SCORESYNTH
a System for the Synthesis of Music Scores based on Petri Nets and a Music Algebra †

Goffredo Haus, Alberto Sametti

L.I.M. - Laboratorio di Informatica Musicale
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Moretto da Brescia, 9
I-20133 Milano (Italy)

0. Abstract

In this paper we show how we can synthesize music scores by executing formal models. The kind of models we use is an "ad hoc" arrangement of Petri Nets and a music algebra; music objects are associated to *places* ; music transformations are described by algorithms associated to *transitions* and are coded by expressions which are based on the set of operators and the syntactic rules we have defined. This approach allows us to describe music objects and their transformations at various levels of representation.

The ScoreSynth system is made up of the Editor and the Executer of the models. The Editor allows us to describe hierarchical, timed, concurrent, deterministic/non-deterministic and weighted Petri Nets models; both places and transitions are used as the "morphism" nodes to implement hierarchy. Common Petri Nets structures can be described as macros. The Executer executes the models, synthesizes MIDI scores and shows a graphic map of the music objects as they are generated by the model execution; the musician, while looking at the graphic map, can interact with the model.

keywords: *music objects, music operators, music algorithms, Petri nets, modelling, timing, concurrency, synchronization, hierarchy, score synthesis, graphic user interface, macro structures*

† This research is partially supported by the Italian National Research Council in the frame of the MUSIC Topic (LRC C4, Grants N° 89.00031.69 and 90.00678.PF69): "INTELLIGENT MUSIC WORKSTATION", Goal C: SISTEMI AVANZATI DI PRODUTTIVITA' INDIVIDUALE, Subproject 7: SISTEMI DI SUPPORTO AL LAVORO INTELLETTUALE, Finalized Project SISTEMI INFORMATICI E CALCOLO PARALLELO.

1. Introduction

In this paper we show how music structures can be described, processed and synthesized by means of a more flexible kind of representation than the staff; in fact, symbols are chosen and organized depending deeply on instrumental needs within common music notation; the level of representation within scores is more detailed than the one of music composition and more abstract than the one of timbre modelling; the new kind of representation we are proposing makes up a conceptual music framework with as many different levels of abstraction as the musician needs and allows us to explicitly describe and process what we call *music objects* (MOs), both traditional music and non-traditional music objects. An MO means anything that could have a musical meaning and that we think of as an entity, either simple or complex, either abstract or detailed, an entity with a name and some relationship with other MOs.

Therefore, we can describe MOs at various abstraction levels within a hierarchical context of description; for example, we can have the following: the structural level, the score level, the timbre level and any other level we can think as a suitable one for our music representation purposes. Common music notation is characterized by many different languages (one or more languages for each level of representation). Furthermore, modern composers have introduced a myriad of notation conventions to represent concepts which the common music notation cannot represent. Our work is devoted to the definition of a *one and only language*, suitable both for every level of representation of common music information and for contemporary music representation (i.e. *concrete, electronic* and *computer* music).

To identify the most suitable description tool we have sought a formal tool which has the following properties:

* requires few symbols;
* has a graphical form of notation;
* allows hierarchical description;
* allows the description of MOs processing;
* allows time description;
* allows deterministic/non-deterministic models;
* allows macro definitions for common structures;
* shows the musician the score synthesized as the model execution goes on.

2. Score Modelling by Petri Nets and a Music Algebra

Since 1980 we have been using Petri Nets (PNs) as the basic tool for music description and processing (see Degli Antoni & Haus[1]); other experiments have been carried out by S. Pope[2] based on Predicate-Transition Nets.

Throughout our research, we have developed some programs to edit and execute PNs for both music and general purpose applications. The most recent program we have developed is ScoreSynth where we use PNs for describing, processing and synthesizing music scores; while we associate MOs to place nodes for describing music information, we associate transformation rules to PN structures and PN parameters (marking, numeric labels, algorithms, etc.). MOs may be processed modifying the superimposition and juxtaposition laws within PNs structure; on the other hand, PN parameters allow us to create *instances* of MOs which modify themselves according to the behaviour of PN models during executions. Furthermore, we can represent the same musical information by PNs at a lower or higher level of abstraction by means of suitable alternative modelling approaches. The modelling of music by PNs makes natural the description of the *structural* level of music as a multilevel environment within which MOs may flow concurrently and interactively; the more detailed levels of representation (the score and timbre levels) may be described and/or processed depending on the user choices.

Indeed, a PN model with a particular initial marking may represent a family of scores; a particular execution of the model synthesizes a specific score. By changing the initial marking of the model we change the family of scores that can be produced by model executions. A special case: when we have a fully deterministic model, we can produce one only score (given a particular initial marking).

The following paragraphs discuss the definition and implementation of music objects, Petri Nets and music algorithms within the ScoreSynth system. ScoreSynth has been developed on the Macintosh® family of computers within the MPW development environment.

2.1. Music Objects

The meaning of MO, as already mentioned, is strictly related to the place nodes in our implementation. By MO we mean not only any sequence of notes but also something more general. Something that is not exclusively linked to listening and is not connected with the idea of process, because it could last zero seconds, as far as the measurement of time is concerned. For example, an MO may be a control command for a sound synthesis device. Obviously the definition of MOs is partly affected by the standard of the music code we use (MIDI) and must stay within the bounds of this standard. Appendix A gives the complete BNF definition of MOs.

In this section we go through all kinds of MOs. In order to simplify the explanation, we name, without distinction, a place or its associated MO everytime it is obvious. The reader must keep in mind, however, that a place-node and an MO are two very different things. Moreover, we want to remember that the listening of a score takes place in a different time than its synthesis, i.e. the performance of the PN models.

2.1.1. Notes

If an MO is formed only by notes, each note must be specified through the parameters which are necessary for its reproduction through MIDI sound generators. These parameters are: the MIDI channel (that we can nearly compare to the instrument), the pitch, the intensity, the duration. The duration is expressed implicitly, by giving the starting and the ending point of a note. We want to remember that the MIDI standard imposes some limitation on the values of these parameters, which we discuss at the end of § 2.7. Literature on the MIDI standard is widely available.

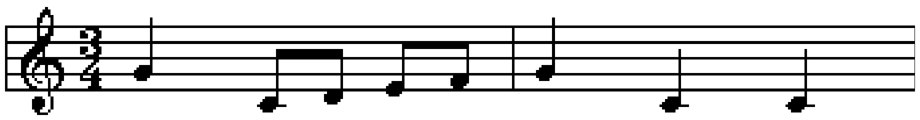


Figure 1: A simple MO.

2.1.2. Metaevents

All the events inserted in the coding of MOs which give general information on the score but are not directly performed because they cannot be coded by MIDI codes belong to the Metaevent class.

Metaevents include the Tempo (beats per minute), the Base (ticks per quarter note), the Time Signature (4/4, 3/4, 6/8, etc.) and rests. If there are rests, they must form a separate single MO because they cannot be explicitly described by MIDI codes.

The general lines of the work impose an intrinsic limitation to the score synthetized by the execution of a net. The first three types of metaevents are not coded directly into the score, but they are only a mark of the last modification. All MIDI events, which build up the score, have an absolute timing reference expressed in multiples of 0.5 msec. For this reason it is not possible to reconstruct a score in traditional notation if there is just one modification, for example, in the time signature. This problem does not occur when the tempo parameter remains fixed from beginning to end. This limitation affects the conversion of a PN model into a traditional score, but not the performance of the model via MIDI.

2.1.3. Other MIDI Messages

All MIDI messages which are not directly concerned with a note can be put in an MO. If the message belongs to the class *Channel Voice*, it can be put in the Note event; if the messages belong to other classes (i.e. *Channel Mode*, *System Common*, *Realtime*, *Exclusive*) the MO must be formed by only one of these messages and nothing else.

2.1.4. Coding Formats

MOs can be coded according to three syntaxes: a coding language which is unique to ScoreSynth, a portable alphanumeric MIDI coding language and a coding format which implements the Standard MIDI Files 1.0 format (both type 0 and 1). The first format is used internally by the program. The second format has been chosen for the sake of editing tools availability (i.e. common word processors). The third format has been implemented for file import/export. So, MOs may be defined either by a common text editor or a MIDI controller (a keyboard, a guitar, a microphone, etc.).

Before executing a PN model, ScoreSynth converts all the MOs (both ASCII and SMF) into the inner MIDI format, the only format that ScoreSynth directly works with. If the original files don't undergo modifications, they can be expelled in order to make the following executions quicker.

