CsoundXML: a meta-language in XML for sound synthesis

Pedro Kröger
Federal University at Bahia, Brazil
kroeger@pedrokroeger.net

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 (see section 3.2.4). 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).

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 preliminary work and introduction can be found at (Kröger 2004).

2 A quick introduction to XML

XML stands for Extensible Markup Language and it is a markup language for structured documents. It is a standard defined by the World Wide Web Consortium. Among the advantages of XML are the creation of domain-specific markup languages, self-describing data, interchange of data among applications, and structured and integrated data (Harold 1999, pp. 6–8).

XML applications have been used in different areas such as mathematics (mathML), graphics (VML and SVG), and computer programming (OOPML). Although the number of XML applications for music has grown considerably, with solutions for musical description and notation—MDL (Roland 2001), musicXML (Good 2001), and EMNML (Mosterd 1999)—there are only a few solutions for sound synthesis and they are in an early stage of development—Javasynth (Makela 2003) and FlowML (Schiettecatte 2000).

With XML is possible to create a meta-language for sound synthesis that can work as an interchange format between already existing languages. Nevertheless it is important that this meta-language be based in concrete principles and already existent programs, in order to avoid an abstract format with no practical use.

2.1 XML syntax

Like HTML, XML uses tags to mark specific parts of text. But while HTML was designed to display data, XML was designed to describe data. For example, in HTML the tag <b> marks a text to appear as bold (ex. 2.1). The example 2.2 shows the result as probably seen in a web browser.

HTML has a fixed set of tags that have to be used accordingly. For instance, the use of tag like <my-tag> would bring a error message or be ignored. XML, on the other hand, does not have any tag defined, all tags have to be defined by the user. The previous example can be rewritten to use a tag to represent the meaning of the text (ex. 2.3). The tag emphasis does not exist in HTML, but it can be defined in XML to indicate some text to have some sort of emphasis, be it typesetted with italics, bold, or underline. This is one of the advantages of describing a document with structural instead of formatting tags. If any change in formatting is necessary, the way how a particular structure is displayed can be changed. In our example, the emphasis tag can be configured to generate an underlined text instead of italic, for example.

The example 2.4 shows a typical use of XML. The tag <books> has information about books. The specific information of each book is inside the tag <book>. This tag has information of author (<author> tag) and title (<title> tag).

This example (2.4) shows the basic characteristics of any XML document:

- information is contained inside tags, or elements. The tag <tag> starts a block while the tag with the same name starting with a slash (/) closes the block, </tag>.
- tags can be nested. In ex. 2.4 we have blocks that define books (<book> inside the tag <books>). And the

---

Example 2.1 Example of HTML markup

Bold face can be used to give &lt;b&gt;emphasis&lt;/b&gt.

---

Example 2.2 Result of ex. 2.1

Bold face can be used to give emphasis.

---

Example 2.3 Example of XML markup

Bold face can be used to give &lt;emphasis&gt;emphasis&lt;/emphasis&gt.;
### Example 2.4 A typical XML file

```xml
<?xml version="1.0"?>
<books>
  <book id="1001">
    <author>Arnold Schoenberg</author>
    <title>Harmonielehre</title>
  </book>
  <book id="1002">
    <author>Jean-Philippe Rameau</author>
    <title>Traité de l'harmonie</title>
  </book>
</books>
```

The XML declaration in the first line identifies the document as XML. Elements can have information in attributes, as seen in lines 3 and 7 of ex. 2.4. Attributes have a name and value, as `name="value"`. The section 2.2 will explore more about the subject.

XML documents must have only one element in the root, the highest level. The root tag in example 2.4 is `<books>`.

### Example 2.5 A chord codified in XML

```xml
<chord>
  <note name="c" octave="1" />
  <note name="d" octave="1" />
  <note name="e" octave="1" />
</chord>
```

There is no agreement if data have to be stored in attributes or elements. Some argue that the DTD has more features to read and process attributes and therefore attributes should be used. On the other hand, some argue that elements are easier to edit and parse (Marchal 2000, p. 111). We agree with Harold that "the data itself should be stored in elements. Information about data (meta-data) should be stored in attributes" (Harold 1999, p. 101). The only exception is when meta-data have to be structured.

### 3 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 (Koenen 1999) 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 (Scheirer 2000).

