A Tree Based Language for Music Score Description.

Dominique Fober, Y Orlarey, S Letz, R. Michon

▶ **To cite this version:**

Dominique Fober, Y Orlarey, S Letz, R. Michon. A Tree Based Language for Music Score Description.. International Symposium on Computer Music Multidisciplinary Research, Oct 2019, Marseille, France. hal-02368958

HAL Id: hal-02368958
https://hal.science/hal-02368958

Submitted on 21 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Tree Based Language for Music Score Description.

D. Fober[1], Y. Orlarey[1], S. Letz[1], and R. Michon[1]

Grame CNCM Lyon - France
{fober,orlarey,letz,michon}@grame.fr

Abstract. The presented work is part of the INScore project, an environment for the design of augmented interactive music scores, oriented towards unconventional uses of music notation and representation, including real-time symbolic notation capabilities. This environment is fully controllable using Open Sound Control [OSC] messages. INScore scripting language is an extended textual version of OSC messages that allows you to design scores in a modular and incremental way. This article presents a major revision of this language, based on the description and manipulation of trees.

Keywords: Music notation · Programming language · INScore.

1 Introduction

There is a large number of musical score description languages (Lilypond [11], Guido [9], MuseData [8], MEI [12], MusicXML [7] etc.) that are all turned towards common western music notation. The extension of some of these languages has been considered, in order to add *programmability* e.g. operations to compose musical scores in Guido [5], or the Scheme language in Lilypond. There are also programming languages dedicated to common music notation, like CMN [13] or ENP [10] that are actually Lisp dialects.

The approach proposed by INScore [4] is different: symbolic music notation is supported (via the Guido language and rendering engine [2,9]), but it constitutes one of the means of music representation among others, without being privileged. Purely graphic scores can be designed. All the elements of a score (including purely graphical elements) have a temporal dimension (date, duration and tempo) and can be manipulated both in the graphic and time space. The notion of time is both event-driven and continuous [6], which makes it possible to design interactive and dynamic scores. Figure 1 presents an example of a score realised using INScore. It includes symbolic notation, pictures, a video, and cursors (the video is one of them) which positions are synchronised by the performer gestures.

INScore has been initially designed to be driven by OSC messages [14]. OSC is basically a communication protocol. A textual version of the OSC messages constitutes the INScore storage format, which has been extended to a scripting language, [3] allowing greater flexibility in music scores design. These extensions (variables, extended addresses, Javascript section, etc.) have nevertheless suffered from a rigidity inherent to an ad hoc and incremental design. For example, the parser makes a clear distinction between OSC addresses and associated data, which prevents the use of variables in OSC addresses. Thus, a major revision of this language became necessary. It is based on the

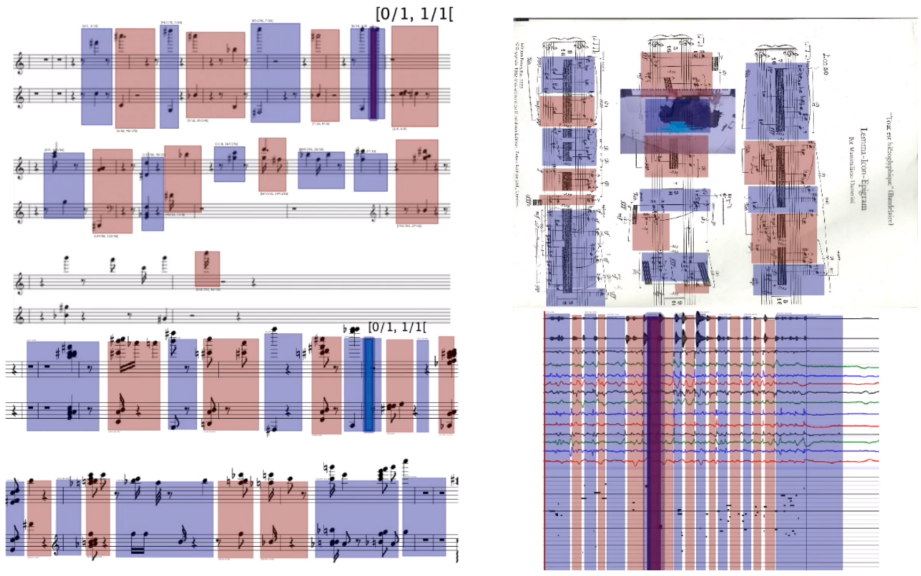

Fig. 1. A score realised using INScore, used as part of a sensor-based environment for the processing of complex music called *GesTCom* (*Gesture Cutting through Textual Complexity*) [1].

manipulation of a regular tree structure that is also homogeneous to the INScore model. Figure 2 gives an example of such model hierarchy, that can be described in the current scripting language (i.e. OSC) by listing all branches from the root.

