Intelligent Scripting in ENP using PWConstraints

Mika Kuuskankare
DocMus, Sibelius Academy
mkuuskan@siba.fi

Mikael Laurson
CMT, Sibelius Academy
laurson@siba.fi

Abstract

In this paper we are going to focus on how a constraint-based language, PWConstraints, can be adapted so that it can be used to script our music notation package, called Expressive Notation Package (ENP). We briefly introduce the basic concepts behind PWConstraints and give an overview of the scripting properties of ENP. We also present and examine in detail a number of example scripts.

1 Background

Scripting languages are programming languages specific to a particular program. They are usually weakly typed or untyped and interpreted rather than compiled. A scripting program, a script, is normally used to automate complex or advanced tasks within the program. Scripts typically interact either with other programs or with a set of functions provided by the interpreter. Scripts are usually stored in ASCII format, thus they can be distributed across different computer platforms. This is an important feature for cross-platform computer applications.

A practical application of a script is to program either repetitive or otherwise laborious tasks so that they can be executed in a single pass. In a music notation program such a task could be to apply a certain articulation pattern to a passage of music, to recalculate the enharmonic identity of selected notes, etc.

There are some commercial music notation programs that provide a scripting language and/or plug-in interface that allows the user to modify the information in the score. In Sibelius, for example, the user can write scripts by using the built-in scripting language called ManuScript. The ManuScript syntax resembles, in many aspects, Java and C++.

ENP (Kuuskankare and Laurson 2002) is a music notation program that we have developed in order to meet the requirements of computer aided composition, music analysis and virtual instrument control. ENP is programmed with Lisp and OpenGL (Woo et al. 1999). It’s purpose is to display scores using the traditional western notation. Furthermore, ENP provides a rich set of both standard and user definable notational attributes, called ENP-expressions, which range from standard articulation marks to a fully editable multi-purpose graphical expression, called score-BPF. All ENP-expressions are dynamic, i.e., they adjust their visual appearance according to their notational context.

PWConstraints (Laurson 1996), in turn, is a family of constraint-based programming tools that are aimed at solving a wide range of constraint satisfaction problems. Constraint satisfaction is a general problem in which the goal is to find values for a set of variables that will satisfy a given set of constraints (Leler 1988). The musical applications of PWConstraints are closely related to ENP and PWConstraints can access the data structures behind the ENP notation front-end.

In the following we compare some important features found in other scripting languages, such as ManuScript, to our scripting system in ENP. (1) Usually, when writing a script, the user has to write code to handle looping through the notational objects in the score. In our approach the user never sees the ‘main loop’ as it is handled by PWConstraints. (2) In case of ManuScript, for example, the scripts are not undoable. In order to revert the effect of the script one must manually remove the attributes one by one. In ENP, on the other hand, the system records knowledge about the individual runs so that the user can later identify and remove the attributes in any combination (selective undo). (3) In most of the music notation programs the notational attributes are not dynamic. Let us assume a case where the user adds some analysis information to the score with the help of a script. Whenever the related musical information changes the analysis results will become invalid. Normally, the user is required to remove the old attributes by hand and run the script again. In ENP, however, the attributes are dynamic (when applicable). Once inserted they adjust themselves according to the changes made in the music notation. (4) Most scripting languages lack the ability of PWConstraints to easily define and identify complex musical patterns with the help of a pattern-matching syntax.

The rest of the paper is structured as follows. First we
give an overview of PWConstraints and the pattern-matching language. Next we discuss the scripting in ENP and give an outline of the scripting properties. We end the paper with some example scripts and concluding remarks.

2 Overview of the PWConstraints

PWConstraints is a general-purpose rule-based programming language where the user writes rules to describe the end result from a number of different viewpoints. 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 that either accepts or rejects the current choice made by the search-engine (i.e., the rules return a truth value).

A PWConstraints rule consists of two parts: the pattern-matching part and the Lisp-code part. A special symbol ?if is used in the rule to distinguish between the pattern-matching part and the Lisp-code part.

The pattern-matching part consists of several different pattern-matching variables. The variables are named as follows:

1. Named variable (e.g., ?1, ?note1, ?note2) is used to extract single search-variables from a partial solution. The name (the portion of the text after the question mark) is user definable.

