Copy-synth-patch: A Tool for Visual Instrument Design

Mikael Laurson and Vesa Norilo
Centre for Music and Technology, Sibelius Academy, P.O.Box 86, 00251 Helsinki, Finland
laurson@siba.fi, vnorilo@siba.fi

Abstract
This paper presents a new approach to design and implement instrument models within a visual programming environment. Our software synthesizer, called PWGLSynth, has been redesigned and rewritten in order to meet this challenge. We introduce a new concept, called copy-synth-patch, which allows to construct in an efficient and compact way complex nested DSP structures that in turn can be used as building blocks when building more high-level entities for physical instrument models.

1 Introduction
During the past five years our research group has been active in building and controlling instrument models of acoustical instruments (Laurson et al. 2001, Välimäki, Laurson, and Erkut 2003, Välimäki et al. 2004). This activity can be split into three main problems. First, it should be possible to implement instruments in an efficient and modular way. Instrument design is an interactive process where components are tested and refined until satisfactory results are achieved. This phase requires a system that allows fast prototyping and interactive listening. Second, the finalized instruments must be tested with some convincing control material. This phase requires the definition of conceptually important control parameters which allow to mimic various playing styles. Finally, control material must be produced that allows testing in a musical context. Our approach in the latter part of the problem is to use a music notation system, ENP, to generate control data (Kuuskankare and Laurson 2002).

In the following we will focus on some specific problems that one encounters when working with the implementation and parametrization part of the instrument design process. While visual synthesis languages have several attractive features, they have also some inherent problems—such as static representation of patches, lack of control structures, etc.—that can lead to instrument definitions that are hard to understand, modify and maintain. We will show that by introducing new control structures and abstraction mechanisms visual definitions can become as economic and compact as it is the case in some of the state-of-the-art textual synthesis languages.

The synthesis examples found in this paper have been realized using a programming environment called PWGL (Laurson and Kuuskankare 2002) and a software synthesizer called PWGLSynth (Laurson and Kuuskankare 2001, Laurson and Norilo 2003).

PWGL is a visual programming language for computer aided composition and sound synthesis. It provides a direct interface to its base languages Common Lisp and CLOS. The graphics part of the system has been realized with OpenGL. Besides a large library of kernel boxes, the system includes also complex editors, such as a 2D-editor, Score-editor and Chord-editor, which allow to inspect and modify musical structures. PWGLSynth, in turn, is a successor of our previous synthesis engine, PWSynth, and it is situated inside the PWGL environment.

PWGLSynth consists of two main parts: the C-component and the Lisp-component. The C-component is written in C++ and it contains a library of DSP-units, real-time scheduling, sequencer, audio hardware support and some other general purpose tools. Our system uses the PortAudio library (Bencina and Burk 2001) for cross-platform audio input and output. The Lisp-component, in turn, is written in Common Lisp and CLOS and is used to access the DSP-units database provided by the C-component. In PWGL this information is used to build popup-menus, boxes and sliders. These visual components can be used to build patches which in turn are translated into a sequence of C function calls. The C-component builds an internal representation of the patch and starts a scheduling process which results in an audio stream. While the scheduler is running it can receive events from the PWGL system. These events can either be user events (such as changing the state of a slider during playback) or pre-calculated sequencer events (typically coming from a music notation package). The synthesis engine updates the outputs and the inputs of the DSP-modules at each sample, which is needed for applications dealing with physical modeling.

PWGLSynth DSP-boxes support vectored inputs and outputs (mono signals are only a special case where the vector length is equal to 1). This scheme is useful as it allows to construct compound entities which are used often in sound synthesis such as banks, parallel structures and serial structures. As any box can return a vectored output, each item in the vector can be processed separately, if
necessary. PWGLSynth provides a rich set of tools that allow to manipulate vectors. For instance vectors can be summed into one signal, vectors can be split into sub-vectors, etc.

This paper is organized as follows. We start with the three main concepts that allow us to visually define loops, control parameters and initial values. After this we extend our concepts so that they can be used to define parallel and nested structures. We discuss also a larger case study where we implement a guitar model using our new scheme.

2 Loop Constructs, Control Parameters and Initialization

In more complex cases visual synthesis languages tend to result in patches that are crowded and confusing. One way to avoid this problem is to introduce special control structures that allow to mimic similar constructs like loops found in some textual synthesis languages, like SuperCollider (McCartney 1998). In order to meet this challenge we propose a new special box called copy-synth-patch having two required inputs, count and patch. A third, optional, input can be given for a name string. copy-synth-patch duplicates a patch connected to the patch input count times. The output of the box is a vector having the length which is determined by the count input. Besides the copy-synth-patch box, our system also includes two other boxes—synth-plug and copy-patch-index—that allow to parametrize and initialize the duplicated patches.

In the next subsections we introduce the three boxes related to the copy-synth-patch scheme and demonstrate the main characteristics of the system with the help of some simple examples.

2.1 copy-synth-patch

The left part of Figure 1 gives a patch consisting of 2 sine wave oscillators which are summed together. The resulting signal is then fed to the output (represented by the synth-box). We can translate this patch to the copy-synth-patch scheme as follows (see the patch to the right in Figure 1). The sine wave oscillator is stated only once and its output is fed to the patch input of a copy-synth-patch box. The count (see the first input) of this box is equal to 2, thus we duplicate the patch input twice and get an output vector of length 2. This output vector is summed by the accum-vector box which is in turn fed to the output. With this specific example it is perhaps not immediately obvious why the patch to the right is ‘better’ than the one to the left. Our point is more clear, however, if we increase the number of oscillators from 2 to, say, 100. This change would mean in the left hand case that we would have to introduce 100 oscillator boxes and connect them by hand to + boxes in order to sum them. By contrast, the only required change in the patch to the right is to edit the first input of the copy-patch-box to be 100.