After some definitions, we will present the basic operations on trees and the corresponding grammar.Then we introduce mathematical operations on trees, the concepts of *variables* and *nodes in intention* and we'll present how this language is turned into OSC messages. The final section gives an example of the new language before concluding.

2 Definitions

A tree t consists of a value v (of some data type) and the (possibly empty) list of its subtrees.

$$t : v \times [t_1, ..., t_k]$$

A tree with an empty list of subtrees $t : v \times [\,]$ is called a leaf.
A value is among literal (i.e., text, number) or special values of the following types:

- forest ($\varnothing$): denotes a tree including only subtrees,
- mathematical operators: indicates a mathematical operation between subtrees,
- variable: denotes a tree whose value refers to another tree,
- expand: indicates a tree to be expanded,
- slash (/): used for conversion to OSC

Use and evaluation of these values is detailed in the next sections.

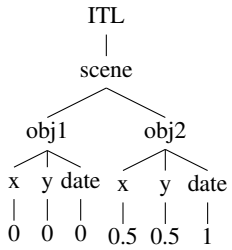

Fig. 2. A score sample including 2 objects $obj1$ and $obj2$, having x, y, and $date$ attributes. Time properties (date, duration) are notably used to represent the time relationship between objects.

3 Operations on Trees

We define two abstract operations on trees: sequencing and paralleling. These operations have no musical semantics, neither from a temporal nor from a graphic point of view. They are defined as methods for algorithmic construction of OSC messages and operate on the topological organisation of the trees.

3.1 Putting Trees in Sequence

Putting two trees t and t' in sequence adds t' as child of all the leaves of t. We will note $|$ the sequencing operation. Let 2 trees $t : v \times [t_1, ..., t_k]$ and t'. Then:

$$v \times [t_1, ..., t_k] |\, t' \rightarrow v \times [t_1 |\, t', \ldots, t_k |\, t']$$

with:

$$\begin{cases} v \times [\,] |\, t' \rightarrow v \times [t'] \\ v \times [\,] |\, \varnothing \times [t_1, ..., t_k] \rightarrow v \times [t_1, ..., t_k] \end{cases}$$

The right arrow ($\rightarrow$) indicates the result of an expression evaluation.

3.2 Putting Trees in Parallel

Putting two trees t and t' in parallel consists in putting them in a forest. We will note $\|$ the parallelisation operation. Let 2 trees t and t':

$$t \parallel\ t' \rightarrow \varnothing \times [t, t']$$

The result is a tree which value $\varnothing$ denotes a forest.
Parallelisation applied to a *forest* preserves the subtrees order:

$$\begin{cases} \varnothing \times [t_1, ..., t_k] \ \parallel\ t' \rightarrow \varnothing \times [t_1, \ldots, t_k, t'] \\ t' \parallel\ \varnothing \times [t_1, ..., t_k] \ \rightarrow \varnothing \times [t', t_1, \ldots, t_k] \end{cases}$$

4 Grammar

A tree is syntactically defined in BNF as follows:

```
tree := value       → t : value [ ]
    | tree tree     → t : tree | tree
    | / tree        → t : '/' | tree
    | tree , tree   → t : tree || tree
    | ( tree )      → t : tree
    ;
```

The right arrow ($\rightarrow$) indicates the tree built for each syntactical construct. The tree whose value is *slash* (*/*) plays a special role in the tree conversion to OSC messages. This role is described in section 6.

5 Values and Evaluation

This section explains how the trees carrying the special *mathematical operators*, *variables* and *expand* special values are evaluated.

5.1 Mathematical Operators

Mathematical operations on trees are seen as operations on their values that preserve the subtrees. These operations include arithmetic and logical operations, trigonometric and hyperbolic functions, exponential and logarithmic functions, power, square root, etc.

We will designate these operations by op. Then for 2 trees $t : v \times [t_1, ..., t_k]$ and $t' : v' \times [t'_1, ..., t'_k]$:

$$\mathtt{op} \times [t, t'] \rightarrow (\mathtt{op}\ v\ v') \times [t_1, \ldots, t_k, t'_1, \ldots, t'_k]$$

5.2 Variables

The special value type *variable* denotes a tree whose value refers to another tree. Evaluation of a variable tree consists in expanding the referred tree at the variable position. Let's define a variable var and a variable tree t' as follows:

$$\begin{cases} var = v \times [t_1, ..., t_k] \\ t' : \$var \times [t'_1, ..., t'_k] \end{cases}$$

then

$$t'_{\{var\}} \rightarrow v \times [t_1, ..., t_k] \mid \varnothing \times [t'_1, ..., t'_k]$$

t'$_{\{var\}}$ denotes the tree t' with an environment containing a definition of the variable var.

