CSOUNDXML: A META-LANGUAGE IN XML FOR SOUND SYNTHESIS

Pedro Kröger
Federal University at Bahia, Brazil

ABSTRACT

The software sound synthesis is closely related to the Music N programs started with Music I in 1957. Although Music N has many advantages such as unit generators and a flexible score language, it presents a few problems like limitations on instrument reuse, inflexibility of use of parameters, lack of a built-in graphical interface, and usually only one paradigm for scores. Some solutions concentrate in new from-scratch Music N implementations, while others focus in building user tools like pre-processors and graphical utilities. Nevertheless, new implementations in general focus in specific groups of problems leaving others unsolved. The user tools solve only one problem with no connection with others. In this paper we investigate the problem of creating a meta-language for sound synthesis. This constitutes an elegant solution for the above cited problems, without the need of a yet new acoustic compiler implementation, allowing a tight integration which is difficult to obtain with the present user tools.

1. INTRODUCTION

The history of software sound synthesis is closely connected to the programs written by Max Mathews in the 50's and 60's such as Music IV and Music V. A large number of programs (e.g. Music 4BF, Music 360, Music 11, Csound, Cmusic, Common Lisp Music, only to cite a few) were developed taking Music V as a model. Usually these programs are called "Music N"-type programs. (Although not entirely correct we will call these programs "Music N implementations").

Despite its strengths, such as unit generators, a flexible score language, power and speed, Music N has a few problems that can be divided in: instrument design, score manipulation, and integration between instrument and score.

Regarding instrument design, the first problem is instrument reuse. Only a few Music V based programs have named instruments and tables instead of numbered. Only very few implementations have great communication flexibility and data exchange between instruments, and none allow the definition of context dependent sound output. The second problem is the well-known ordered list; all parameters are defined as an ordered list. This makes utilization more difficult for the user (it's hard to remember the order and function of all parameters, specially when an unit generator uses a dozen of them) and programs to extract instrument data. The third problem is the lack of scalability of the tools developed to describe instruments graphically. They have to have a deep understanding of the language syntax, not infrequently implementing a (yet another) full parser. Some programs such as Supercollider and Csound have specific opcodes for graphical widgets. Unfortunately, this solution results in having graphical elements in the same level of the sound synthesis. This is one of the reasons this solution is not scalable; if the graphical representation has to be changed, the instrument core has to be modified.

Score manipulation represents an entirely different problem because a composition is described on it. And different composers compose in different ways and need different tools. Some solutions as preprocessors and generic programming languages are useful but limited. At one hand preprocessors usually have only one fixed syntax and paradigm, not being flexible enough to accommodate the composer's style. On the other hand, when using a generic programming language the composer has all the flexibility not found with preprocessors, but it is necessary to learn a complete programming language before composing, which is not reasonable.

The last problem is the lack of integration between the orchestra and the score, and specially the lack of integration between solutions for the score (i.e. preprocessors) and the orchestra. Tools for score processing usually define musical representation in a higher level then the flat note list. However this breaks the communication between the "pre-score"—the file to be processed and converted in the score—and the orchestra (fig. 1). Communication between the preprocessor file and the orchestra, or better yet, between the "pre-score" and a "pre-orchestra" would be highly needed (fig. 2).

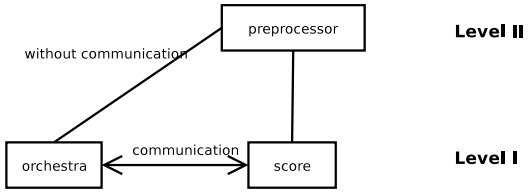

Figure 1. Relationship between score, orchestra, and preprocessor

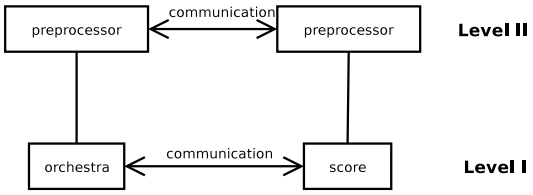

Figure 2. Relationship between score, orchestra, and preprocessor

In this paper we investigate the problem of creating a meta-language for sound synthesis. This constitutes an elegant solution for the above cited problems, without the need of a yet new acoustic compiler implementation, allowing a tight integration which is difficult to obtain with the present user tools. The details of the use of a score language with csoundXML is the subject for another paper. A preliminar work and introduction can be found at [5].

