Instrument Concept in ENP and Sound Synthesis Control

Mikael Laurson and Mika Kuuskankare
Center for Music and Technology, Sibelius Academy, P.O.Box 86, 00251 Helsinki, Finland
email: laurson@siba.fi, mkuuskan@siba.fi

Abstract

This paper gives a summary of our recent developments in applications dealing with sound synthesis control. This work is based on our previous work where the focus was to develop tools to control physical models of musical instruments. Here the starting point was a score that was translated into a sequence of control events. These were used to control a real-time implementation of an instrument model. Recently we have extended this approach so that it can be used to control new virtual instruments. We concentrate in this paper in describing the instrument definition of our system and how it is used to calculate control information for sound synthesis.

Keywords

sound synthesis control, music notation, musical instruments, musical instrument modeling

1. Introduction

Our research team at Sibelius Academy has worked during the last three years to develop tools that can be used to control physical modeling of instruments. This research activity has resulted in new tools such as Expressive Notation Package (ENP, Kuuskankare and Laurson 2001) and PWSynth (Laurson et al. 2001; Laurson and Kuuskankare 2001a). The system has been used to control various instrument models such as the classical guitar (Laurson et al 2001), the clavichord (Välimäki et al. 2000), and two members of the lute family (renaissance lute and Ud, Erkut et al. 2001). Lately it has been also used to realize a piece for a 24-string virtual super guitar.
We use music notation as a starting point for generating control information for model based synthesis. The user enters a piece of music using a graphical user interface similar to the ones used by commercial notation packages. The difference is, however, that we have complete access to the underlying musical structures behind the notation front-end allowing to us to generate control information in a flexible and precise manner. Our system can be used by musicians without any programming expertise. Moreover, the system is capable of producing with a reasonable effort complete musical pieces thus allowing to the test model based instruments in a wider context than before. More information of our project - including several musical excerpts - can be found in the following home page:

www.siba.fi/soundingscore

During this work it has become obvious that our tools could be used also for other purposes. For instance, it can be used to generate control data in MIDI-format (an early attempt is reported in Laurson 2000b). Also it can be used to generate control information that is similar to the concept of 'orchestra-file' and 'score-file' that has been already used by several sound synthesis programs. While many of these control strategies have already been used in various systems, our scheme attempts to extend these concepts with the help of a rich score representation combined with a instrument class hierarchy.
Previous work related closely to our research is for example the Swedish "Rulle" system (Friberg 1991), where a rule based system was used to calculate control information from musical scores for MIDI synthesizers and for sound synthesis. The Chant system (Rodet et al. 1984), containing a compositional environment called FORMES (Rodet and Cointe 1984) and a software synthesizer, was designed to simulate convincingly the human voice. When compared to these systems, our environment has the advantage that the user can work directly with a high-level representation of a musical piece.

The rest of this paper is divided in four main sections. First we briefly describe our programming tools. Next we show how instruments are organized in our system as a class hierarchy. We also give several examples how this scheme is used for practical applications. Section four describes in detail the three stage process that is used to calculate sound synthesis control events. The final section discusses how special expressions, called macro-expressions, are implemented in the system.

2. PWGL and ENP2.0

This section briefly presents the programming environment we are currently using in our research activities. PWGL is a new visual programming language written in Lisp, CLOS and OpenGL. PWGL is based on PatchWork (Laurson 1996), which has been completely rewritten and redesigned. PWGL contains a score editor, which internally utilizes

a new notation software called ENP2.0. Like PWGL, ENP2.0 has been rewritten and its graphical layer is using OpenGL. Both PWGL and ENP2.0 will be presented soon elsewhere.
ENP2.0 is especially well suited for synthesis control as it supports a rich set of expressions. ENP expressions can be either standard (i.e. expressions with a more or less established meaning in instrumental music) or non-standard. The latter expressions also include groups. Groups can overlap and they can contain other objects such as break-point functions. Figure 1 shows an ENP2.0 example containing both standard and non-standard expressions. As ENP expressions have already been presented in several earlier papers we will concentrate here on the other entity of the system, namely the ENP instruments.