2. Anonymous variable (?) is used when a search-variable is required to define a pattern but its value can be ignored.

3. Wild card (*) matches any continuous part of a partial solution. In the pattern-matching part the wild card can occur only once.

4. Index variable (e.g., i1, i12) extracts search-variables from an absolute position in the partial solution.

The following gives an example of a PWConstraints rule. This rule forces an ascending series of numbers:

```
(* ?1 ?2 (?if (< ?1 ?2)))
```

The pattern-matching part contains a wild card and two named variables. The rule loops through every adjacent pair of search-variables (indicated by ?1 and ?2) in the given search-space. The Lisp-code part, in turn, applies a test function (< ?1 ?2) to each pair and returns a truth value.

PWConstraints contains also a collection of more specialized tools that extend the behavior of the basic scheme described above (Laurson and Kuuskankare 2001). These tools include Score-PMC, Texture-PMC, and MA-PMC (MA stands for music analysis). Score-PMC is used to solve polyphonic search problems. An input score (an ENP score with the rhythmic structure prepared in advance) 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. Texture-PMC, in turn, is used to produce rhythms and textures. Finally, MA-PMC is used to analyze existing rhythmic, melodic and harmonic material in a score. Unlike the former cases MA-PMC does not use backtracking.

In all aforementioned tools a search-variable represents a note in the input score. In these cases the search-variables contain knowledge about pitch, rhythm, harmony, etc. This knowledge can be accessed in a rule with a set of specialized Lisp-functions.

In this study, MA-PMC is used as a starting point to implement our scripting language. This approach is further explained in the following Section.

3 Scripting in ENP

The scripting language of ENP uses MA-PMC to provide the basic ability to map through the notational objects in the score. ENP script can contain an arbitrary number of scripting rules. In case of the scripting rule the pattern-matching syntax does not require any changes when compared to an ordinary PWConstraints rule. The Lisp-code part, however, does not return a truth value. Instead it contains a list of Lisp expressions that perform various side effects to the score such as adding expressions. The approach described here has several advantages:

1. The syntax of the pattern-matching language is compact and powerful.

2. There is a built-in mechanism in MA-PMC to map through all the notational objects in an ENP score. There is no need for the user to write tedious loop constructs.

3. MA-PMC provides rich knowledge about the melodic, rhythmic and harmonic properties of the score. This information can be used in the scripts.

ENP provides a set of specialized Lisp-functions that can be used to manipulate the appearance and content of ENP objects. Virtually every notational attribute can be accessed and manipulated with the scripts. Among the scripting properties of ENP are: color; channel and velocity; note-head type; enharmonic identity; time-signature, key-signature and tempo; start-time and duration; x-offset and y-offset, etc.
4 Some Scripting Examples

In this section we examine some ENP scripts in detail. Due to space limitations the user-interface is not shown here. The ENP script-editor, however, contains a text editor to input the script code, a few options to choose from (e.g., if the script is applied to a selection only) and a list of the scripts already applied with the possibility for selective undo.

4.1 Adding an Articulation Pattern

In our first example we use a script to insert a repeating articulation pattern to a passage of sixteenth notes. In this case the selected articulation pattern is *slur-staccato-staccato*. This particular example underlines the fundamental idea behind scripting: to perform a potentially laborious and repetitive task. We begin by defining the script:

```lisp
(* ?1 ?2 ?3 ?4 ;1
  (?if (when (downbeat? ?1) ;2
       (add-expression 'slur ?1 ?2) ;3
       (add-expression 'staccato ?3) ;4
       (add-expression 'staccato ?4)))) ;5
```

The first row of the script (1) shows the pattern-matching part containing a wild card and four named variables (?1, ?2, ?3 and ?4). The second row (the beginning of the Lisp-code part), in turn, checks if the first of the four notes falls on a downbeat. Provided that it does, the next row (3) of the script adds a slur to the first two notes, ?1 and ?2, and the remaining two rows (4 and 5) add staccatos to the last two notes, ?3 and ?4, respectively. Figure 1 gives the output of the script. The benefits of this kind of a script are naturally more obvious when the musical passage in question is very long.