Example :

```
x = x 0;
y = y 0;
/A/B $x, $y;   ⇒ /A/B (x 0), (y 1);
```

Local Environnements Each tree is evaluated in an environment containing the list of all the variables of its parent. However, a variable can be evaluated in a local environment, which is defined inside braces:

$$\begin{cases} var = t \\ \$var\{a = t1, b = t2, \dots\} \rightarrow t_{\{a,b,\dots\}} \end{cases}$$

5.3 Expand Value

An *expand value* is a special value that is expanded to a forest. It can also be seen as a *loop* control structure. The syntactic form is as follows:

$id[n \dots m]$ where n and m are integers
$id[ab \dots xy]$ where a, b, x, y are letters.

We will note ε the expansion operation:

$$\begin{cases} \varepsilon(id[n \dots m]) \rightarrow \varnothing\ [id_n, id_{n+1}, \dots, id_m] \\ \varepsilon(id[ab \dots xy]) \rightarrow \varnothing\ [id_{ab}, id_{ac}, \dots, id_{ay}, \\ \qquad\qquad\qquad\qquad \dots, \\ \qquad\qquad\qquad\qquad id_{xb}, id_{xc}, \dots, id_{xy}] \end{cases}$$

where each id_n is a tree $v \times [\,]$ whose value v is the concatenation of the base value id and of the current index n.

Special Forms An *expand value* can also take the following special forms:

$id[i : n \dots m]$ where i is an identifier
$id[i : j : ab \dots xy]$ where i, j are identifiers.

The identifiers denote variables that are instantiated in the environment by the expansion operation, with the current index value. For example:

$$\varepsilon(id[i : n \dots m]) \quad \rightarrow \varnothing\ [id_{n\{i=0\}}, id_{n+1\{i=1\}}, \dots, id_{m\{i=m-n\}}]$$

6 Conversion to OSC

An OSC message is made of an OSC address (similar to an Unix path) followed by a list of data (which can possibly be empty) The *slash* special value of a tree is used to discriminate the OSC address and the data. In order to do so, we type the values and we define @ as the type of a value part of an OSC address.We'll note $type(v)$ to refer to the type of the value v.

We'll note t^a a tree t which value is of type @ . Then we define a @ operation that transforms a tree in to a *typed* tree:

$$@(v \times [t_1, ..., t_k]) \rightarrow \begin{cases} \varnothing \times [t_1^a, ..., t_k^a], v = / \\ v\ \times [t_1, ..., t_k], v \neq / \end{cases}$$

The conversion of a tree t into OSC messages transforms the typed tree $@(t)$ into a forest of OSC addresses followed by data:

$$\texttt{OSC}(v \times [t_1, ..., t_k]) \rightarrow \begin{cases} \varnothing \times [v \times \texttt{OSC}(t_1), \dots, v \times \texttt{OSC}(t_k)], & type(v) = @ \\ v\ \times [\texttt{OSC}(t_1), \dots, \texttt{OSC}(t_k)], & type(v) \neq @ \end{cases}$$

7 Example

The script below presents an example of the new version of the INScore scripting language. Variables are indicated in blue. Local variables are declared in red.

```
# variables declaration
pi    = 3.141592653589793;

# '$step' makes use of 'count' a local variable
step  = / ( * 2, $pi), $count;

# '$i' is defined by the expansion of the address 'n_[i:1...9]'
x = math.sin ( * $step, $i );
y = math.cos ( * $step, $i );

# the following variables select part of guido
# music notation code to build a short score
dyn = (? (% $i, 3), '\i<"p">', '\i<"ff">');
note = (+ $dyn, "␣", (?  (% $i, 2), "e2", "g1/8"));

# this is a classical OSC message that simply clears the scene
/ITL/scene/* del;

# this is the main variable. It will be expanded to create
# a series of small scores. The variables are computed
# using locally defined variables.
notes = (/ITL/scene/$addr
        (set gmn (+ "[", $note, "]")),
        (scale 0.7),
        (x * $x, $radius),
        (y * $y, $radius));

# finally '$notes' is used  with addr, count and radius as local
# variables, which could be viewed as a function call.
$notes{addr=n_[i:1...9], count=9, radius=0.7};
```

Evaluation of this script produces OSC messages fully compatible with the previous version of the language, and which are schematically presented below.

```
/ITL/scene/n_1 set gmn '[ ı<"ff"> g1/8]';
/ITL/scene/n_1 scale 0.7;
/ITL/scene/n_1 x 0.0;
/ITL/scene/n_1 y 0.7;
...
/ITL/scene/n_9 set gmn '[ ı<"ff"> c2]';
/ITL/scene/n_9 scale 0.7;
/ITL/scene/n_9 x -0.411452;
/ITL/scene/n_9 y 0.56631;
```

In practice, this example expresses the score illustrated in Figure 3 in just a few lines.