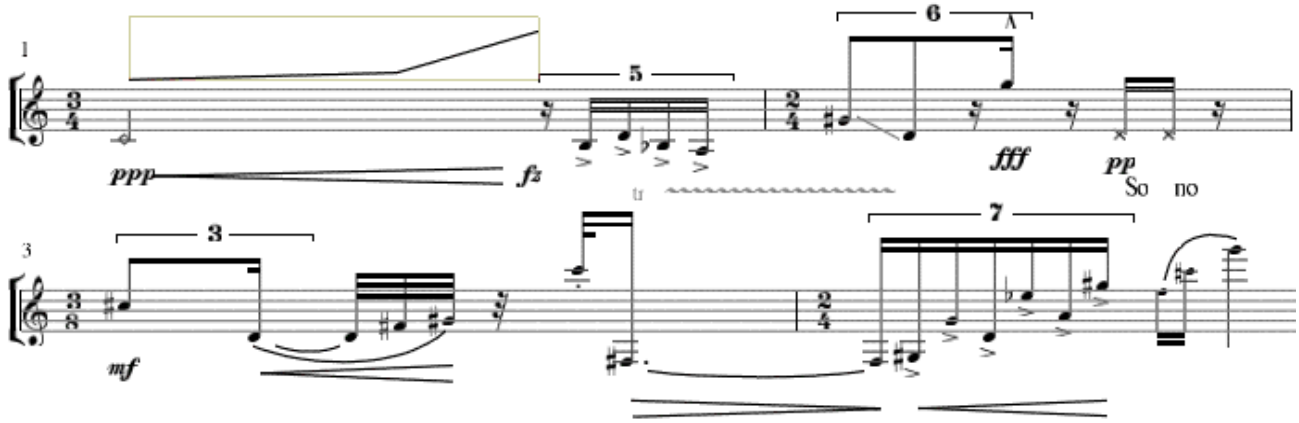


Figure 1. A score example with ENP expressions.

3. ENP Instruments

The instrument concept is essential when using ENP scores for calculating control information for sound synthesis. The score provides general information such as pitch, start time, duration, etc. that can be used by any synthesis instrument. The instruments are used, in turn, to define how a specific note or a note group should be realized by the system. We start by describing the instrument class hierarchy of ENP.

3.1 Traditional Acoustical Instruments

Instrument classes in ENP are organized as a class hierarchy. A part of this class tree is shown in the Appendix 1. All ENP instruments are subclasses of an abstract 'instrument' class. The 'instrument' class has direct subclasses such as 'aerophones', 'strings', 'keyboard', 'percussion', and so on. The 'strings' class has in turn two subclasses: 'plucked-strings' and 'bowed-strings'. This kind of categorization can be found in many instrumentation treatises for western musical instruments (see for instance Adler 1989). The 'instrument' class has following slots:

- name
- score-name
- abbreviated-name
- key
- transposition
- range
- written-clef
- sounding-clef
- instrument-group
- instrument-order

The 'instrument' class has many useful properties. It can be used by a notation program (such as ENP2.0) to denote the default clef, key and transposition of a given instrument. It can also be used as a data-base for other programs in our system. A typical example of this is for instance PWConstraints where the ENP instruments have been used to solve constraint-based problems that require detailed information of acoustical instruments (Laurson and Kuuskankare 2001b).
The figure in the Appendix 1 is not complete. For instance the 'strings' class inherits also from a 'string-collection' class which in turn contains instances of 'instrument-string' classes. The latter class represent the individual strings of a given string instrument. The 'instrument-string' class has following slots:

- name
- number
- base-pitch

- fret
- range

This scheme allows us to describe individual string instruments in a fairly detailed manner. For instance a classical guitar contains besides all slots of the 'instrument' class (ranges, clef, transposition, etc.) also information concerning the individual strings it owns. Thus we can represent instances of instruments having different number of strings (for instance the 6-string classical guitar and the 10-string alto guitar). These instances can be tuned in different ways (a guitar in standard tuning, a guitar where the sixth string is tuned one semitone lower than usual, etc.).