2. CSOUNDXML

CsoundXML is a meta-language in XML for sound synthesis developed by the author of this paper. A meta-language is usually used to define or describe another language. It describes the Csound orchestra language with a few added features.

The ideal and highly desirable goal would be a unique meta-language for sound synthesis capable of describing all synthesis algorithms. However, this language is very difficult to develop, if not impossible. The original goal of the MPEG-4 Structured Audio [4] was to function as an intermediate format between any synthesis program,

> but it rapidly became clear that this idea is untenable. The different software synthesizers—Csound, SAOL, SuperCollider, Nyquist, and the commercial graphical ones—all have different underlying conceptions of events, signals, opcodes, and functions that makes it impossible to have a single format that captures anything but the very simplest aspects of behavior [9].

Since a universal language for synthesis is not viable, one solution is to create a standard and wait for its adoption [9]. Another solution is to define a generic and extensible language with a few target languages [2]. CsoundXML is an example of the latter while SAOL [3] is an example of the former.

2.1. Advantages

2.1.1. Languages conversion

XML has been used with success in the creation of meta-languages for conversion between different languages [6, 1, 8, 7]. CsoundXML works as a starting point in the sense that instruments written in it can be converted to different synthesis languages such as Csound, Cmix, and so on. Although CsoundXML is not a "universal language", it is compatible with the Csound orchestra format, and consequently, other programs in the Music N family.

2.1.2. Databases

The existence of a large collection of Csound instruments is one of the main sources of learning. Now that the number of these instruments is more then 2000, is necessary the creation of a more formal database. Having these instruments converted to CsoundXML allows the use of meta-information tags such as author, description, localization, and so on. This information can be easily extracted and manipulated.

2.1.3. Pretty-print

Pretty-Print is more than an eye candy feature. The possibility to print Csound code with typographical quality is a necessity of book and article authors. Having an instrument written in XML, the conversion to Csound can be done in different ways. One simple example is the use of comments. One can choose if they will or will not be printed and *how* they will be printed; if above, below, or sideways of an expression.

2.1.4. Graphical tools

Because CsoundXML is a formal and structured language it is possible to describe instruments graphically automatically. There are two basic problems:

1. design decisions to define how elements will be drawn. Sound generators such as oscillators are easy to represent while opcodes that convert values and flow control are hard to represent graphically.

2. algorithms to distribute the synthesis elements in the screen avoiding collision. Having the previous item solved is necessary to have "smart" algorithms to allow different kinds of visualization and complexity.

2.1.5. Integration

XML allows the integration of different paradigms and visualizations modes. For example, a system can be built on top of CsoundXML to display instruments as flowcharts or as a parameter editor, or to emulate a Music N-style syntax (fig. 3).

2.2. CsoundXML syntax

This section will show a few syntatic elements to give an idea of how CsoundXML looks like. In addition it also supports flow control, different types of output, functions, expressions, and meta information.

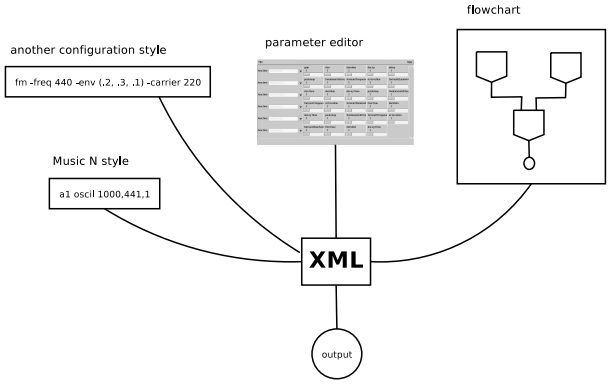

Figure 3. XML helps integration

Example 2.1 A typical Csound opcode

```
afoo  oscil  10000, 440, 1  ; some comment here
```

2.2.1. Opcodes

The heart of Csound instruments are the *unit generators*, implemented as opcodes. The ex. 2.1 shows a typical opcode, `oscil`, where `afoo` is an a-variable that will hold the opcode output. `10000` is the amplitude, `440` is the frequency, and `1` is the function number with a wave shape. The text after the semi-colon is a comment that will be disregarded by Csound.

