INSTRUMENT REUSABILITY SCHEME IN PWGLSYNTH

Mikael Laurson
CMT,
Sibelius Academy
Helsinki, Finland
laurson@siba.fi

Vesa Norilo
CMT,
Sibelius Academy
Helsinki, Finland
vnorilo@siba.fi

Mika Kuuskankare
CMT,
Sibelius Academy
Helsinki, Finland
mkuuskan@siba.fi

ABSTRACT

This paper presents our recent developments that aim to make the instrument definition process of our visual synthesis environment more accessible to a broader audience. There are several novel aspects that aim to overcome some of the classical limitations found in sound synthesis systems. After an introductory section we discuss two advanced examples where scores containing several model-based instrument parts can be realized without having to edit the original instrument definitions. In the first one, we can duplicate instrumental parts of a single instrument definition. In the latter example, we can mix in a single score several instrument models with the help of a mixer patch that is automatically created by system.

1. INTRODUCTION

The instrument/score paradigm in sound synthesis has already been investigated for about 50 years. For instance the recent article [1] enumerates some typical problems related to so-called *Music-N* programs such as instrumental reuse, parameter mapping between orchestra file and score file, and lack of graphical tools. Our focus in this article will be somewhat similar and we will present our solutions to these classical problems. Our synthesis environment, PWGLSynth [2], is related to the ones of the Music-N tradition as we deal with instrument definitions and scores. Our system is, however, different as it is strongly visually oriented. In the following we will address problems that are related to parameter mapping with musical scores, and especially focus on the reusability scheme of complex instrument definitions.

PWGLSynth is part of our visual programming language PWGL [3]. PWGL and its tight integration to its music notation program, ENP [4], provides a new and unique system for complex instrument design and synthesis control [2].

Until now our synthesis environment has mainly been used as a research tool where we have studied how model-based instruments can be controlled using ENP. In a typical case we have worked with one instrument model at a time. Thus the workflow using one instrument along with one or several scores has been quite straightforward. Recently, along with our public release of PWGL (www.siba.fi/PWGL), our focus in sound synthesis has shifted more towards end users, so that PWGLSynth could also be used for music production. For this end we need a more systematic and modular way to define instruments and how they relate to scores.

In a typical case the user defines the instrument visually along with special accessor methods that define the interface between the score and the instrument. In smaller instrument definitions this part of the process can be stored in a single patch. In more complex cases the user can use the user-library scheme provided by PWGL. Here the library normally consists of a folder containing a load file, an accessor code file, instrument definition file, and possibly various data files (typically sound samples). The latter library scheme supports also an autoload facility, i.e. an instrument is loaded automatically if the user opens a score that contains musical parts that utilize the instrument definition. All instruments known to the system are stored in a database.

Musical scores are central in our new scheme as they manage which instruments should be loaded in the system. Also scores are responsible for calculating control information and starting the sound synthesis engine for real-time playback. This protocol will help a user in complex instrument setups as several steps of the system are done automatically in the background. In more advanced cases the system is also able to instantiate several instances of an instrument automatically. A score can also combine several different instruments: in this case the score creates a special mixer patch where links are created to the required instruments.

We start this paper with an introductory section that provides a case study where we define an instrument along with a score that utilizes the instrument. After this we go over to more complex instrument definitions and show how the system manages scores that contain several instances of complex model-based instruments. We end with an advanced example where we combine several different instruments within one score. Here we can reuse instrument definitions with the help of a mixer patch that links and mixes dynamically the required instruments. We also show briefly how to deal with cases when the number channels of incoming instruments differ.