3.2. Synthesis Instruments

As can been seen in the Appendix 1 the 'instrument' class tree contains special synthesis instruments ('ENP-instrument', 'Synth-instrument') that are not part of the standard western instrument repertoire. These instrument definitions are used by our system mainly to produce control information for sound synthesis. Synthesis instruments can either be related closely to their acoustical counterparts or they can be fairly abstract without any close relation to existing acoustical instruments.

An example of the former instrument definition is for example 'ENP-guitar'. 'ENP-guitar' is a direct subclass of the 'guitar' class and thus it contains all the knowledge of the acoustical guitar. 'ENP-guitar' has been used to model a classical guitar (Laurson et al. 2001). In this context our definition of the ENP 'strings' instrument – described in the previous subsection - has proven to be very useful. For example, when simulating a stringed instrument - such as the classical guitar - it essential to be able to finger the given score information (i.e. each note must know its string).

The second category of synthesis instruments consists typically of abstract synthesis algorithms where many of the restrictions and properties of acoustical instruments – such as pitch, range, clef and key – have no definite meaning. The reason that we anyway wanted to add this category to the 'instrument' class tree is that our scheme allows to mix instruments with highly diverse properties. Thus a score can consist of parts for traditional instruments, physical models of acoustical instruments and/or abstract synthesis algorithms in any combination.

4. Calculation of Control Information

In this section we describe the overall three stage process for calculating control information from an input score in our system. We start with a preprocessing stage which takes an input score and applies all performance rules and tempo functions found in the score. After this the scheduler translates the note information into time-stamped processes. In the final stage we show how processes trigger specific methods that in turn generate the final control information.

4.2. Preprocessing

The preprocessing stage modifies various aspects of the note information given by the input score. The most important parameters that are processed here are the onset and offset times of the notes. Other possible parameters are for instance dynamics, vibrato rate, vibrato amount, etc. Timing information can be changed either by using tempo functions and/or performance rules, the other parameters, in turn, are calculated by the performance rules.

4.2.1. Tempo Functions

ENP allows fine-tuning of timing with the help of graphical tempo functions. In order to assure synchronization of polyphonic scores, all tempo functions are merged and translated internally into a global time-map (Jaffe 1985). Tempo modifications are defined by first selecting a range in the score where the tempo change should occur. After this a score-BPF is applied to the range. In order to facilitate the edition process, the score-BPF can be edited directly in the score (see Figure 2) which allows the user to synchronize visually note events with the tempo function:

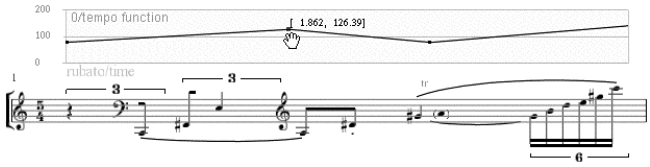


Figure 2. Edition of a tempo function.

Tempo function values are given in percentage (i.e. 100 means 'a tempo', 200 twice as fast, 50 half tempo). For instance an accelerando from 'a tempo' to 'twice as fast' is achieved with a ramp that starts from 100 and ends at 200.

In addition to conventional accelerandi and ritardandi, the user can apply special rubato effects ("give and take") to a group. This mode is especially useful when the resulting tempo starts to fluctuate too wildly. As in the previous case the user starts with a selection in the score and applies a tempo function to it. The difference is that the duration of the range is not affected by the time modification. Time modifications are only effective inside the selected range.

