harmonic formations. Finally we check that these pitch-values do not form modulo 12 cross-relations (see also Figure 2).

```
(* ?1 ?2 :harmony
   (?if (if (m ?2 :complete-case?)
      (let* ((sop1 (m ?1 :max t)) (sop2 (m ?2 :max t))
             (bas1 (m ?1 :min t)) (bas2 (m ?2 :min t)))
         (and (/= (mod12 sop1) (mod12 bas2))
              (/= (mod12 sop2) (mod12 bas1))))
        t))
   "no sop/bas mod12 cross-relation")
```

While the previous rule was fixed for soprano and bass parts only, the next rule can be applied for any set of two parts. This can be accomplished with the keyword ‘:part’ (in this example we forbid modulo 12 cross-relations for parts 1 and 3, i.e. ‘p1’ is 1 and ‘p2’ is 3). Also here we must check whether all parts are present with the expression ‘(and m11 m12 m21 m22)’.

```
(* ?1 ?2 :harmony
  (?if (let*((p1 1) (p2 3)
             (m11 (m ?1 :part p1))
             (m12 (m ?2 :part p1))
             (m21 (m ?1 :part p2))
             (m22 (m ?2 :part p2)))
      (if (and m11 m12 m21 m22)
        (and (/= (mod12 m11) (mod12 m22))
             (/= (mod12 m12) (mod12 m21)))
          t)))
   "no mod12 cross-relation in parts p1+p2")
```

In order to demonstrate the power of the pattern-matching syntax we can make an interesting variation of the previous rules by changing the pattern-matching part to the following:

```
(* ?1 ? ?2 :harmony  . . . .)
```

This minor change handles now cross-relations that work in a context of *three* successive harmonic formations—instead of two—where the rule is applied to the first and third formations (Figure 3):

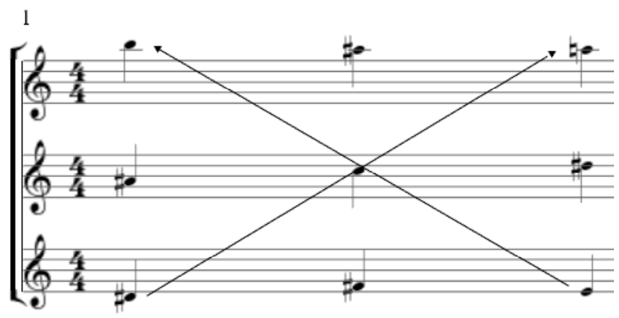

Figure 3. A 3-part score with a ‘stretched out’ cross-relation.

7. SCORE-SORT ACCESSOR

While the score accessors keywords presented in the previous sections (i.e. ‘:beat’, ‘:measure’ and ‘:harmony’) are quite obvious choices for our syntax, the compiler can be extended with more rare and ‘exotic’ cases. This section presents a ‘:score-sort’ score accessor keyword that can be used in a similar manner than for instance the melodic rules that were presented in the beginning of this article.

Score-sort is an ordering of the note objects of the input score that is used internally by the search engine. While score-sort is normally not accessed by the user it can have an interest in a musical context. The ordering is accomplished as follows: We read a score from left to right and sort notes in the order they appear in it. If two or several notes share the same attack time, they are sorted so that the longest notes are placed before the shorter ones. If two or more notes have the same attack time and the same duration, we can order them freely. We use the convention that notes having the highest part number are considered first. Figure 4 shows a score and the score-sort order is shown with a curve connecting consecutive notes. The score-sort can be called a kind of ‘inter-melodic’ formation where the notes—which are typically distributed between several parts—are processed in the order they appear in a sequence defined by the score-sort algorithm.

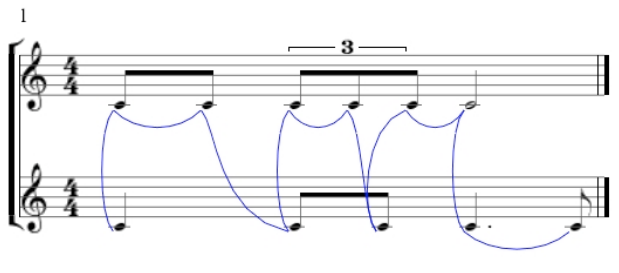

Figure 4. A score-sort ordering example. The ordering is shown with a Bezier function starting from the left-most note in part 2 and ending with the right-most note in part 2.

A score-sort rule is written using the ‘:score-sort’ score accessor keyword. For instance a rule where all 3 note score-note successions should not form a major or minor triad can be written as follows (we use here a pitch-class set theoretical approach where ‘3-11b’ and ‘3-11a’ stand for major and minor triads):

```
(* ?1 ?2 ?3 :score-sort
  (?if (not (eq-set '(3-11b 3-11a) (m ?1)(m ?2)(m ?3))))
  "no score-sort major/minor triads")
```

8. CONCLUSIONS

This paper presents a new syntax that allows to write in a flexible manner Score-PMC rules that are applicable to a wide range of musical contexts. The pattern-matching part can be used systematically for all score accessor types. The most complex part of the system is now localized in the compiler which means that user written rules are much simpler and easier to maintain than before. New score accessors can be added by special compilation methods.

While the new syntax is functional it is clear that there is still much to be done. The system must be tested with a larger corpus of rules. This task will be much easier than before as our system allows to add incrementally new score accessor keywords. Also the second main component, the 'm' method, is easily extensible in order to support new accessor types. The other main issue is how this development will affect the third main component of our constraint-based system called *Texture-PMC* [7]. Here a again the dynamic nature of our compiler should turn out to be useful. Probably this will mean that new rhythm and texture related accessors will be incorporated in the system.

9. ACKNOWLEDGEMENTS

The work of Mikael Laurson has been supported by the Academy of Finland (SA 105557).

10. REFERENCES

[1] Laurson, M. PATCHWORK: *A Visual Programming Language and Some Musical Applications*. Doctoral dissertation, Sibelius Academy, Helsinki, Finland, 1996.

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

[3] Anders T. "Arno: Constraints Programming in Common Music", *Proceedings of the International Computer Music Conference*, 2000.

[4] Truchet C., G. Assayag, and P. Codognet. "Visual and Adaptive Constraint Programming in Music", *Proceedings of the International Computer Music Conference*, Havana, Cuba, pp. 346-352 , 2001.

[5] Anders, T. *Composing Music by Composing Rules: Computer Aided Composition employing Constraint Logic Programming.* Sonic Arts Research Centre Queens University Belfast, Northern Ireland, 2003.

[6] Laurson, M., and M. Kuuskankare. "PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL", *Proceedings of the International Computer Music Conference.* Gothenburg, Sweden, pp. 142–145, 2002.

[7] Laurson M., and M. Kuuskankare. "A Constraint Based Approach to Musical Textures and Instrumental Writing", In *CP01 workshop on Musical Constraints*, Cyprus, 2001.