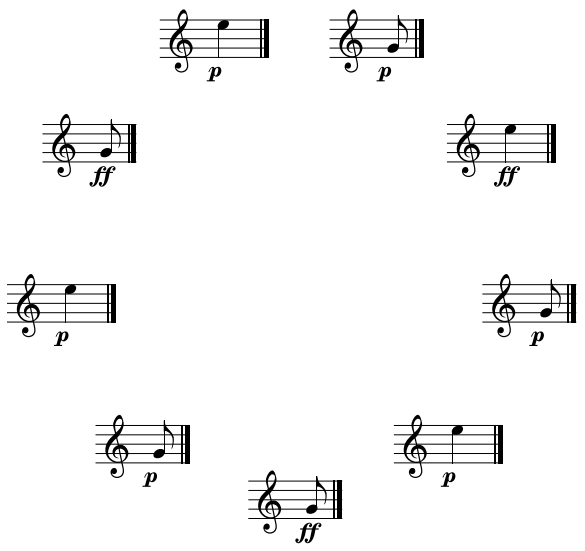

Fig. 3. INScore scene corresponding to the sample script given in section 7.

8 Conclusions

From two elementary operations on trees - sequencing and parallelisation - we have homogeneously introduced the notions of variables and of mathematical and logical operations on trees. The resulting language is much more expressive and more flexible than the previous version of the INScore scripting language. It supports parallelisation of the arguments of a message, variables to describe addresses, series of addresses expressed in a concise manner, use of local variables allowing reusing scripts or parts of scripts in different contexts.

References

1. Antoniadis, P.: Embodied navigation of complex piano notation : rethinking musical interaction from a performer's perspective. Theses, Université de Strasbourg (Jun 2018)
2. Daudin, C., Fober, D., Letz, S., Orlarey, Y.: The Guido Engine – A toolbox for music scores rendering. In: LAC (ed.) Proceedings of Linux Audio Conference 2009. pp. 105–111 (2009)
3. Fober, D., Letz, S., Orlarey, Y., Bevilacqua, F.: Programming Interactive Music Scores with INScore. In: Proceedings of the Sound and Music Computing conference – SMC'13. pp. 185–190 (2013)
4. Fober, D., Orlarey, Y., Letz, S.: INScore - An Environment for the Design of Live Music Scores. In: Proceedings of the Linux Audio Conference – LAC 2012. pp. 47–54 (2012)

5. Fober, D., Orlarey, Y., Letz, S.: Scores Level Composition Based on the Guido Music Notation. In: ICMA (ed.) Proceedings of the International Computer Music Conference. pp. 383–386 (2012)
6. Fober, D., Orlarey, Y., Letz, S.: INScore Time Model. In: Proceedings of the International Computer Music Conference. pp. 64–68 (2017)
7. Good, M.: MusicXML for Notation and Analysis. In: Hewlett, W.B., Selfridge-Field, E. (eds.) The Virtual Score. pp. 113–124. MIT Press (2001)
8. Hewlett, W.B.: MuseData: Multipurpose Representation. In: E., S.F. (ed.) Beyond MIDI, The handbook of Musical Codes. pp. 402–447. MIT Press (1997)
9. Hoos, H., Hamel, K., Renz, K., Kilian, J.: The GUIDO Music Notation Format - a Novel Approach for Adequately Representing Score-level Music. In: Proceedings of the International Computer Music Conference. pp. 451–454. ICMA (1998)
10. Kuuskankare, M., Laurson, M.: Expressive Notation Package. Computer Music Journal **30**(4), 67–79 (2006)
11. Nienhuys, H.W., Nieuwenhuizen, J.: LilyPond, a system for automated music engraving. In: Proceedings of the XIV Colloquium on Musical Informatics (2003)
12. Roland, P.: The Music Encoding Initiative (MEI). In: MAX2002. Proceedings of the First International Conference on Musical Application using XML. pp. 55–59 (2002)
13. Schottstaedt, B.: Common Music Notation., chap. 16. MIT Press (1997)
14. Wright, M.: Open Sound Control 1.0 Specification (2002), `http://opensoundcontrol.org/spec-1_0`