4.2.2. Performance Rules

Besides tempo functions, ENP supports user definable performance rules which allow the user to modify score information in a similar way as in the Swedish "Rulle" system. In Rulle, performance rules are used to calculate timing information, dynamics and other synthesis parameters. The main difference is that ENP rules use a syntax which was originally designed for PWConstraints (more information can be found in Laurson 1996).
ENP rules are written in two parts: (1) a pattern-matching part which is followed by (2) a Lisp code part. The pattern-matching part checks when the rule should be applied and also extracts relevant note information which is used later by the Lisp code part. The Lisp expression, in turn, executes the actual alteration.
For instance, a variant of the well known "notes inégales" rule ("in a beat of two eighth-notes the first note is made longer than the second one") can be translated into PWConstraints syntax as follows:

```
;; 1. pattern-matching part
(* ?1 ?2  (rtm-match? (1 ((?1 1) (?2 1)))            ;; 2. Lisp-code part
   (?if (write-key ?1 :duration
              (+ (read-key ?1 :duration) 0.1))
        (write-key ?2 :duration
              (- (read-key ?2 :duration) 0.1))))
   "notes inégales")
```

The pattern-matching part (line 1.) states that if two notes (?1 and ?2) are adjacent and form a two eighth-note beat, then the first note (?1) is lengthened by 0.1 s and the second one (?2) shortened by 0.1 s.
Another rule example that affects the durations of notes is the "tenuto" rule where a note is lengthened by 0.05 s if the note has a tenuto expression:

```
(* ?1
   (?if
    (when (find-expression? ?1 :tenuto)
          (write-key ?1 :duration
             (+ (read-key ?1 :duration) 0.050))))
   "tenuto")
```

Our final rule example affects the MIDI-velocity of notes. A note is played louder if it has a sforzando expression:

```
(* ?1
   (?if
    (let ((orig-vel (read-key ?1 :velocity)))
      (when (find-expression? ?1 :sfz)
         (write-key ?1 :velocity
           (min 127
             (+ orig-vel (* orig-vel 0.2)))))))
       "sforzando")
```

4.3. Scheduler

After the first preprocessing phase all notes know about their exact onset and offset times. In the second phase the notes are scheduled. This is done by creating a process object for each note. The process object has a slot that contains the current note. Thus the process object has all the knowledge of the note such as pitch, dynamics, instrument, etc. Furthermore, an active process object responds to two important timing methods: global-time and local-time. Global-time returns an absolute value in seconds denoting the current global clock-time of the scheduler. Local-time is a relative value – always scaled between 0.0 and 1.0 – which gives the current relative local time position within the life span of a note.
While the scheduler is running each process object sends to its instrument two main methods. The first method is responsible for generating discrete control information typically when the note is starting. The other method, in turn, triggers special methods that produce continuos control information. Both methods are discussed in the next subsection.
If the current note contains a macro expression (trill, portamento, arpeggio, etc.) then new note instances are created on the fly by instrument specific methods and each new note is inserted in the scheduler queue. Macro expression are discussed in more detail below in Section 5.

4.4. Control Methods

When a process object (containing the current note) is scheduled the system calls two main methods:

- `discrete-instrument-expression`
- `continuous-instrument-processes`

Both methods have two arguments: `instrument` and `process`, where the first argument is the instrument of the current note and the second argument is the current process object.

4.4.1. Discrete Control

The `discrete-instrument-expression` method creates any needed events to start the current note. A simple example is for instance a MIDI instrument (we use here a instrument called 'ENP-MIDI-instrument' which is a subclass of the 'synth-instrument' class) that creates a MIDI note-event for each new note:

```
(defmethod discrete-instrument-expression
            ((self ENP-MIDI-instrument) proc)
  (let ((note (note proc))
        (time (global-time proc)))
    (mk-ENP-ctrl-struct-MIDI-note
      time
      (* 1000 (read-key note :ENP-duration)))
      (pack-midi-info
           (chan note)
           (midi note)
           (vel note)))))
```

A more complex example is for instance a classical guitar instrument model such as 'ENP-guitar'. Here the `discrete-instrument-expression` method first collects information from the current note about the fingering, pluck position and playing technique and sends an excitation sample to a string model, which in turn results in a ringing tone (a more detailed discussion can be found in Laurson et al. 2001).