2.2 synth-plug

The control of complex instruments requires that the system should be able to deal with a large number of control parameters. For instance in a harpsichord model with three sets of string choirs (Välimäki et al. 2004) the number of parameters can exceed even 1500. It is obvious that within such a large system the parametrization process should be automatic.

In order to distinguish between different duplicated patch instances the copy-synth-patch scheme generates automatically symbolic references to specific user defined entry points. These entry points are defined by connecting a synth-plug box at the leafs of a synthesis patch (Figure 1). The entry-points are used afterwards to control the synthesis process. The symbolic references are pathnames, similar to the one found in OSC (Wright and Freed 1997), such as ‘guitar1/1/freq’ or ‘guitar2/6/lf gain’.

The synth-plug box has two inputs, pathname and initval. In the right hand part of Figure 1 the frequency inputs of the oscillators are connected to two synth-plug boxes. The symbolic pathnames, ‘1/fr’ and ‘2/fr’, have been entered by hand by the user. Exactly the same result can be achieved automatically by the patch to the right. When the copy-synth-patch box loops (2 iterations in our case) the system merges the current loop index with the symbol found in the first input of the synth-plug box (in our case ‘fr’), resulting in the pathnames ‘1/fr’ and ‘2/fr’. The benefits of this automatic scheme is again more obvious if we increase the number of iterations.

2.3 copy-patch-index

In the copy-synth-patch example of Figure 1 all synth-plug inputs are initialized to the same value, 440. It is often, however, useful to be able to initialize each input with a separate value during the copying process. To achieve this the second input of a synth-plug box is connected to a box called copy-patch-index. copy-patch-index can either have no inputs or one input containing a list of initial values. In the first case the copy-patch-index box returns simply the current loop index. In the second case—an example of this
case is shown in next figure—the box returns at each iteration step from the list one initial value at a time. In Figure 2 the box contains the list \((440 \ 660)\), thus the first oscillator gets the value \(440\) and the second one \(660\).

Figure 2. synth-plug initialization using the copy-patch-index box.

### 3 Some Configuration Examples

Several copy-synth-boxes can be used to define more complex constructs than the ones presented in the previous section. Figure 3 gives two examples where the left one uses two parallel copy-synth-boxes to build two separate oscillator banks. In order to be able to distinguish the pathnames generated by the two copy-synth-patch boxes, the user can add an extra input which allows to give a name string that is merged in front of the automatically generated pathname. These strings ('sine1/' and 'sine2/') will result in pathnames like 'sine1/1/fr', 'sine1/2/fr' and 'sine2/1/fr'.

The second example demonstrates a case where a copy-synth-patch box is inserted inside another copy-synth-patch box. These kind of nested patches can be useful to define 'bank inside bank' structures. Each level will add an extra sub-path to the resulting pathnames. Thus the nested example to the right will produce pathnames such as 'bank1/1/fr', 'bank1/2/fr', 'bank2/1/fr', etc.

Figure 3. Two copy-synth-patch box schemes. To the left: a parallel use of copy-synth-patch boxes that creates two sine wave banks. To the right: a nested example where a copy-synth-patch box is connected to the patch input of another copy-synth-patch box.

### 4 An Instrument Model Example

We end with a more realistic case that demonstrates how our system can be used to construct a simplified classical guitar model. We start with a top-level patch that defines the main structure of the instrument (Figure 4). The example consists of an abstraction box, called 'string', which defines one string model (the content of this abstraction is given in the Appendix). A copy-synth-patch box copies the string model 6 times. The box has also a name string, 'guitar1/', in the third input. The vectored output of length 6—one signal for each string—of the copy-synth-patch box is summed to a mono signal by an accum-vector box, which is in turn connected to a synth-box.

Figure 4. A Guitar model consisting of 6 strings.

The example given in Figure 4 can easily be extended to include several guitar model instances. Figure 5 shows an example of three guitar models, 'guitar1/', 'guitar2/' and 'guitar3/'. Although they all share the same basic string model, each instrument instance has its own automatically generated parametric name space. For instance, we can point to the loop filter gain parameter, 'lfgain', of the second string of the third guitar model, 'guitar3/', with the following pathname: 'guitar3/2/lfgain' (the 'lfgain' parameter is given in the string model, see the Appendix).

The system is flexible as all instrument instances share the same string model. For instance, we can add new modules or replace old ones simply by locally editing the string abstraction. We can also easily change the parametrization scheme by adding or removing synth-plug boxes. Any changes in the string model will automatically be included in all string model instances generated by the copy-synth-patch scheme.

Figure 5. An example with three guitar models using the parallel copy-synth-patch approach.

Our final example (Figure 6) gives an alternative way to define three guitar instances using the nested approach:

Figure 6. A Guitar Model using the nested approach.
Figure 6. Nested copy-synth-patch scheme for three guitar
models.

5 Conclusions

We have presented our recent developments dealing with instrument design using a visual synthesis environment. The most important concepts introduced in this paper are loop constructs, parameter definition and initial values. These allow to define visually various DSP structures in an economic way. Although these tools were initially developed for virtual instrument design purposes they can be used also effectively to define other DSP oriented tasks such as reverbs, filter banks, etc.

This research has been conducted within the project “CLARA – Control Language for Audio Related Applications” financed by the Academy of Finland.

References


Appendix (string model abstraction):