In CsoundXML the opcodes are defined by the `opcode` element and its parameters by the `par` element. The opcode and parameter name is defined by the `name` attribute. The `id` attribute defines a unique name for each element. It can also be used to provide connection between elements, like variables (see 2.2.2). The ex. 2.2 shows the code of ex. 2.1 in CsoundXML.

Information about the opcodes (e.g. how many and which parameters) and parameters (e.g. the possible values) is defined in an XML library for Csound, CXL [1] , also developed by the author of this paper. A kind of cross-reference between CsoundXML and CXL is achieved using the `name` attribute.

The `type` attribute indicates the variable type (e.g. `k`, `i`, or `a`). Variables can have any name, CsoundXML makes sure the variable will start with the right letter when converting to Csound. This is a valuable feature for automatic conversion between variables.

[1] This is the subject for another paper, yet to be published.

Example 2.2 A Csound instrument in XML

```
<opcode name="oscil" id="foo" type="a">
  <out id="foo_out"/>
  <par name="amplitude">
    <number>10000</number>
  </par>
  <par name="frequency">
    <number>440</number>
  </par>
  <par name="function">
    <number>1</number>
  </par>
  <comment>some comment here</comment>
</opcode>
```

Example 2.3 Parameter definition

```
<defpar id="gain" type="i">
  <default>20</default>
</defpar>
```

Each parameter may have three kinds if input, a simple numeric value (e.g. “1”), a variable (e.g. “iamp”), or an expression (e.g. “iamp+1/idur”). If the input is a numeric value, the `number` element is used (line 4 of ex. 2.2). If the input is an expression, the `expr` element is used. Finally, if the input is a variable, the par element will be empty and the variable will be defined by the `vvalue` attribute. Vvalue stands for “variable value”. The value of `vvalue` must be the same of the `id` of the variable defined by `defpar` (see section 2.2.2).

One may be bothered by the verbosity of XML documents. Our original example (ex. 2.1) has only one line while the CsoundXML version (ex. 2.2) has 13! Nevertheless, XML verbosity is a feature and not a bug. It permits, among other things, more complete searches. Still in the example 2.2, a program for drawing functions could quickly and easily see how many and which functions an instrument is using looking for the “function” attribute in the `<par>` tag. It is important to keep in mind that regardless the first impression, having structured information in XML make life easier for the programmer/user. All the process of reading the XML file, determinating the structure and propriety of data, dividing the data in pieces to send to other components is done by the XML parser. Since there are many parsers available, both commercially and freely, the developer does not have to make one from scratch.

2.2.2. Parameters and variables

In CsoundXML variables are defined with the `defpar` element. It also has the *id* and `type` attributes (ex. 2.3).

A more complex example is shown in ex. 2.4 where the `gain` parameter is defined. The `description` element contains a brief description, the `default` element has a valid default value for the parameter, and the `range` element defines the numerical range. A graphical tool could extract the information in `range` to automatically create sliders for each parameter.

If the `auto` attribute is equal to “yes” its value will be automatically assigned to a pfield. That is, the CsoundXML code `<defpar id="notes" auto="yes"/>` is equivalent to the code `inote = p4` in Csound. The difference is that the exact pfield is not determined by the instrument designer but by the program implementing CsoundXML. This is a more flexible solution than the conventional use since the parameters in the score will be called by the variable names, not by pfields.