4.4.2. Continuous Control

The `continuous-instrument-processes` method is called each time a new note is encountered in the scheduling queue. It is used to create new process objects that will generate continuous control information for the synthesis. Our simplistic 'ENP-MIDI-instrument' example could be defined so that each note will have a gain envelope (this is simulated here by using the MIDI volume controller). The `continuous-instrument-processes` method calls another method `continuous-gain-proc,` which in turn makes a new instance of a process that contains a break-point function object (`mk-ENP-bpf-object`). This process is called '`gain`' and it will be called regularly by the scheduler each 1/100th of a second (tick = 0.01).

```
(defmethod continuous-instrument-processes
             ((self ENP-MIDI-instrument) proc)
  (continuous-gain-proc self proc (note proc))))

(defmethod continuous-gain-proc
          ((self ENP-MIDI-instrument) proc note)
  (let ((dur (read-key note :ENP-duration))
        (gain (calc-ENP-vel self note)))
    (add-process
       (clock-obj proc)
       (mk-ENP-bpf-object
        proc note 'gain
        (mk-bpf (list 0 dur) (list gain 0))
        :tick 0.01))))
```

Each time the scheduler calls a continuous control process it delegates a method called `continuous-instrument-expression` back to the instrument of the current process. In our example we generate a MIDI-volume controller event by sampling the break-point function object inside the process object by calling the expression`(value proc)`:

```
(defmethod continuous-instrument-expression
            ((self ENP-MIDI-instrument) proc)
  (let ((note (note proc))
        (time (global-time proc)))
    (case (param proc)
      (gain
        (mk-ENP-ctrl-struct-MIDI-controller
          time
          (pack-midi-info
```

```
(chan note)
7
(truncate (value proc))))))))
```

In more complex and realistic cases the `continuous-instrument-processes` could create several continuous control processes for various synthesis parameters such as gain, pitch, vibrato, pan, and so on (an example of this can be found in Laurson et al. 2001).
Our scheme also handles the classical synthesis control problem where a group of notes should be controlled by a single continuous control process. If, for instance, we would like to simulate a slurred passage of notes with our 'ENP-MIDI-instrument' example we could change the `continuous-gain-proc` method given above by simply changing the `dur` parameter so that it would match the total duration of the group. Also, we should check that `continuous-gain-proc` creates a gain envelope only when the current note is the first one in the group. The result would be a sequence of notes with a global gain envelope.

5. Macro Expressions

Macro expressions are useful when the user wants to use a notational short-hand in order to avoid a score crowded with details. These kind of expressions or ornaments have been used already for centuries (e.g. trill, tremolo, portamento). The details of the final execution of these expressions are typically left to the responsibility of the performer. Our system supports currently two ways of defining macro expressions.
The first one is text-based, where the user defines a method having three arguments: instrument, process and note. Appendix 2 gives a moderately complex and realistic example of a macro expression, called 'portamento-notes' that is specialized for the 'ENP-guitar' class. Here a portamento between two successive notes is simulated by inserting an extremely rapid slurred succession of chromatic notes between the main notes. Due to space limitations we cannot give a detailed explanation of the code here. The main idea is that we dynamically generate additional note events using the function `mk-ENP-macro-note`.
Although the text-based approach has been used to convincingly simulate ornaments it is clear that for more complex cases the text-based approach can be quite hard to code and understand. This is why we have utilized lately also a compositional system called Viuhka (Laurson 2000a) to generate ornaments. Viuhka has been originally developed for the Finnish composer Paavo Heininen in order to produce multi-layered textures. Viuhka is especially well suited to generate ornaments for our purposes for it is based on an idea where a given amount of time is filled with note events. The user can add tempo functions to make accelrandos or ritardandos. Also the result can be shaped by controlling the articulation and dynamic of individual notes within the expression in a flexible manner. Figure 3 gives a patch of a complex and long Viuhka ornament – the total duration is 6 s – that is divided in two main parts. The first part (4 s) is a trill with alternating pitches. The first part has also a complex tempo function with several accelerando/ritardando gestures (with tempo percentage values 100-200-150-200-100). The second part (2 s) is a ritardando gesture (with values 200-100) with note repetitions. Figure 4 gives a written-out version of the result.