Building a net, the user can decide if a place node must have only a control role, namely without any chance to have MOs associated. On the other hand, the user can choose if the MO must be always saved on a file (permanent MO), for possible uses at the end of the execution, or kept in the memory (volatile MO), and so remain only till the end of the execution. These choices cannot be modified during the execution.

There are two other possibilities for the user. The first is to redirect all the MIDI events of an MO on a single specified channel among the attributes of the place to which the MO is associated. The second is to associate an MO to a place and to set whether it must be executed or it must be subject of following transformations.

2.2. Petri Nets Implementation

A Petri Net is usually defined by a quadruple <**P**, **T**, **A**, **M**>, where **P** is the set of places, **T** is the set of transitions, **A** is the set of arcs connecting both places to transitions and transitions to places, **M** is the marking of the net. Our implementation of Petri Nets extends the "classic" definition of Petri[3] (which is well exemplified with respect to modelling features of PNs in Peterson[4]), so that the musician can build up hierarchical music models defining MOs and music algorithms (MAs) made up of operator expressions. In this way, a Petri Net is defined by means of the triplet <**P**, **T**, **A**> and the following three statements:

(1) $\mathbf{P} \ni \mathbf{p}$ ≡ <*identifier, tokens, capacity, MIDI channel, object, play, file* >

(2) $\mathbf{T} \ni \mathbf{t}$ ≡ <*identifier, algorithm* >

(3) $\mathbf{A} \ni \mathbf{a}$ ≡ <*node-from, node-to, multiplicity* >

where:

P is the set of places; **T** is the set of transitions; **A** is the set of arcs; *identifier* is the label of the node (both place and transition); *tokens* is the number of tokens within the place, that is the value of the marking function **M** (see below); *capacity* is the maximum number of tokens allowed for that place; *MIDI channel* is the number of the MIDI channel where the MO, if one exists, has to be played; *object* defines whether an MO may be associated to the place or not; *play* defines whether the object has to be played or not; *file* defines whether the MO associated to the place, if one exists, is stored on file or not; *algorithm* defines whether the MA associated to the transition, if one exists, has to be executed or not; *node-from* is the identifier of the node which starts the arc; *node-to* is the identifier of the node which ends the arc; *multiplicity* is the numeric label of the arc. If a place has an associated MO which is stored on file, its *identifier* represents both the label of the node and the file identifier.

P, **T** and **A** are finite sets; **P** cannot be empty. Both objects and algorithms have their own syntax definitions; they are discussed in Appendices A and B, respectively.

The marking of a net is the distribution of tokens into the place nodes; their number and distribution may change during net execution. A marking function is defined as the following array:

(4) $\mathbf{M} \equiv (\mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_i}, \ldots, \mathbf{m_n})$

where **n** is the number of places within the net, **N** is the set of positive integers and every $\mathbf{N} \ni \mathbf{m_i} = \mathbf{M}(\mathbf{p_i})$ with $\mathbf{P} \ni \mathbf{p_i}$.

Then, the hierarchy of the model is described by the set:

(5) $\mathbf{S} \ni \mathbf{s}$ ≡ <*identifier, node-begin, node-end, recursion-level* >

which is the set of hierarchy links,where:

identifier is the label of the net, *node-begin* is the identifier of the beginning node for the morphism application, *node-end* is the identifier of the ending node for the morphism application, *recursion-level* is the level of recursion allowed for that net. We discuss how hierarchy and recursion are implemented in the ScoreSynth program in § 2.4. and 2.5., respectively.

The **S** set represents all the information about the hierarchy of the model, while the **P**, **T**, **A** sets completely describe a particular net of the model; in other words, a model is fully defined by an **S** set and as many triplets <**P**, **T**, **A**> as the number of the nets within the model.

Petri nets are commonly represented in the graphic form; we too use a graphic form of representation with the following set of icons:

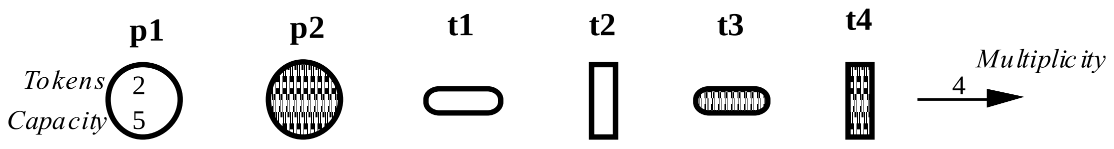


Figure 2: Petri nets icons.

The above icons represent (from left to right): a named simple place with tokens and capacity attributes, a named place which represents a subnet (see § 2.4. about morphisms), a horizontal named simple transition, a vertical named simple transition, a horizontal named transition which represents a subnet, a vertical named transition which represents a subnet and, at last, an arc with the multiplicity attribute set to four.

2.2.1. The Firing Rule

The execution of a net is made up of transition firings. A transition may fire if each one of the input places, i.e. places which are connected with oriented arcs to the transition, has at least one token. The transition firing has two effects: to decrement the marking of each input place by one token and to increment the marking of each output place, i.e. a place which is connected with an oriented arc from the transition, by one token.

In the following paragraphs we see how this basic firing rule is affected by capacity, multiplicity and timing parameters; this firing rule can be considered as a particular case of the firing rule we have defined for our extended implementation of PNs; from this point of view we can describe the basic firing rule with the following settings: all multiplicities are equal to 1, all capacities are infinite and all durations of transition firings are null.

After the starting of a net, firings follow one another until there are no more transitions which may fire. At the end of transition firings, the execution of the net stops.

2.2.2. Capacity

Capacity is a feature of place nodes which defines the maximum number of tokens allowed for each place. The capacity feature implies that the firing rule previously defined has to be modified; conditions of transition firings do not change with respect to the input places while transitions cannot fire if the marking of one output place, at least, will exceed its capacity after transition firing, i.e. the marking of that place added by the number of tokens due to transition firing is greater than its capacity.

Two simple examples show this concept. In Fig. 3a transition **t1** cannot fire because place **p2** is marked with as many tokens as its capacity; in Fig. 3b only one transition may fire, either **t1** or **t2** but not both. This simple net is a conflict structure, which is a non-deterministic one.

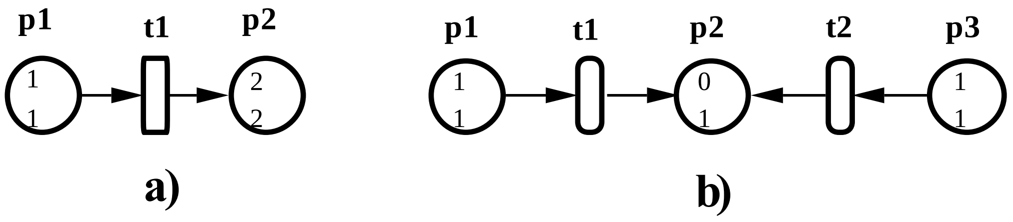


Figure 3: a) firing inhibition due to the capacity of the output place.

b) firing conflict due to the capacity of the output place.

2.2.3. Multiplicity

To improve Petri Nets' descriptive power and for the sake of graphic simplicity, we have assumed the multiplicity extension, that is, the ability to add numerical labels on the arcs. The multiplicity extension is a partial implementation of self-modifying nets, which were introduced by Valk[5].

Also the multiplicity feature requires modifications to the firing rule: a transition may fire if each one of the input places has at least as many tokens as the numerical label on the arc connecting the place to the transition; the transition firing has now the following effects: it decrements the marking of each input place by as many tokens as the numerical label on the arc connecting the place to the transition, and increments the marking of each output place by as many tokens as the numerical label on the arc connecting the transition to the place. Fig. 4 shows the behaviour of a transition firing within a simple net with multiplicity on some arcs.