Example 2.4 A parameter with a default value and range

```
<defpar id="gain">
  <description>
    gain factor , usually between 0 − 1
  </description>
  <default>1</default>
  <range steps="float">
    <from>0</from>
    <to>1</to>
  </range>
</defpar>
```

2.3. Parameter editor

The figure 4 shows an overview of the creation of a parameter editor. After DTD validation (in order to check the correctness of the XML file) the needed data is extracted from the instrument. The program looks for elements with the `auto="yes"` attribute to create sliders for each parameter and selection boxes for each function. Since the functions are defined in a separate file, the program reads all functions in that file and shows all of them in the selection box.

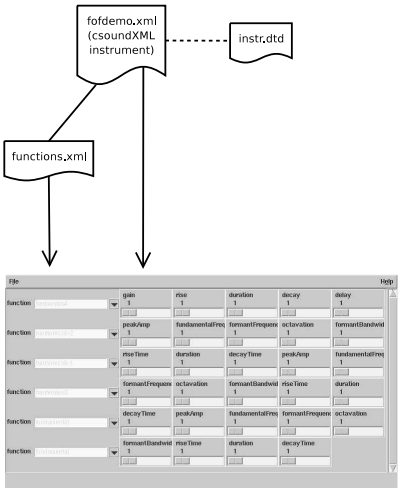

Figure 4. Parameter editor—GUI creation

The greatest advantage of this approach is that the GUI is created from a regular CsoundXML, that is, no specific graphical information has to be coded in the instrument. The GUI is generated automatically.

Data can easily be obtained from the instrument by using Xpath queries. The Xpath code for extracting all functions is `//par[@name='function']`, for example. This kind of data can be very useful for creating instrument debuggers, for knowing the most used opcodes in a collection of instruments, for controlling functions, and so on.

3. CONCLUSIONS AND FUTURE WORK

The creation of a meta-language for sound synthesis solves some of the problems raised in section 1.

Instrument reuse is made possible by a high-level description, named instruments, flexible signal input and output, and mainly the possibility to be able to define multiple outputs depending on context.

The use of a structured syntax (such as XML's) allows bypassing the limitations of Music N's flat lists. It is possible to extract informations from the instrument easily (section 2.3).

Unlike other solutions that add graphical commands in the instrument, the meta-language XML structure allows the automatic creation of graphical instruments, without extra opcodes (section 2.3).

Finally, the problem of lack of integration between the solutions for the score (preprocessors) and the orchestra is solved with a description of both in a higher level and the use of parameter automation and context. The proposed solution allows the creation of an integrated system that can be accessed with different interfaces.

The solutions presented in this work can be applied in different context. They can be implemented as tools to expand programs already existent like Csound, constitute the basis for a new compositional system, be incorporated to existent sound synthesis programs, be extended to use other synthesis languages than Csound as basis.

4. REFERENCES

[1] Yannis Chicha, Florence Defaix, and Stephen M. Watt. *A C++ to XML translator*. The FRISCO consortium, 1999.

[2] Michael Gogins. Re: SML (synthesis modelling language), Jan 2000.

[3] ISO/IEC. *Information technology—coding of audio-visual objects*, 1999.

[4] Rob Koenen. Overview of the mpeg standard. Technical report, ISO/IEC JTC1/SC29/WG11, 1999.

[5] Pedro Kröger. *Desenvolvendo uma meta-linguagem para síntese sonora [Developing a meta-language for sound synthesis]*. PhD thesis, Federal University at Bahia, Brazil, 2004.

[6] Manuel Lemos. MetaL: XML based meta-programming engine developed with php. In *PHP Conference 2001*, Frankfurt, November 2001. PHP-Center and Software & Support Verlag.

[7] Alagappan Meyyappan. GUI development using XML, 2000.

[8] Soumen Sarkar and Craig Cleaveland. Code generation using xml based document transformation, 2001. Available at `http://www.theserverside.com/resources/articles/XMLCodeGen/xmltransform.pdf`.

[9] Eric Scheirer. Re: SML (synthesis modelling language), 2000.