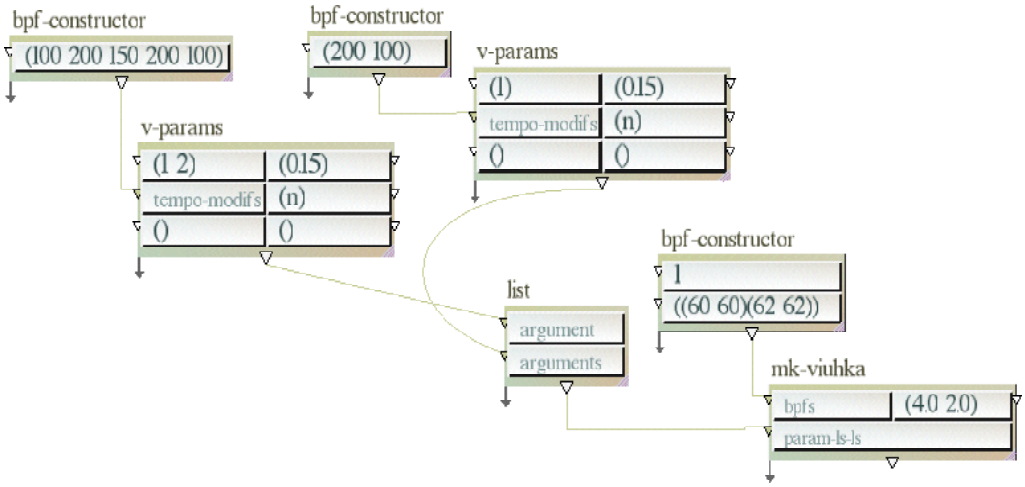


Figure 3. A Viuhka patch generating a complex two part ornament.

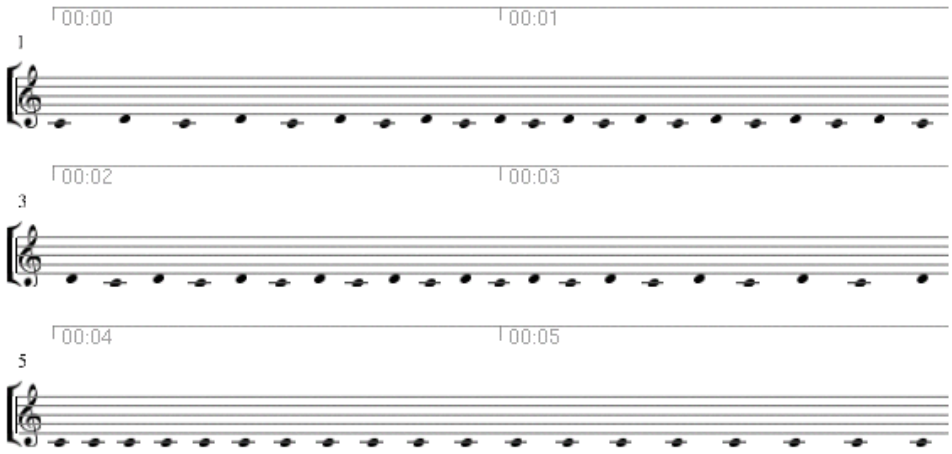


Figure 4. Result of the Viuhka patch of Figure 3.

6. Conclusions

This paper has presented how a music notation system can be used to generate control information for sound synthesis. Our focus has been in describing the instrument concept of ENP and how control information is calculated using the score information in conjunction with ENP instruments.