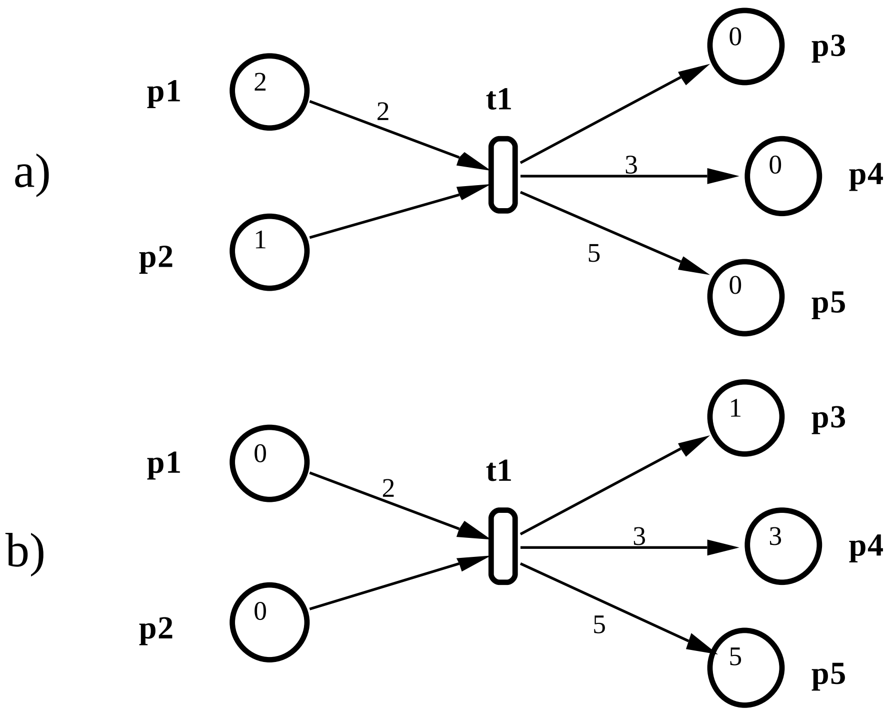


Figure 4: an example of transition firing: a) transition t1 may fire;

b) the effect of the firing of transition t1.

2.3. Timing

PNs are particularly suitable for describing concurrent processes and for controlling their synchronization or lack thereof. When a transition may fire, we don't know when it fires; and the duration of firing is assumed to be instantaneous. When we use PNs to describe asynchronous models we have no problem with this feature. There are no measurements of time within Petri Nets: it is the behaviour of the net that implicitly determines timings. It is the structure of the net which determines the firing sequence, without respect for their durations. On the other hand, we can synchronize MOs simply by the suitable structuring of the nodes' connections.

Finally, the firing sequence changes among different executions of the net; so there is no correspondence between a net and its firing sequence. We can have many transitions qualified for firing at the same time and we cannot know which one fires first: the net does not describe this kind of information. So, every execution of a PNs model may give a different firing sequence.

This feature may seem to be too abstract for the modelling of real complex systems. Several approaches are known regarding how to describe timed events' durations within PNs. Most of them represent timed events by means of timed transitions. Our approach associates timed MOs to places and their transformations to transitions. Both the firing of a transition and the execution of an associated MA have a null duration. In this way, when a token is put into a place with an associated MO the token cannot be considered for the firing of transitions connected to the place until the associated MO has ended.

Let us consider the net of Fig. 5; there are two places with associated MOs: MO **A** with place **A** and MO **B** with place **B**. This example is concerned with the duration of processes but not with their meaning. MO **A** lasts 1 second and MO **B** lasts 6 seconds. Every token which is put into place **A** is not disposable for 1 second; tokens which are put into places **p1** and **p2** are immediately disposable, i.e. the amount of time which occurs between the firing of **t1** and that of **t2** is null.

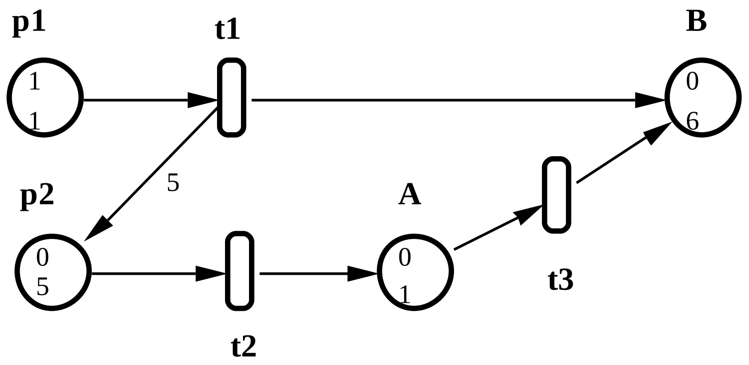


Figure 5: example of timed net.

If place **A** has one token, then **t2** cannot fire due to the capacity of place **A**; if place **B** has one token, then **t3** can add tokens to place **B** (6 as the maximum) when it fires. The effect is that a certain number of instances of MO **B** may run concurrently.

Let us analyze some steps of the net execution. Fig. 6 shows the marking at time 0 after both **t1** and **t2** firings; we can see that both places **A** and **B** have their own token, so that MO **A** and MO **B** are running (see Fig. 7); at the end of MO **A** (the shorter one), the token of place **A** is disposable and starts the firing of **t3** which makes the second instance of MO **B** running; at time 1, one of the tokens of place **p2** lets **t2** fire, which makes MO **A** running (see Figg. 8 and 9).

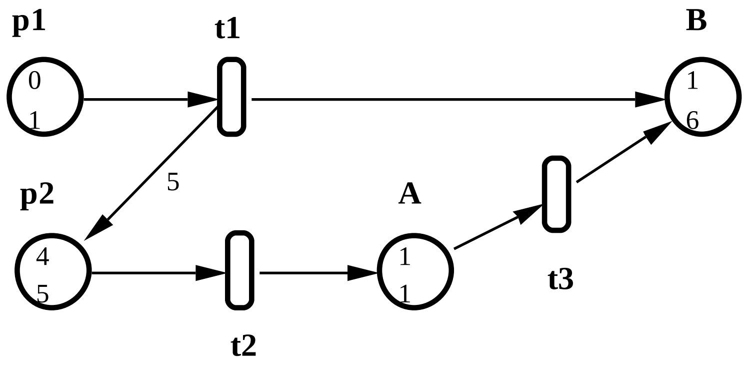


Figure 6: the marking at time 0.

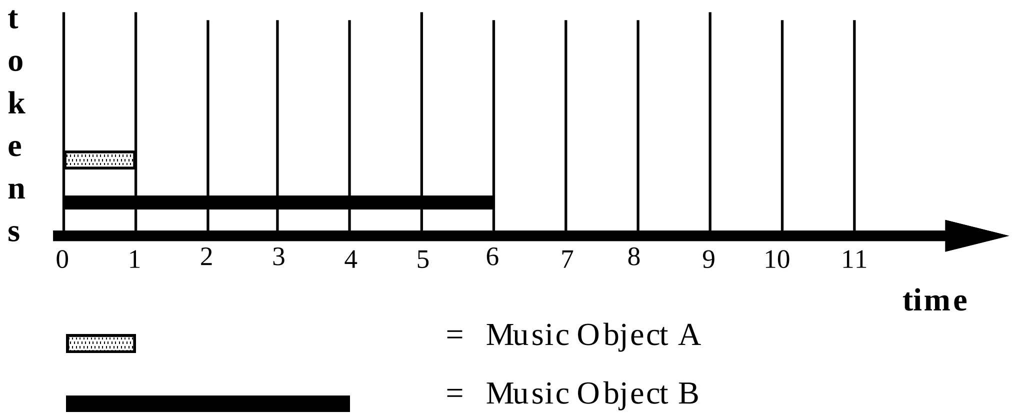


Figure 7: MOs started at time 0, after t1 and t2 firings.

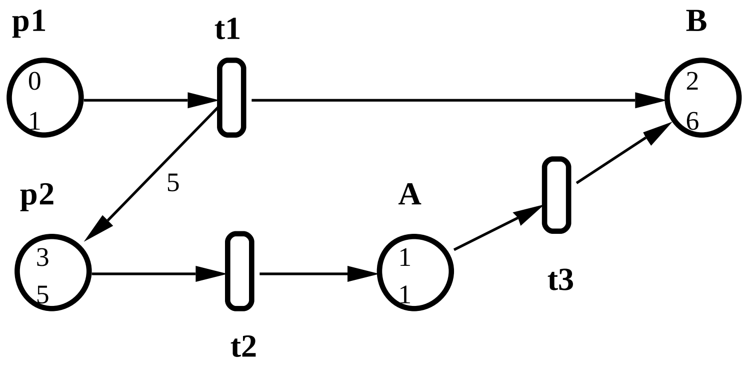


Figure 8: the marking at time 1, after both t3 and t2 firing.