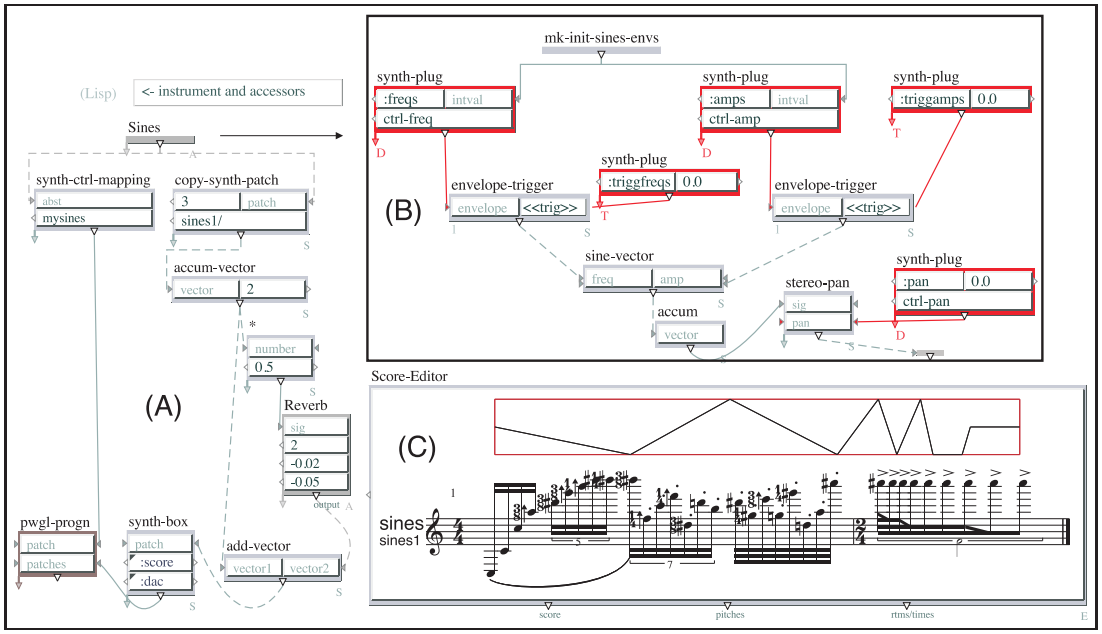

Figure 1: *An additive synthesis patch utilizing 3 banks of sine wave oscillators. (A) gives the top-level definition with three main entities: an abstraction 'Sines', a 'synth-ctrl-mapping' box, and a 'copy-synth-patch' box; (B) shows the contents of the 'Sines' abstraction with five 'synth-plug' boxes; (C) a musical excerpt is used to control the instrument definition given in (A) and (B). The break-point function above the staff controls the pan parameter.*

2. BASIC COMPONENTS: A SINE WAVE OSCILLATOR BANK CASE-STUDY

PWGLSynth instrument definitions are realized normally using the following scheme. A special box called 'copy-synth-patch' is used to copy the contents of an abstraction box count times. The abstraction contains a patch consisting of DSP-modules (boxes marked with 'S') and 'synth-plug' boxes. Thus the user defines the abstraction only once and the system automatically copies this model as many times as required. In order to distinguish between different duplicated patch instances 'copy-synth-patch' 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. The entry points are used afterwards to control the synthesis process. The symbolic references are pathnames, such as 'guitar1/1/freq' or 'guitar2/6/lfgain'.

Our scheme allows to associate an instrument to a score by adding to the 'synth-plug' boxes information about accessors. These accessors are short Lisp-based textual methods that are used to access information from a note object, such as midi, velocity, start-time, duration, and expressions. What is more important, the system is able to access the musical surroundings of a note, and thus accessors can deal with melodic, harmonic, and voice-leading formations. Furthermore, the user gives control labels - 'D', 'T', or 'C' - to the 'synth-plug' boxes whether they are used for discrete, trigger or continuous control purposes.

From this information (i.e. entry point pathnames, accessors and control labels) the system creates automatically both discrete and continuous control methods for the instrument in question. These control methods are generated by the 'synth-ctrl-mapping' box.

Figure 1 shows an overview of a typical instrument definition that can be split in three major parts. In a top-level patch, (A), a 'copy-synth-patch' box copies an abstraction called 'Sines' 3 times (the count value is given in the first input). The outcoming signal is mixed to a stereo signal that is reverberated by a 'Reverb' abstraction. The dry and wet signals are mixed together and the resulting signal is fed to a 'synth-box' that represents the final output.

The top-level patch has in the upper part a box called '(Lisp)' that contains a text-editor. Here we find in textual form an instrument class definition and accessor methods. In our example the instrument class called 'sines' is defined as follows:

```
(create-enp-instrument sines (synth-instrument)
  ()
  (:default-initargs
   :name "sines"
   :instrument-group :synth-instruments))
```

This class name is used later in accessor method definitions and for instrument names in a score.

The contents of the the 'Sines' abstraction is shown in (B). The patch contains besides ordinary synth-boxes such as 'sine-vector', 'stereo-pan', and 'accum', three discrete 'synth-plug' boxes (see the label 'D') and two triggered ones (labeled with 'T'). The third input of the discrete 'synth-plug' boxes are labeled with symbols that represent the accessor methods (we find here three accessors: 'ctrl-freq', ctrl-amp' and 'ctrl-pan'). Accessor methods have three arguments: an instrument class, a note object, and an instrument name. For instance the left-most 'synth-box' utilizes a fairly complex accessor method called 'ctrl-freq' that generates 10 frequency envelopes (one for each partial), and it is defined as follows:

```
(defmethod ctrl-freq ((self sines) note name)
  (flat
   (loop for b from 1 upto 10
```

```
collect
(let* ((fund (m->f (midi note)))
       (ys (loop for i from 1 upto 10
                 collect
                 (* fund (g-random 1.0 1.02)))))
  (convert-to-synth-env
      (mk-bpf (interpol 10 0.0 1.0 t) ys)
      (dur note))))))
```

Another accessor method, called 'ctrl-amp', is quite similar and it generates 10 amplitude envelopes.