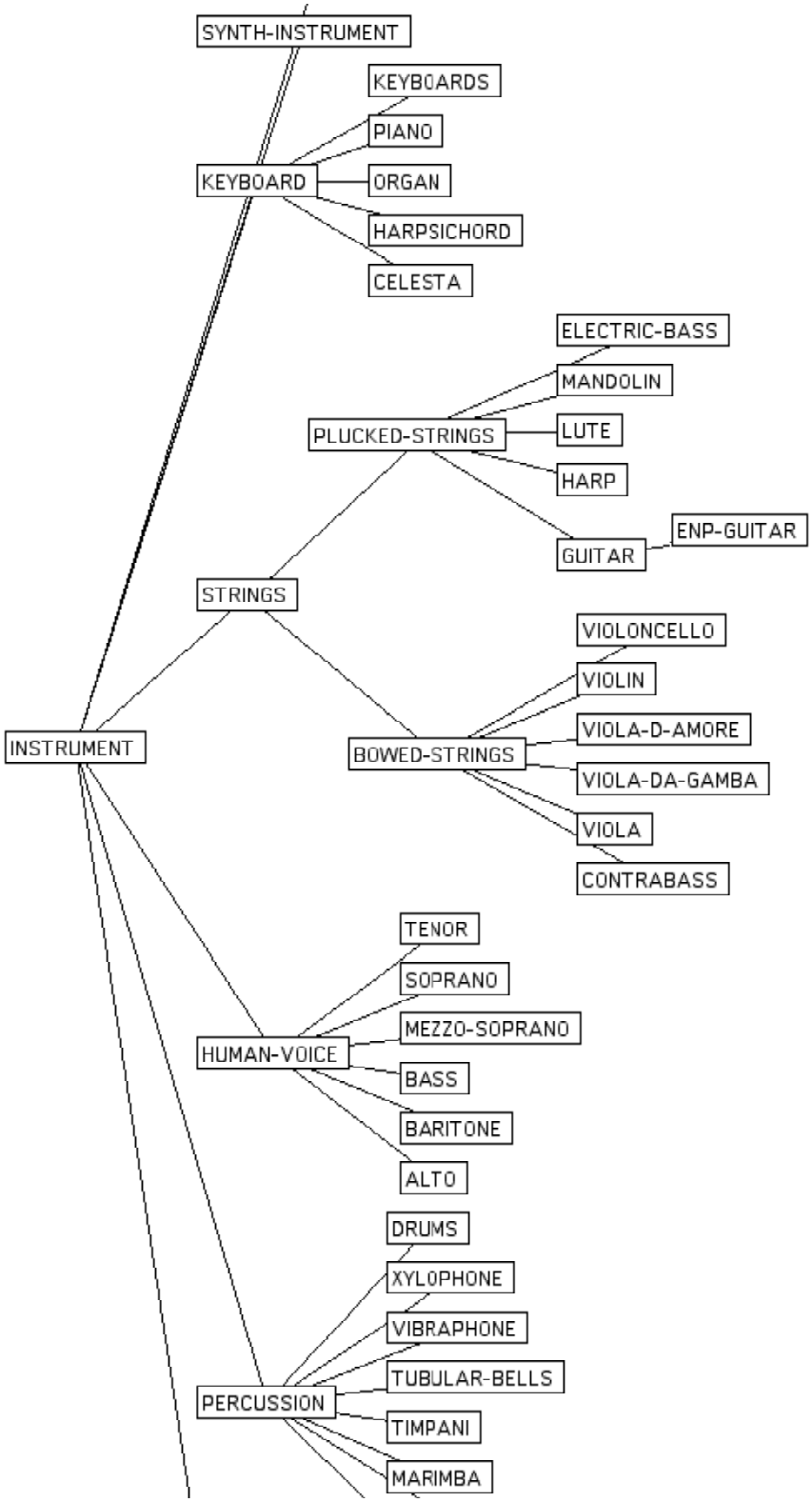
7. Acknowledgments

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

Adler, S. 1989. *"The Study of Orchestration"* W.W. Norton and Company. 2nd edition. New York - London.
Erkut, C., M. Laurson, V. Välimäki and M. Kuuskankare. "Model-Based Synthesis of the ud and the Renaissance lute". In Proc. of ICMC'01, Havana, Cuba.
Friberg, A. 1991. "Generative rules for music performance: a formal description of a rule system". *Computer Music J.*, vol. 15, no. 2, pp. 49-55.
Jaffe, D. 1985. "Ensemble Timing in Computer Music" . *Computer Music J.* , vol. 9, no. 4, pp. 38-48.
Kuuskankare M. and M. Laurson. 2001. "ENP, Musical Notation Library based on Common Lisp and CLOS". In Proc. of ICMC'01, Havana, Cuba.
Laurson, M. 1996. "*PATCHWORK: A Visual Programming Language and Some Musical Applications*". Doctoral dissertation, Sibelius Academy, Helsinki, Finland.
Laurson, M. 2000a. "Viuhka: A Compositional Environment for Musical Textures". In VII Brazilian Symposium on Computer Music, Curitiba.
Laurson, M. 2000b. "Real-Time Implementation and Control of a Classical Guitar Synthesizer in SuperCollider". In Proc. ICMC'2000, pp. 74-77, Berlin, Germany.
Laurson, M., C. Erkut, V. Välimäki, and M. Kuuskankare. 2001. "Methods for Modeling Realistic Playing in Acoustic Guitar Synthesis". *Computer Music J.*, vol. 25, no. 3, pp. 38-49.
Laurson M. and M. Kuuskankare. 2001a. "PWSynth: A Lisp-based Bridge between Computer Assisted Composition and Sound Synthesis". In Proc. of ICMC'01, Havana, Cuba.
Laurson, M. and M. Kuuskankare. 2001b. "A Constraint Based Approach to Musical Textures and Instrumental Writing". In CP01 workshop on Musical Constraints, Cypros.
Rodet, X., Y. Potard, and J.-B. Barriére. 1984. "The CHANT project: from the synthesis of the singing voice to synthesis in general". *Computer Music J.*, vol. 8, no. 3, pp. 15-31.