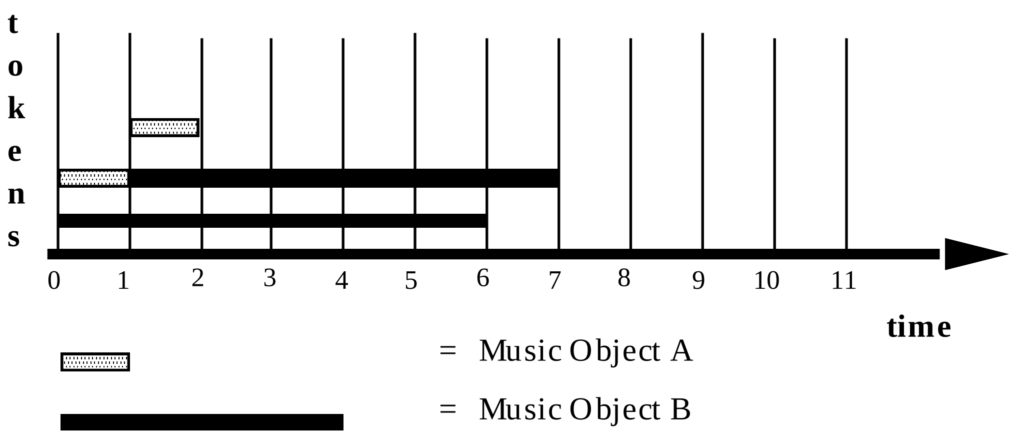


Figure 9: running MOs at time 1; new MOs started at time 1.

The firing sequence happens every second in the same way four times; the last firing happens at time 5; the complete result of the net execution is shown in Fig. 10.

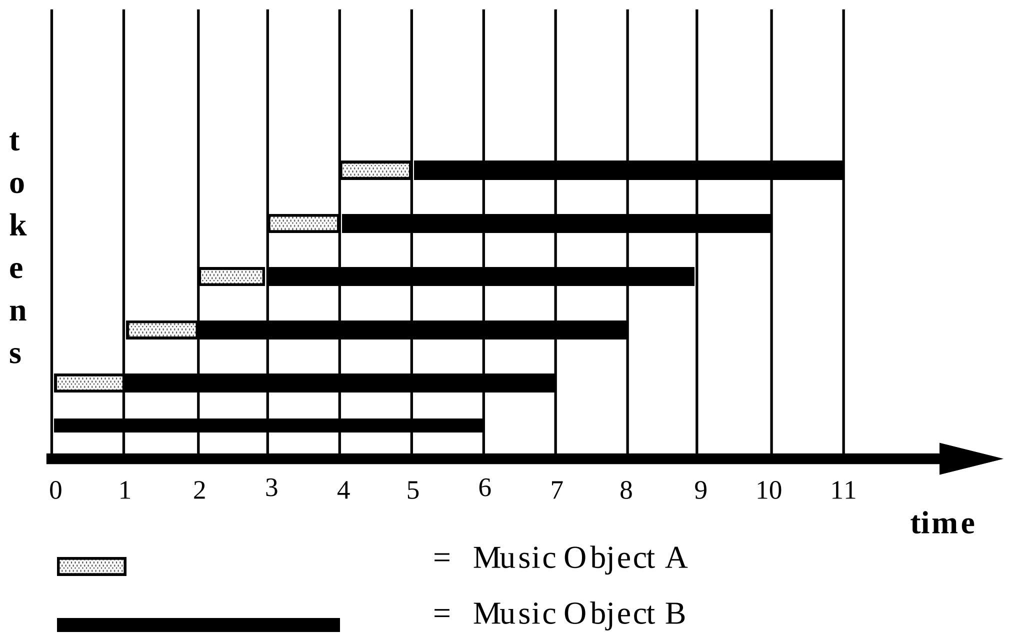


Figure 10: the complete map of the MOs generated at the end of the firing sequence.

2.4. Morphisms

Theoretical literature about morphisms and related formalisms is broad and multifaceted. We limit our discussion to a particular kind of morphism which seems to be suitable for our purposes: the refinement morphisms. The hierarchical approach introduced by Kotov[6] is rather close to our implementation. Refinements can define very complex PNs models by means of simple PNs and hierarchical structures, i.e. allowing models to be designed by either a top-down or a bottom-up approach. A refinement, that we call a *subnet*, is a PN which gives a more detailed description of a node (either a place or a transition) of the upper level of abstraction. We call that node the "father" node and that subnet the "daughter" net.

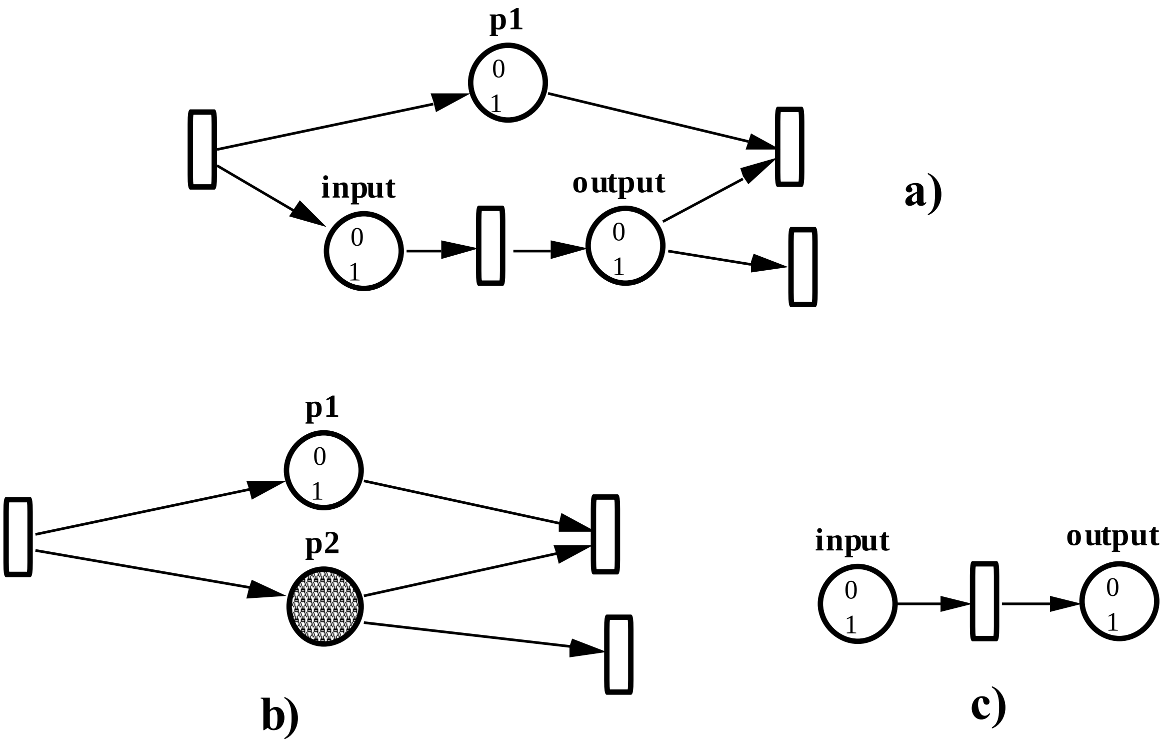


Figure 11: a) the net we have designed; b) the high level net; c) the p2 subnet.

If we choose to define a subnet associated to a place, then the daughter net must have two special places: the input place and the output place; input arcs of the father place are input arcs of the input place of the daughter net; output arcs of the father place are output arcs of the output place of the daughter net. Transitions can be refined in the same way. An example of a subnet which refines a place is shown in Fig. 11a, 11b and 11c.

The conceptual framework of our approach is discussed in Haus & Rodriguez[7]. This approach describes a whole complex model as a single large net; this large net may be either the result of many refinement steps in the frame of a top down detailing (see also Goguen[8]), or the starting information basis on which we can apply abstractions as many times as we want to implement the hierarchical structure in the frame of a bottom up modelling (see also Bertoni et al.[9]).

2.5. Recursion

Recursive nets can also be modelled. Both direct and mutual recursion are allowed. In ScoreSynth to avoid recursive expansions with no end, we must assign to every recursive net its recursive level number. That is to say, we have to specify how many times the net can call itself. As an example see Fig. 12. In this model we use a recursive net: **p1** (Fig.12b). If we choose a recursion level equal to 1 (only one call of a net inside itself), we obtain the expanded net of Fig. 13, in which the dashed lines represent the deleted arcs during the sequence of recursive refinement steps (depth first) that give the expansion and the bold lines represent the added arcs needed to implement refinement morphisms. Input is the input place of net **p1**. **p1** is the output place of net **p1**. Numbers in the small squares near the bold lines indicate the step of the expansion in which the arcs are added. Numbers in the small circles near the dashed lines indicate the step of the expansion in which the arcs are deleted.