Since a universal language for synthesis is not viable, one solution is to create a standard and wait for its adoption (Scheirer 2000). Another solution is to define a generic and extensible language with a few target languages (Gogins 2000). CsoundXML is an example of the latter while SAOL (ISO/IEC 1999) is an example of the former.

### 3.1 Advantages

#### 3.1.1 Languages conversion

XML has been used with success in the creation of meta-languages for conversion between different languages (Lemos...
2001; Chicha, Defaix, and Watt 1999; Sarkar and Cleaveland 2001; Meyyappan 2000). 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.

3.1.2 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 (section 3.2.6).

3.1.3 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 Csoud 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.

3.1.4 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.

3.1.5 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).

3.2 CsoundXML syntax

3.2.1 Opcodes

The heart of Csound instruments are the unit generators, implemented as opcodes. The ex. 3.1 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 3.2.2). The ex. 3.2 shows the code of ex. 3.1 in CsoundXML.

Example 3.2 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 3.3 Parameter definition

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

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, 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.

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. 3.2). 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.

One may be bothered by the verbosity of XML documents. Our original example (ex. 3.1) has only one line while the CsoundXML version (ex. 3.2) has 13! Nevertheless, XML verbosity is a feature and not a bug. It permits, among other things, more complete searches. Still in the example 3.2, 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, determining 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.

3.2.2 Parameters and variables

In CsoundXML variables are defined with the defpar element. It also has the id and type attributes (ex. 3.3).

A more complex example is shown in ex. 3.4 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 pfeld. 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 pfeld 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 pfelds.

3.2.3 Flow control

XML works surprisingly well with flow control structures. Unfortunately the Csound orchestra language is arcane and largely influenced by Assembler and Fortran (Pope 1993). One of the main consequences is the lack of structured constructions found in C and Pascal. Csound uses labels to delimit code blocks (highnote, lownote, and playit in ex. 3.5).

However, in CsoundXML it is possible to have fully structured blocks (ex. 3.6), the conversion to Csound automatically

---

2This is the subject for another paper, yet to be published.
Example 3.6 Flow control in CsoundXML

```xml
<if expression="iparam == 1">
  <then>
    <defpar id="ifreq">880</defpar>
  </then>
  <else>
    <defpar id="ifreq">440</defpar>
  </else>
</if>
```

Example 3.7 Different outputs

```xml
<output>
  <outtype name="mono">
    <in id="mono" vvalue="basic_out"/>
  </outtype>
  <outtype name="stereo">
    <in id="left" vvalue="basic_out"/>
    <in id="right" vvalue="basic_out"/>
  </outtype>
</output>
```

3.2.4 Sound output

CsoundXML has the generic element `output` that maps Csound output opcodes (e.g., `out`, `outs`, and `outq`). The composer can define beforehand how the output will be in different scenarios, such as mono or stereo. This is particularly useful for having different versions of the same music, or when a stereo version of a four-channel music is needed, for example. The composer only has to indicate the output type and the program will choose the right one. This is, to our knowledge, a unique concept in sound synthesis language.

The `output` element has the `outtype` element that defines the output type. `output` can have one or more `outtype` elements. The `name` attribute (in `outtype`) defines the output type as mono or stereo. Finally, the `in` element and the `vvalue` attribute codes the input signal to the output (ex. 3.7).

Example 3.8 Declaring a function

```xml
<defunc name="10" id="clarinet">
  <time>0</time>
  <size>1024</size>
  <par name="partialStrength" order="1">1</par>
  <par name="partialStrength" order="3">5</par>
  <par name="partialStrength" order="5">3</par>
</defunc>
```

Example 3.9 Including meta-information

```xml
<author>Russell Pinkston</author>
<description>
  Basic vowel formant generation instrument. This uses 5 fo units to simulate vowel formants. The formant data is taken from the Dodge book, pp. 230–231
</description>
<url>www.utexas.edu/cofa/music/ems/</url>
```

the `par` element to define parameters. The `name` attribute indicates the parameter type and works similarly to the homonymous attribute in `opcode`. Differently of Csound, CsoundXML functions are defined in a separate file. This enhances reuse.