A third accessor method, called 'ctrl-pan', calculates the panning value for the current note by accessing the break-point function that is seen above the note information of the score (C):

```
(defmethod ctrl-pan ((self sines) note name)
   (e note :bpf :sample :at note))
```

Finally, in (C), we have a score that generates control information for the instrument definition shown in (A) and (B). The score is associated to our instrument class definition from above (see 'sines' just before the staff-lines). Below the instrument class the user adds an instrument name (here 'sines1'). The instrument name is added to the entry point pathnames and can thus be used to distinguish different instrument instances belonging to the same class.

When calculating the score the system proceeds from left to right and for each note calls the accessor methods of the 'synth-plug' boxes that in turn feed the DSP modules with score information. To listen a score in real-time the user selects the 'Score-Editor' box and types the space key. This will automatically search for the associated synthesis patch, calculate the score, start the current synth box, and finally start the playback. Non-real-time mode can be evoked by changing the third ':dac' input of the 'synth-box' to ':file'.

3. SINGLE INSTRUMENT DUPLICATION PROBLEM

In the previous section we discussed a fairly typical and straightforward sound synthesis implementation problem, i.e. how to define a bank of sine wave oscillators. Here the sine wave bank is one conceptual unit that is simply copied as many times as needed. Thus we could, without problems, have a score that contains several polyphonic parts that plays our instrument.

A special problem arises, however, if we consider more complex instruments that are internally built out of several sub-entities such as strings. Figure 2 shows a top-level guitar model implementation. Here we copy a 'string' abstraction 6 times. This definition works only with scores that have one guitar part. If we introduce scores with several parts, then our guitar model definition cannot distinguish between different guitar model instances.

One solution to this problem would be to duplicate the 'copy-synth-patch' box in Figure 2 as many times as needed and then mix their outputs. While this workaround would fix our problem it is not optimal. In the worst case each score having different number of instruments would require a new instrument patch definition. A more elegant solution is to use in the patch two 'copy-synth-patch' boxes where one box calls the other one in a nested manner as can be seen in Figure 3. Thus here we can reuse our model definition for all scores having multiple guitar parts. Figure 4 shows such a score: here we need two guitar models that are distinguished by the control system.

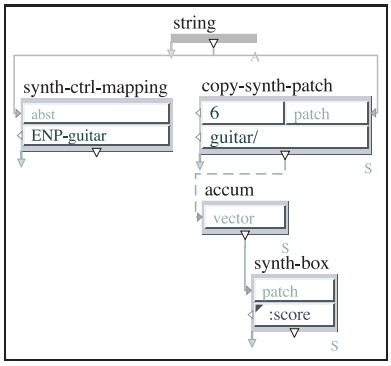

Figure 2: *A single guitar model definition.*

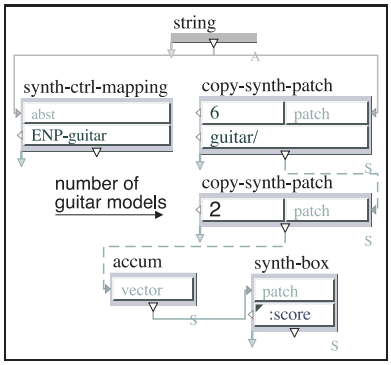

Figure 3: *A nested guitar model definition that allows to copy several models.*

4. SCORES MIXING MULTIPLE INSTRUMENTS

A similar problem arises when we have scores that have parts that should control different instrument models, say one part is utilizing a harpsichord model and another a guitar model (see Figure 5). Our problem is here similar to the one presented in the previous section as we want obviously to be able to reuse existing instrument model definitions. Otherwise we would be forced to combine the two model definitions by hand in a single patch and mix their outputs. What is worse we would have to redo this process for each new score that utilizes a new combination of instruments. Thus the reusability problem becomes here even more pronounced than in the previous section.

To solve this problem we create for each score referring to different instrument models a new mixer patch that has two main tasks. First, the mixer patch creates a link to all required instrument definitions. Second, it combines all signals for the final output. This scheme requires that all instrument definitions are stored in a database. Thus each time a score needs an instrument its definition can be accessed automatically. Figure 6 shows the resulting mixer patch when the user attempts to play the score given in Figure 5. Here the upper row of 'mixer-connect' boxes are accessing the two required instruments (i.e. 'enp-harpsichord' and 'enp-guitar'). The 'combiner' box and the 'accum' box in turn combines the two incoming signals and mixes them to the output.