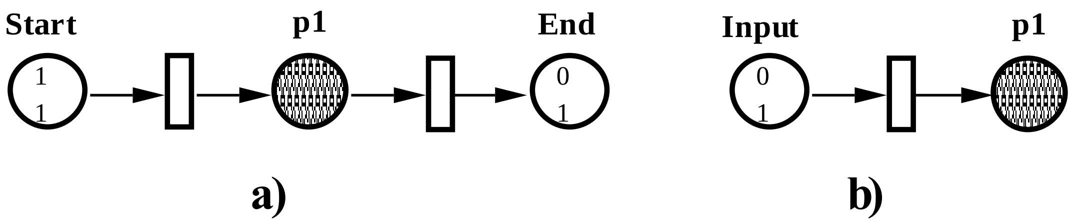


Figure 12: a) simple recursion example net; b) recursive subnet p1.

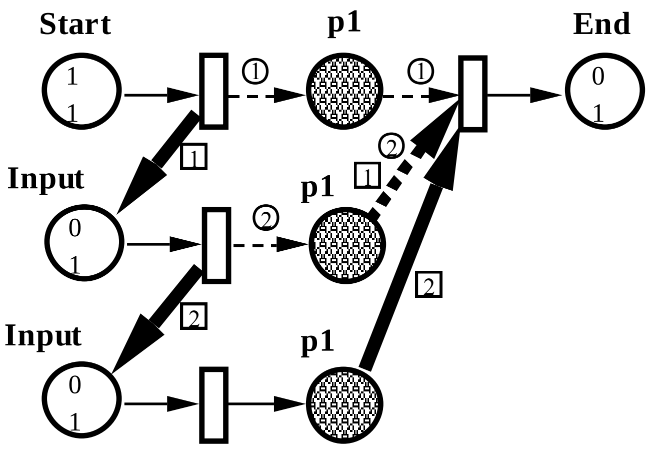


Figure 13: the result of the expansion of place p1 with recursion level 1.

2.6. Structuring Music Objects

Starting from two simple MOs (**p1** and **p2** of Fig. 14), we can see the elementary structures in Fig. 15 necessary to design more complex nets. The places with a name are those to whom an MO is associated. When a token reaches a place with an MO, it activates a "process" that is the execution of the MO itself. This "process" stays in the *running* state for all the duration of the MO. In Fig. 15 we also show the scores resulting from the executions of the corresponding nets.

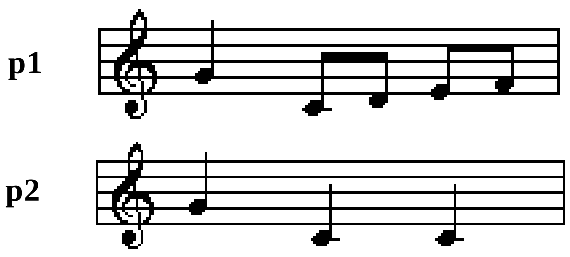


Figure 14: the two MOs p1 and p2

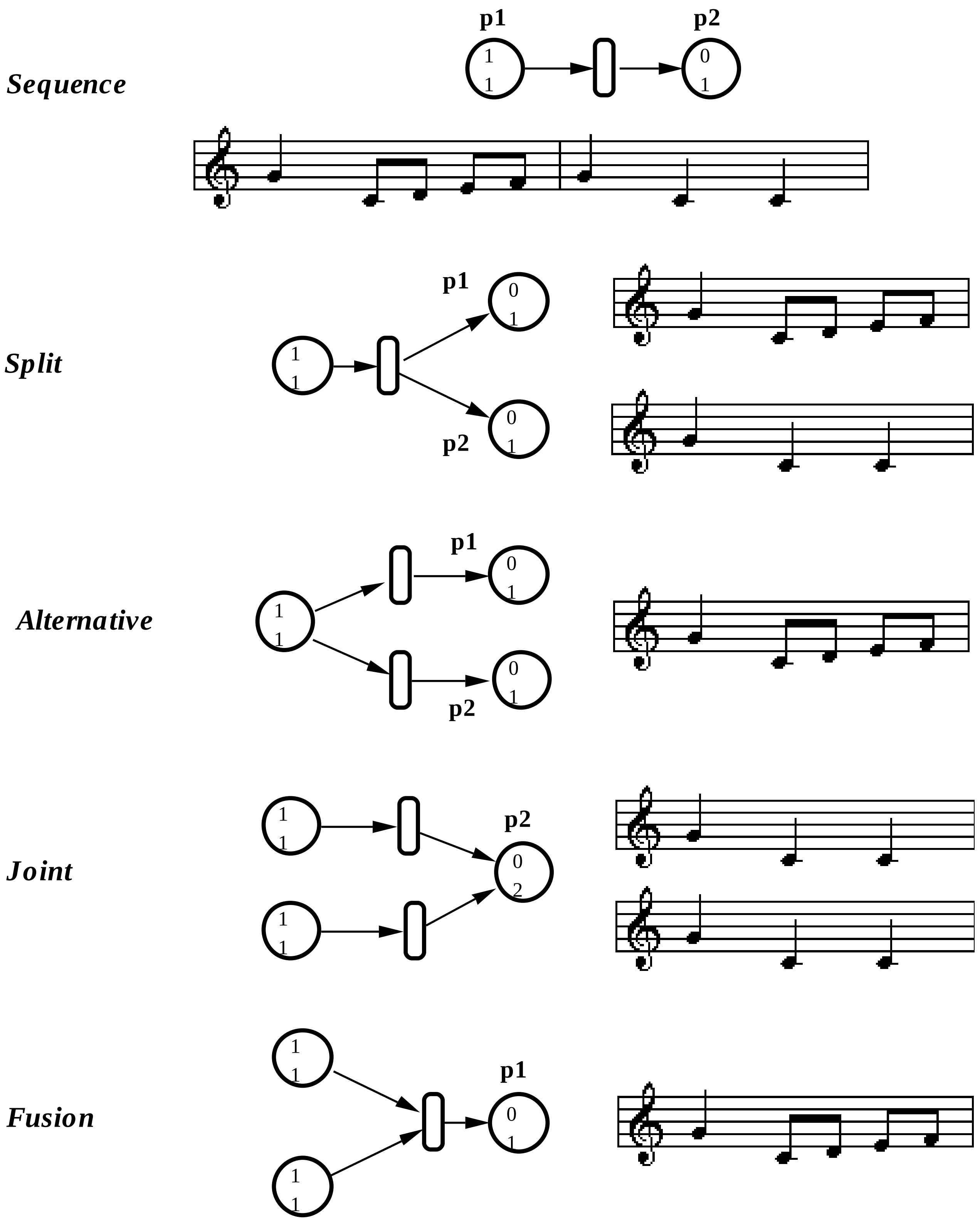


Figure 15: elementary PN structures.

A very special case is the third one. This kind of net introduces non-determinism. In this case ScoreSynth determines the firing sequence with a pseudorandom uniform distribution. All the considered elementary structures, can freely be joined to form more complicated PNs structures.

A little more complex structure is the loop structure. Two possible implementations are shown in Fig. 16.

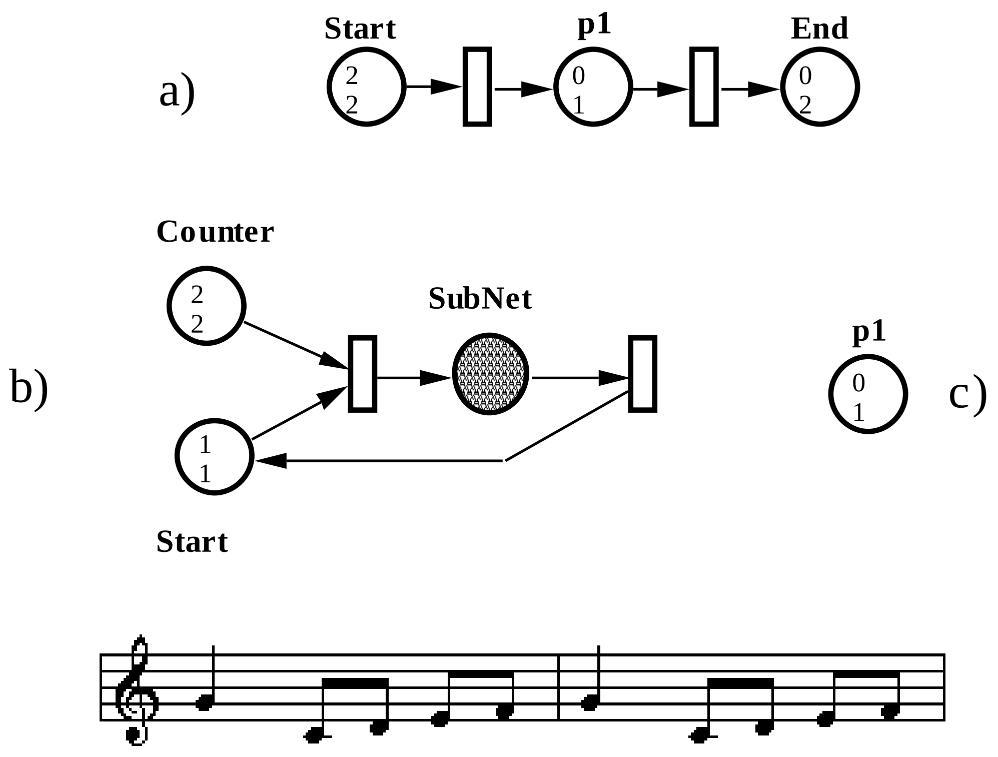


Figure 16: two different PNs (a , b + c) for the loop structure and the score resulting from their execution.

