Abstract

This paper introduces a new Lisp based software synthesizer called PWSynth. PWSynth can be used like other visual synthesis languages with the help of graphical patches. The main aim with this project is, however, to integrate sound synthesis with a powerful music notation package. This scheme allows to control sound synthesis interactively with a familiar representation of complex musical information. The system has also proven to be useful when designing different control strategies for physical models of acoustical instruments.

1 Introduction

PWSynth is a PatchWork (Laurson and Duthen 1989, Laurson 1996) user-library that aims at a better integration of computer assisted composition and sound synthesis. PWSynth is a part of a project which investigates different control strategies for physical models of acoustical instruments. PatchWork is used to generate control data from an extended score representation system called Expressive Notation Package or ENP (Laurson 1999; Kuuskankare and Laurson 2000). Originally, the actual synthesis was meant to be realized with a synthesis environment outside PatchWork. This approach, however, turned out not to be fruitful. It became obvious that we need better integration of the high-level aspects and the DSP parts of the system.

The rest of this paper is divided in five main sections. First we discuss the low-level interface between Lisp and C and show how the user can add new synthesis modules to the system. In the next section we discuss several PWSynth patch examples in order to demonstrate how PWSynth can be used as a general-purpose synthesis language. After this we briefly describe how synthesis patches can be controlled directly from Lisp. Section 5 shows how ENP and PWSynth are interfaced and how ENP is used to control the synthesis engine. The final section discusses how the system can incorporate complex physical models of acoustical instruments with the help of parameter matrices.

2 Interface between Lisp and C

The starting point in PWSynth is a patch consisting of boxes and connections. This patch is as any other PatchWork patch except that each PWSynth box contains a private C-structure. The C-structures are visible both to the Lisp environment and to a collection of C subroutines which are interfaced to the Lisp system. The Lisp environment is responsible for converting a PWSynth patch to a tree of C-structures. Also it fills and initializes all needed structure slots with appropriate data. Once the patch has been initialized PWSynth calls the main synthesis C routine which in turn starts to evaluate the C-structure tree. PWSynth supports also embedded definitions of C-structures (a C-structure can contain other C-structures to any depth). This feature is of primary importance when designing complex instrument models.

The user can add new modules as shown in the following simple code example:

```c
float PW+ (struct PW_box *bx, float n1, float n2)
{
    bx->box_buffer = n1+n2;
    return bx->box_buffer;
}
```

First, a C function is defined with input arguments (the first argument is always a pointer to a C-structure). Next, in the body part, the result is written in a box buffer. The C function always returns the box buffer. This buffering
scheme is used to ensure that the body part of a module is evaluated only once within each sample calculation pass. Finally, the following Lisp expression defines the respective PW box, adds it to a “Math” submenu and links it to the C function defined above:

\[
\text{(add-PWsynth-module ' + 1 () "Math" :function-def? () :default-values '(0.0 0.0))}
\]

3 PWSynth Patches

PWSynth resembles many other visual synthesis languages where the user works with basic entities such as oscillators, noise generators, filters, delays, envelopes and other modules typically used in software synthesis. Real-time sliders, MIDI devices and mouse coordinates can be used to control the patch. These controllers can be scaled to the desired range either using linear or exponential scaling.

Figure 1 gives a simple PWSynth example containing a ‘sinosc’ module with two inputs: ‘freq’ and ‘amp’. These inputs are connected to two slider modules that can be modified by the user in real-time. The output of the ‘sinosc’ module is connected to the first input of a ‘synth’ box that in turn is responsible in converting the synthesis patch to a C-structure tree. Thus every PWSynth patch must contain at least one ‘synth’ box. The second input gives the current synthesis mode. There are three modes. In our case the mode is ‘RT’ (or real-time) which means that the output is sent directly to the DACs. If the mode is ‘File’ then the system runs in a non-real-time mode and the output is written to an AIFF file. In the ‘List’ mode the ‘synth’ box returns the output as a list of floating-point numbers. The latter case is especially useful when the user wants to inspect the output either as numerical values or plot them in PatchWork in a special ‘points’ view box (see below Figure 2). Figure 1 shows also the ‘Synth RT Status’ window that is always shown when the system runs in real-time. It contains some general synthesis information such as the number of channels, the number of the C-structures used by the current patch, etc.