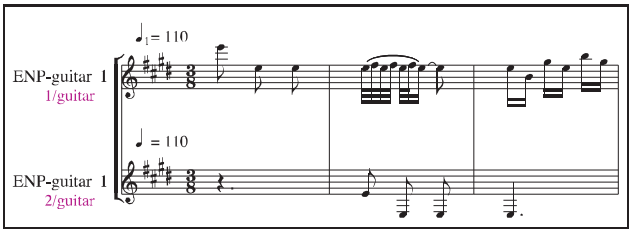

Figure 4: *A two part score that utilizes two guitar model instances.*

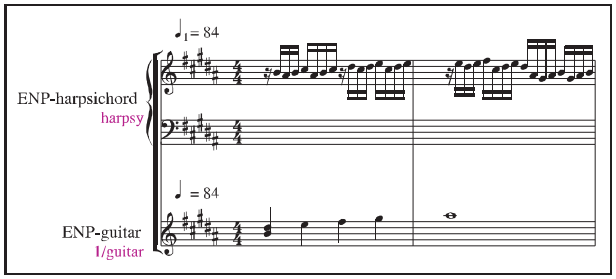

Figure 5: *A score that controls a harpsichord model and a guitar model.*

5. CHANNEL CONVERSION SETUPS

We end this paper by shortly discussing some problems that will appear if the incoming signals to the mixer patch in Figure 6 do not match. In our previous case this is not a problem as both incoming signals are mono signals. For cases, however, where one instrument results, say, in a stereo signal, and another one outputs a mono signal, we need rules how to combine these conflicting signals.

As the main rule the number channels of the mixer output is chosen from the source that has the maximum number of channels: thus if one source has 4 channels and another one 2, then the mixer will output 4 channels. In these cases we need to "upgrade" sources having less channels than the mixer output so that they will match the main output. Figure 7 outlines six commonly used cases, for instance how to convert a mono signal to stereo (for more details on how PWGLSynth operates with multichannel signals see [5]). In all cases we assume that the incoming signal is connected the the left-most input.

6. CONCLUSIONS

This paper gave an overview of how classical problems that are found in sound synthesis programs of the Music-N tradition are solved in our environment. We showed in an tutorial section how the main components of our system, such as the visual top-level patch, DSP abstraction, plug scheme, and accessor methods interact with a musical score. Then we focused on our main topic of this paper, and discussed two main cases how we can reuse existing instrument definitions in various score configurations. In a concluding section we outlined a scheme that allows to handle automatically cases where incoming signals to the main output mixer do not match.

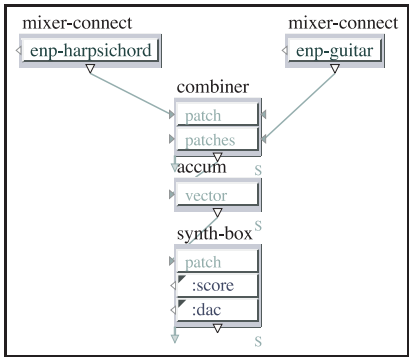

Figure 6: *An automatically created mixer patch where two instrument models are combined.*

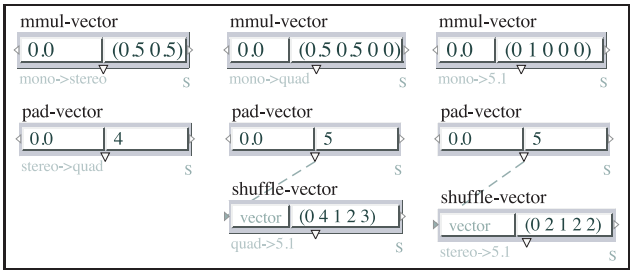

Figure 7: *Six output setups for converting mono signal to stereo, quad or 5.1 (upper row); stereo to quad, or 5.1; quad to 5.1 (lower row).*

7. ACKNOWLEDGMENTS

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).

8. REFERENCES

[1] Pedro Kröger, "CSOUNDXML: A META-LANGUAGE IN XML FOR SOUND SYNTHESIS," in *International Symposium on Music Information Retrieval*, Barcelona, Spain, 2004.

[2] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare, "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control," *Computer Music Journal*, vol. 29, no. 3, pp. 29–41, Fall 2005.

[3] Mikael Laurson and Mika Kuuskankare, "Recent Trends in PWGL," in *International Computer Music Conference*, New Orleans, USA, 2006, pp. 258–261.

[4] Mika Kuuskankare and Mikael Laurson, "Expressive Notation Package," *Computer Music Journal*, vol. 30, no. 4, pp. 67–79, 2006.

[5] Mikael Laurson and Vesa Norilo, "Multichannel Signal Representation in PWGLSynth," in *Conference on Digital Audio Effects*, 2006.