2.7. Music Algorithms

In this paragraph we briefly discuss how we have implemented the ability to work with MOs in ScoreSynth. The main idea is to allow transformations of both the parameters of sound (pitch, timbre, intensity, duration) and the order in which notes appear. Appendix B gives the complete BNF definition of MAs.

MAs join with transitions. If a transition has an associated MA, when the transition fires the MA is applied to the MOs associated to the input places and the outcoming object is placed in all the output places allowed for MOs. During the processing of an MA, it is also possible to use information from every MO available at that moment. We have observed that the modelling of music is as greatly improved as the expressive power of the formal language adopted if it is possible to have all the model information at disposal. We have studied formalisms which allow us to reach all nodes within the model without loss of time. In early research we have considered an approach which was similar to that of the statecharts introduced by Harel et al.[10]; then, we have defined a simpler mechanism which allows the immediate availability of all the information of the whole PN model as possible input arguments for MAs.

An MA can be formed by one or more single transformation. Each transformation may affect either the notes' parameters or their order. Before making a transition fire, ScoreSynth merges all the MOs in the entrance of a transition; it applies the transformation on the single MO obtained and associates the outcome of the transformation to each of the output places to which is possible to associate an MO. The transformation can be applied only to note events. All the other MIDI commands are filtered and lost.

2.7.1. Metacharacters, constants, variables

If **X** stands for the object on which the MA is processed at the input, the MA can be applied to the whole **X** or just to one of its subsequence, chosen by the user by the definition of the positions of both the first and the last note. The MA is applied to the range of notes which is set in its header, to each note as it occurs. The header also defines the kind of parameter to which the operator expression must be applied: pitch, duration, intensity (key velocity), MIDI channel and the order of the affected notes.

When we draw up an operator expression we can use metacharacters, i.e. special characters, which are needed to determine relevant transformations. The implemented metacharacters are: ***$***, ***?***, ***!***, ***%***. While ***$*** and ***%*** are the same as constants, ***?*** and ***!*** are variable quantities. We examine them one by one:

- ***$*** contains the total number of the notes in **X;**
- ***%*** contains the total number of the notes in the subsequence of **X** on which we have defined to operate;
- ***?*** contains the value of the parameter, referred to the current note, that we want to change;
- ***!*** is an index that contains the position of the note on which the operator expression works; the position values is referred to the first note to be changed within the sequence.

2.7.2. Operators' Syntax

As we have already mentioned, the parameters of the sound that we can modify at the level of MIDI code are pitch, duration, MIDI channel and intensity (key velocity).

Without dwelling on the concept of channel in MIDI, we want to explain that a channel stands approximately for an available instrument; we cannot directly operate on the characteristics of the sound, but only change the channel on which we want to send the notes.

The nucleus of an operator expression is simply an integer expression, that is, an expression with operators +(sum), -(subtraction), *(multiplication), /(division) and parentheses (,). The terms of an operator expression can be integers, metacharacters, alphanumeric pitch codes.

Here is an example to show a syntactically correct operator expression, with symbols, pitch, metacharacters and integers:

(2 * C3 / 4 - ?) * !

Each of these operator expressions must be preceded by a header that specifies to which parameter the operator expression is applied and which part of an MO must be transformed.

We mention the meaning of each terminal character that identifies the kind of transformation:

C (Channel)	= operator on channels;
D (Duration)	= operator on durations;
L (Loudness)	= operator on intensities;
M (Multiply)	= operator for multiplying notes;
R (Rotate)	= operator for rotating notes;
P (Pitch)	= operator on pitches;
I (Inversion)	= operator for inverting the ordering of notes;
K (Kill)	= operator for deleting notes;
S (Save)	= operator for preserving notes.

In order to use information about any MO with respect to duration (rhythm), pitch, MIDI channel and intensity, it is possible to specify the identifier of the MO and the starting note inside it.

For example, **P: 1, $, [Theme, 1], ? + 2** means that, to compute the new values for the pitches of the current MO, the pitch values of the notes within the MO **Theme** (starting from the first note) are used. In this example metacharacter **?** is referred to the MO **Theme**, while **$** to the current MO. See figure 17 for a simple net that shows the use of such an MA associated to **Alg** transition. The new MO obtained from the transformation of **Theme** is joined to the place **Theme.mod**.

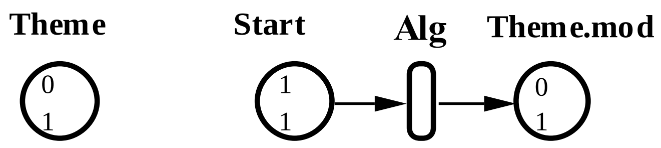
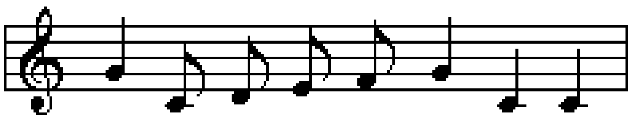


Figure 17: a simple net with an MA.

If the upper bound of the range exceeds the ordering number of the last note in **Theme**, then it is automatically modified to fall inside the bounds. If the current MO is empty then not only the pitch values of **Theme** are taken, but also channel, duration and intensity values, according to the range specifications.

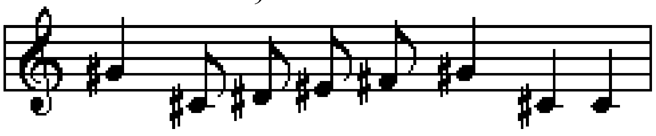
2.7.3. Operators' Examples

Starting from an MO (see Fig. 18 below) we will show some examples of feasible operators with a table containing on the left the operator and on the right the resulting MO.



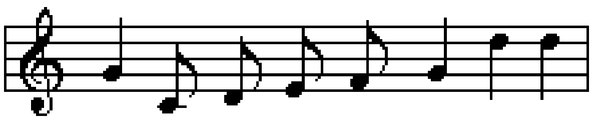
{the starting MO}

P: 1, $, ? + 1



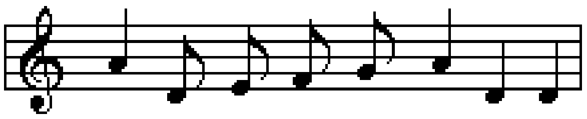
{Tonal Transposition}

P: $ - 2, $, 2 * G3 - ?