3.2.5 Functions

The `deffunc` element is used to describe functions, a basic Csound element. The `name` attribute indicates the GEN and the `id` attribute defines the function name (ex 3.8). Again, the program chooses when to use `ftgen` of `f`, according to the case and necessity. Similarly to `opcode`, `deffunc` uses

Example 3.10 Simple sum in CsoundXML

```xml
<plus>
  <el>2</el>
  <el>2</el>
</plus>
```
Example 3.11 Expression with a variable

```xml
<expr>
  2 <div>
    <el vvalue="dur"/>
  </div>
  4 <el vvalue="dur"/>
</expr>
```

The CsoundXML parser can quickly determinate how many variables an instrument has, just looking the elements that have the `vvalue` attribute.

3.3 An instrument example

A complete CsoundXML instrument can be seen in ex. 3.12 and its flowchart representation in fig. 4. Note that only one variable (`dur`) is defined and five pfields are automatically designed (using the `auto` attribute). These automation features allow instruments to be described in a more generic way, without depending so much on syntax details of each language, in this case, Csound. The `instr` element defines an instrument.

![Figure 4: A simple CsoundXML instrument](image)

```
<expr>
  2 <div>
    <el vvalue="dur"/>
  </div>
  4 <el vvalue="dur"/>
</expr>
```

3.4 Parameter editor

The figure 5 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 (section 3.2.5), the program reads all functions in that file and shows all of them in the selection box.

The greatest advantage of this approach is that the GUI is created from a regular CsoundXML (for example, ex. 3.12),

![Figure 5: Parameter editor—GUI creation](image)
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. Figure 6 shows statistic data inserted in the
parameter editor. 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 instru-
ments, for controlling functions, and so on.

Figure 6: Statistics incorporated in the parameter editor

4 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 descrip-
tion, 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 by-
passing the limitations of Music N’s flat lists. It is possible to
extract informations from the instrument easily (section 3.4).

Unlike other solutions that add graphical commands in
the instrument, the meta-language XML structure allows the
automatic creation of graphical instruments, without extra op-
codes (section 3.4).

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 so-
lution allows the creation of an integrated system that can be
accessed with different interfaces.

The solutions presented in this work can be applied in dif-
ferent 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.

References

C++ to XML translator. The FRISCO consortium. Available at
http://www.nag.co.uk/Projects/Frisco/reports/cpp2xml.ps (2002-09-12).

Gogins, Michael. 2000, Jan. “Re: SML (Synthesis Modelling Lan-
guage).” saol-dev@media.mit.edu: MPEG-4 Structured Audio
Mailing List.

Good, Michael. 2001. “MusicXML for Notation and Analysis.” In
The virtual score: representation, retrieval, restoration, edited
by Walter B. Hewlett and Eleanor Selfridge-Field, Volume 12
of Computing in Musiciology, 113–124. Massachusetts: The
MIT Press.

Books Worldwide.

ISO/IEC. 1999. Information technology—coding of audio-visual
objects. Available at http://sound.media.mit.edu/


síntese sonora [Developing a meta-language for sound synthe-
sis].” Ph.D. diss., Federal University at Bahia, Brazil.

Lemos, Manuel. 2001, November. “MetaL—XML based meta-
programming engine developed with PHP.” PHP Conference
2001. PHP-Center and Software & Support Verlag, Frankfurt.


Marchal, Benoît. 2000. XML: conceitos e aplicações. São Paulo:

Meyyappan, Alagappan. 2000. GUI development using XML.

sheet music via the internet.” Master’s thesis, University of
South Dakota.

for software sound synthesis.” Computer Music Journal 17

Roland, Perry. 2001. “MDL and musicat: an XML approach to
musical data and meta-data.” In The virtual score: representa-
tion, retrieval, restoration, edited by Walter B. Hewlett and
Eleanor Selfridge-Field, Volume 12 of Computing in Musici-

Sarkar, Soumen, and Craig Cleaveland. 2001. Code genera-
tion using XML based document transformation. Available at
http://www.theserverside.com/resources/
articles/XMLCodeGen/xmltransform%.pdf.

Mailing list 19 Janeiro. saol-dev@media.mit.edu: MPEG-4
Structured Audio Mailing List.

Schiettecatte, Bert. 2000. A format for visual orchestras: flowML.
Available at http://wendy.vub.ac.be/~bschiett/
saol/FlowML.pdf (2003-12-21).