Rodet, X., and P. Cointe. 1984. "FORMES: composition and scheduling of processes". *Computer Music J.*, vol. 8, no. 3, pp. 32-50.
Välimäki, V., M. Laurson, C. Erkut and T. Tolonen. 2000. "Model-Based Synthesis of the Clavichord". In Proc. ICMC'2000, pp. 50-53, Berlin, Germany.

APPENDIX 1: ENP Instruments



APPENDIX 2: Portamento Expression

```
(defmethod portamento-notes ((self ENP-guitar) main-proc note)
  (let* ((time (read-key note :ENP-start-time))
         (dur (read-key note :ENP-duration))
         (str (instrument-string (read-key note :fingering)))
         (fret (fret (read-key note :fingering)))
         (vel (calc-ENP-vel self note))
         (next-fret (fret (read-key (first (notes main-proc)) :fingering)))
         (instrument self)
         (portspeed 0.06)
         (cur-time 0)
         notes)
    (when (and next-fret (find-expression? note :portamento))
      (let* ((portlen (1- (abs (- fret next-fret))))
             (up? (> next-fret fret))
             (frets (pw::arithm-ser fret (if up? 1 -1) next-fret)))
        (setq cur-time (- dur (*  portlen portspeed)))
        (push (mk-ENP-macro-note time cur-time (midi note) vel str (pop frets)
                                 t (give-pluckpos-val instrument  note)
                                (give-vibrato-val instrument note))
              notes)
        (while (cdr frets)
          (push (mk-ENP-macro-note (+ cur-time time) portspeed
                                (convert-fret-to-transp-midi instrument str (first frets))
                                (* 0.05 vel) str (pop frets)
                                (if (second frets) t nil) (give-pluckpos-val instrument  note)
                                (give-vibrato-val instrument note))
                notes)
          (incf cur-time portspeed)))
             (nreverse notes))))
```