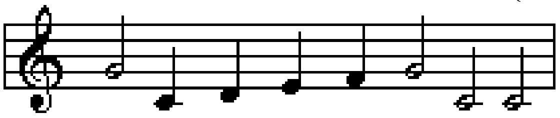
{Specular Inversion }

P [C] : 1, $, ? + 1

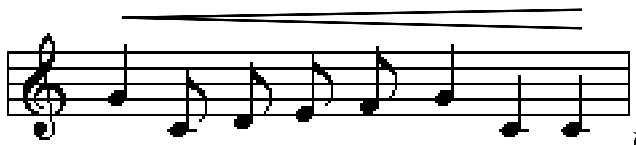


{Degree Transposition}

D : 1, $, ? * 2

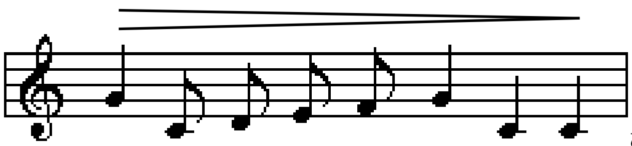


L: 1, $, ! * (127 / %)



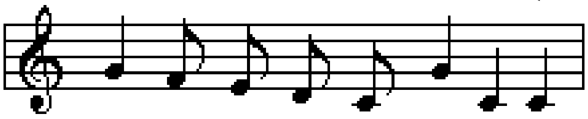
{Crescendo}

L: 1,$, (%-!+1) * (127/%)



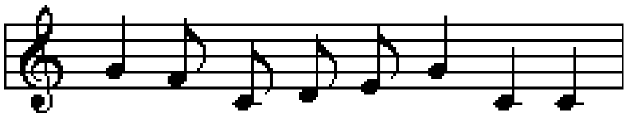
{Decrescendo}

I: 2, 5



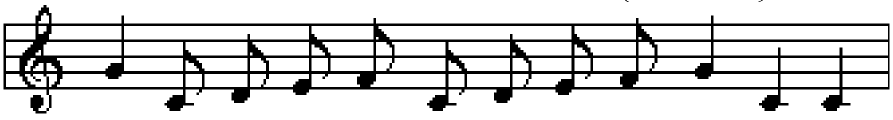
{ Retrogradation }

R: 2, 5, 1

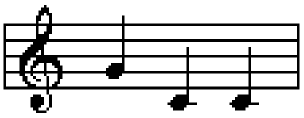


{ Rotation }

M: 2, 5, 2



K: 2, 6



S: 5, 7

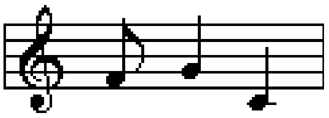


Figure 18: examples of operators.

Operators on pitches can even be applied according to scales which are different from the chromatic one; this must be done if we want the outcoming sequence in the same tonality as the beginning one. ScoreSynth has the following set of scales on which we can operate: *major, harmonic minor, pentaphonic, minor thirds, Debussy's whole tones*. The third operator, for example, uses a C major scale as indicated by the statement ***[C]***.

These are the choices made by ScoreSynth when the result of operator expressions goes beyond upper and lower bounds:

Channel (1..16): we obtain the channel with a mod16 operation;

Pitch (0..127,C -2..G 8): if the value obtained is more than 127, then we subtract the cardinality degree of the current scale as many times as is needed to obtain an allowable pitch value; if the value obtained is less than 0, then we must add the cardinality degree of the current scale as many times as necessary to obtain an allowable pitch value;

Duration : if it is less than 0, then it is set to 0;

Intensity : if it is less than 0, then it is set to 0; if it is more than 127, then it is set to 127.

2.8. "Macro" Petri Nets

When we are designing a net model, we often have to create many similar nets, which differ in their place attributes and MAs, but which are identical in their structures. With a *macro* net we can use just one net as the base model and then modify its attributes and/or MAs at our need. So it is also possible for the user to create his own libraries, containing commonly used PN models. Appendix C gives the complete BNF definition of macro nets.

The way we can do this is by writing, for every subnet place or subnet transition we want, a modifier list for the net which is loaded into memory and actualized before the execution of the whole expanded net.

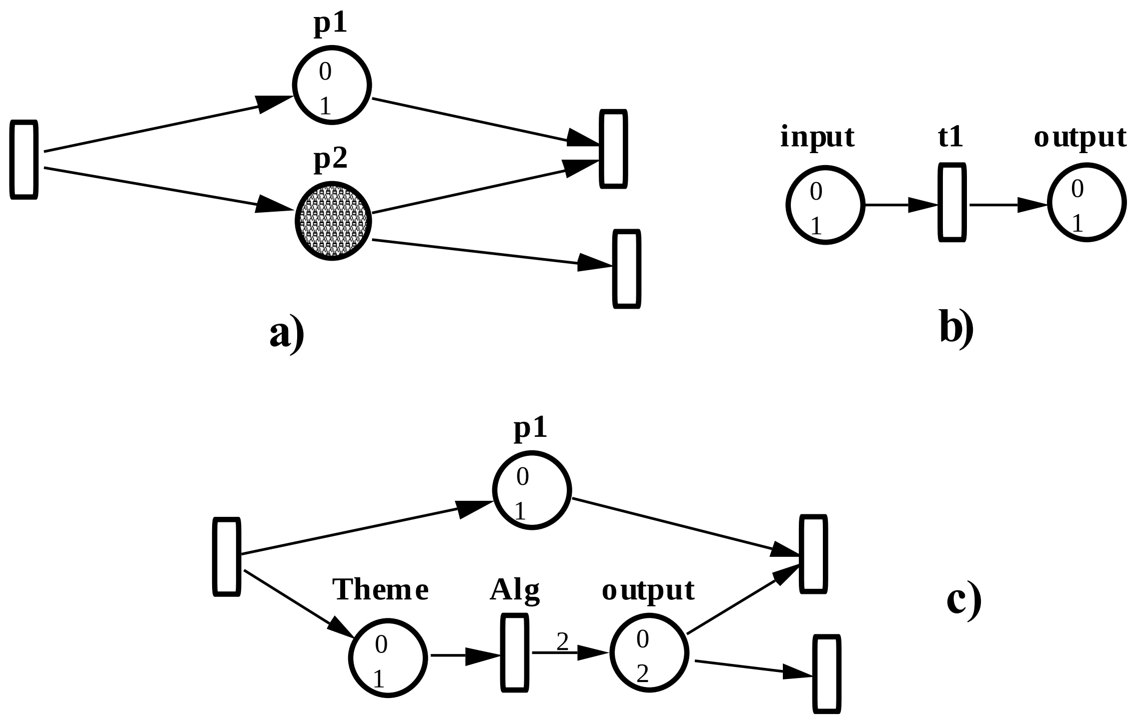


Figure 19: a) the top level net; b) the subnet p2;

c) the net resulting from the macro expansion.

An example that shows the use of a macro net invoked by a modifier list is included in the net model of Fig. 19; we choose to expand place **p2** using the following modifier list :

```
input : Theme              { change the name of place "input"
mute                          and tell not to play it }
output : C = 2             { change the capacity of place "output" }
t1 : Alg {C : 1, $, 3   { change the name of transition "t1"   and add an algorithm
P : 1, $, ? + 1}             consisting of two operators }
Alg -> output = 2          { change the multiplicity of arc "Alg -> output" }
```

3. Score Synthesis

3.1. Model Execution

Here we briefly discuss the main steps of the model execution algorithm. Appendix D gives a pseudocode version of the algorithm, but first we introduce such a "code" with a natural language explanation.

The main loop is included in the *Execute* procedure. The *Execute* procedure controls again and again the passing from a temporary blocking to the following; a temporary blocking happens when all the tokens within a net are in places either with an associated MO or without output arcs or with all output transitions which cannot fire.

The minimum time interval (*DeltaTime*) is computed for every cycle among all the active MOs within which one process at least terminates. Next, all active processes are executed for a *DeltaTime* number of time units; the variable which represents the virtual clock is increased by *DeltaTime* units; all terminated processes are deleted from the list of active processes. Execution stops when the *Firings* function evaluates that no transitions have fired and no active processes exist.

The *Firings* function takes care of transition firings. It simulates again and again the execution of all transitions either which may fire in the frame of alternative and/or conflict net structures or which may have simple firings (i.e. which don't belong to alternative and/or conflict net structures), until there are no more transitions which may fire. The *Firings* function returns control to the *Execute* procedure when there is a temporary block, that is, no more transitions may fire at that time.

3.2. The ScoreSynth Executer

In the Executer environment the user can interact with some window dialogues that we are going to describe. The main dialogue is that of Fig. 20. It gives to the user some facilities concerning the playback of the generated scores. The user can control up to 10 scores in this dialogue and can modify some listening parameters of the current selected score. It was not our intent to design a complete sequencer environment. If the user wants all the features included in any commercial sequencer he has only to export the score as Standard MIDI File.

These are our sequencer features: Play, Stop, Pause and Rewind the song; changing the tempo velocity in the column named Vel; starting playing the score at different points using the scale under the Pause button; looping the song from the starting point to the end; reassigning MIDI channels, defined during the PNs model execution phase, to other ones.

It is also possible to interact with the execution of the net model invoking the Music Objects Map dialogue that we'll discuss in the next paragraph.