![Figure 1: An articulation pattern is applied to passage of music with the help of a script.](image)

4.2 Adding Analysis Information

The second script analyzes the given melodic line by calculating the set class identity of every group of three consecutive notes (see Figure 2). This example demonstrates the ability of the script to access the information contained by the notational objects. The script is defined as follows:

```lisp
(* ?1 ?2 ?3 ;1
  (?if (add-expression 'group ?1 ?2 ?3) ;2
       :info (set-class-name ?1 ?2 ?3))) ;3
```

In this case, the pattern-matching part (1) shows a wild card and a group of three named variables (?1, ?2 and ?3). The first row of the Lisp-code part (2) adds a special expression, called group, to the three notes. The set-class identity of the notes is supplied as a parameter for the group indicated by the keyword :info (3).

![Figure 2: The melodic excerpt is analyzed by the script and the analysis information is added to the score accordingly (Berg: Lyrical Suite).](image)

It is to be noted, however, that in this case the information added to the score is static. Should the user, for example, transpose any of the notes in Figure 2, the analysis information shown by the expressions would become incorrect. ENP provides dynamic expressions for this kind of analytical purposes, and they should be preferred over the static ones in practical applications. The next two examples show the advantage of dynamic expressions used along with scripting.

4.3 Adding Trumpet Fingerings

There is sometimes the need to explicitly display the fingering information of different instruments in a score. This could be done for pedagogical purposes or for providing control information for a virtual instrument model (see for example Laurson et al. 2001).

The script for adding trumpet fingerings for each note in the score is defined as follows:

```lisp
(* ?1 (?if (add-expression 'trumpet-fingering ?1)))
```

Figure 3 shows the output of the script. This may seem trivial at first but the complexity of the case, however, becomes apparent after the user changes the musical information, for example, by transposing. In Figure 4 the user has transposed the musical material by a whole-step. This shows that the fingering expressions inserted by the script are dynamic, i.e., they adjust their visual representation according to the notational objects that they are attached to. There is no need to double check the notation or re-run the scripts in order to keep the score up to date.

1 A group is a specialized expression that displays any user definable textual information as a part of the music notation. The extent of the group is also visually indicated in the score.
4.4 An Advanced Example

We conclude with a relatively complex example. Here we use an excerpt from opus 25 by Arnold Schoenberg as a starting point. The aim is to fill a score with some of the articulations that are found in the original score with the help of a script. To make the example more straightforward we concentrate on two types of articulations, grave accents and slurs. All other attributes that are found in the original are omitted. In the original score there are always slurs between two notes if they form a large interval (the adjacent notes are more than an octave apart) provided that there is not a rest between them. Also, each sixteenth note has a grave accent. In this case we use the musical knowledge of PWConstraints and ENP to determine the type and position of the inserted articulations. This is done according to the following two principles: the size of the melodic movement or the beaming level of the notes.

The result, which is shown in Figure 5, can be accomplished with the following script containing two scripting rules:

\begin{verbatim}
(* ?1 ?2 (?if (when (and
 (> (abs (- (m ?2) (m ?1))) 12) ;2
 (not (prev-rest? ?2))) ;3
 (add-expression 'slur ?1 ?2)))) ;4

(* ?1 (?if (when (= (beaming-level ?1) 2) ;5
 (add-expression 'accent-grave ?1)))) ;6
\end{verbatim}

The first row of the script shows a wild card and two named variables (?1 and ?2). In the next two rows we check if the size of the melodic movement is greater than an octave (2) and also that there is no rest between the two notes (3). If both conditions are met we insert a slur expression (4). The latter part of the script is quite straightforward.

If the beaming of the note has two levels (5) a grave accent is added (6).

Figure 3: Trumpet fingerings inserted into the score by the script.

Figure 4: Fingering expressions are dynamic, so when the musical material is transposed the fingerings are updated accordingly.

Figure 5: The slurs and grave accents inserted by a script. The type and placement of the expressions has been determined by the script according to the duration of the notes or the size of the melodic movement.

5 Conclusions

This paper demonstrated how ENP can be scripted using the PWConstraints pattern-matching language. The power of the pattern-matching language combined with the intelligent expression handling of ENP provides a powerful tool for different kinds of compositional and analytical applications. The scripting scheme presented here is useful when used simply as a speedy entry to attach, for example, articulations or articulation patterns. It can also be used to perform more sophisticated operations including music analysis and visualization.

This research has been partly conducted within the project 'CLARA - Control Language for Audio Related Applications' financed by the Academy of Finland.

References