A slightly more complex patch is given in the next example (Figure 3). It contains besides several ‘sinosc’, ‘smooth’, ‘+’, and ‘*’ modules also two real-time modules – ‘mousex’ and ‘mousey’ - that allow to control the synthesis according to the current mouse position within the main window.

Figure 4, in turn, gives a fairly complex patch where there are two banks of resonators (band-pass filters). The
interesting point here is to note that both banks - see the boxes with the name ‘resonbank’ – accept Lisp lists as the third input. In our case the list consists of three sublists – amplitudes, frequencies and bandwidths – that define the behaviour of the resonators. This scheme is very powerful as it allows the user to pass PatchWork data to the synthesizer in a flexible and dynamic way. Thus, for instance the number of filters can be modified simply by changing the size of the sublists of the main-list.

The value of an ‘ENP-box’ is updated using a Lisp function called ‘update-Synth-value’. For instance, the frequency value of the ‘sinosc’ box of Figure 5 can be controlled using the following simple Lisp code:

```
(dotimes (i 20)
  (update-Synth-value :pitch (+ 200 (* i 100)))
  (sleep 0.5))
```

The code plays a sequence of 20 pitches starting from 200 Hz and incrementing the frequency value by 100 Hz each time when calling the ‘update-Synth-value’ function. The timing of the updates of our example are controlled by the Lisp function ‘sleep’ that delays the next function call by 0.5s.

5 Synthesis Control using ENP

Whereas the control modules discussed until now – real-time sliders, mouse coordinates, MIDI controllers and Lisp code interface – are adequate in many situations, the control of acoustical instrument models, however, require often more sophisticated control tools. ENP was developed to meet these new challenges. The user enters in ENP the musical material in standard notation. The user can also add both standard and non-standard expressions that allow to specify instrument specific playing styles with great precision. ENP allows fine-tuning of timing with the help of graphical tempo functions and supports user definable performance rules. After the input is finished the score is translated into control information. The control system has been described in more detail in Laurson et al. 1999, Laurson 2000 and Laurson et al. 2001.

Figure 6 gives a typical example of how ENP is interfaced with the synthesis part of the system. The synthesis part of our example defines a guitar synthesizer based on the ideas presented in Karjalainen et al. 1998. On the left we have a window containing two patches: the first one, called ‘*SYNTH*’, defines the synthesis part whereas the second one - ‘*ENP*’- is responsible for the calculation of the control data. The input ENP score is shown in the background behind the main window.
6 Parameter Matrices

The interaction between ENP and PW_Synth is realized in complex cases – such as the guitar model example given in Figure 6 - with the help of graphical matrices. Each row and column are named. By taking the intersection of the row and column names a large symbolic parameter space is easily generated. Each resulting parameter name (pointing to an address in a C-structure) is a pathname similar to the ones found in other synthesis protocol schemes such as OSC (Wright and Freed 1997).

Our guitar synthesizer in Figure 6 is defined with the help of two matrices. The first one is a 6×6 matrix used for the sympathetic coupling of the strings. The second matrix is a 6×18 matrix that defines the string models (we assume that guitar model has 6 strings each having 18 parameters). Thus, if we assume that our current guitar synthesizer has the name ‘guitar1’, we can for instance refer to the loop filter coefficient (‘lfcoef’) of the second string (‘2’) of our guitar model with the pathname ‘guitar1/2/lfcoef’. We can use this protocol to build very complex instrument models such as a clavichord that can have over 50 string pairs. For instance a clavichord with 51 strings can be defined with a 51×18 matrix (we assume here that each string pair has 18 parameters). This naming scheme is powerful also because it allows the simultaneous use of many instrument instances with separate parameter paths. For instance we can easily add new instrument boxes (such as ‘guitar2’, ‘clavichord1’, etc.) each having a unique parameter name space.

Figure 7 below shows part of the contents of the 6×18 matrix of Figure 6 defining the parameter space for the string models. The left-most column contains the string numbers and the up-most row, in turn, gives the names for the individual parameters (due to space limitations we see here only the first 9 parameters). The intersection of each string/parameter pair contains a number that is used as a default value when the patch containing the guitar synthesizer is triggered.

7 Acknowledgments

This research has been conducted within the project “Sounding Score—Modelling of Musical Instruments, Virtual Musical Instruments and their Control” financed by the Academy of Finland.

References