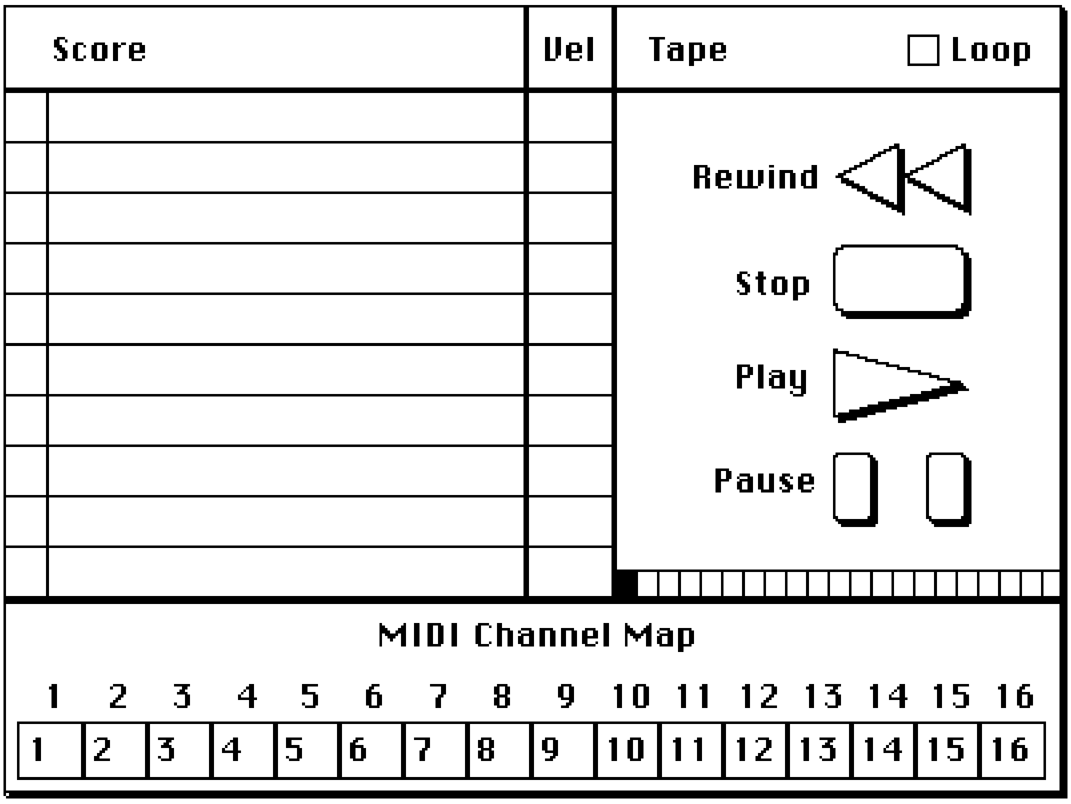


Figure 20: the Executer window.

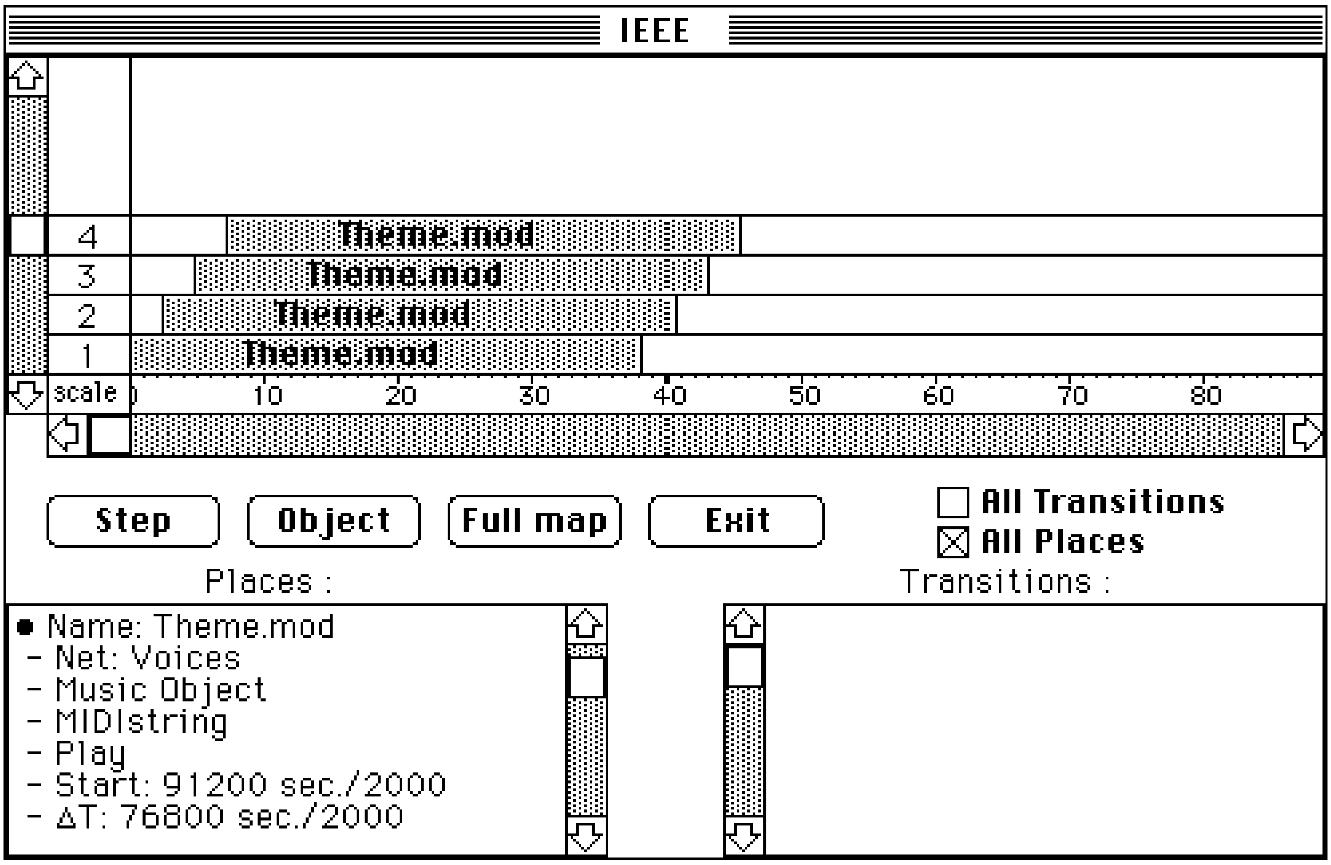
3.3. Music Objects Map

A musician trying to create a net model suitable for his purposes may often want to see on the screen how the execution of his model proceeds and the causal relationships between transitions' firings and score synthesis, i.e. MOs appending in the score. In this way, he can stop the execution as soon as he finds a bug in the model and go to the editor environment in order to modify it. This is possible simply by running the execution of the model and looking at a graphic window that shows on a time-MIDI_channel plane the map of MOs as they are appended in the score.

Two possibilities are available: either simply to run the execution and look at the growing map, or to run the execution, while looking at the map, and automatically stop it every time a new MO is created (i.e. every time a token reaches a place with an associated MO).

In the last case, the user who discovers a bug (e.g. a token gone elsewhere) can obtain all the information about places' and transitions' attributes and make the needed improvements on the current model.

Fig. 21 shows the dialogue that displays in the upper part the running execution in a graphic form. The vertical scroll bar lets the user move on the MIDI channel axis (1 to 16); the horizontal scroll bar lets the user move on the time axis. The user can select the time scale opening a dialogue by clicking the "scale" box near the point (0, 0). In the lower part the dialogue shows all the information the user may need to control the execution of the net model (all the attribute values of places and transitions every time the user stops the execution).



● Name: Theme.mod
- Net: Voices
- Music Object
- MIDIstring
- Play
- Start: 91200 sec./2000
- ΔT: 76800 sec./2000

Figure 21: object map dialogue with attributes.

4. Ending Remarks

Our research has shown that music structures within PN models can be transformed simply by:

* modifying markings, MIDI channel specifications, capacities of places;

* modifying labels of arcs;

* modifying MOs associated to places;

* modifying MAs associated to transitions;

* modifying the structure of PNs;

* executing non-deterministic PN models many times.

While the editing of PN structures and parameters affects causal/structural relationships within the architecture of scores, the editing of both MOs and MAs affects the basic information units and their transformations. The ScoreSynth system seems to be a powerful means for describing and processing music, closer to music thinking and perception than common music notation. People interested in a wider listening to of pieces synthesized by PN models execution can consider the collection "Musiche per poche parti" by G. Haus (compact disc Stile Libero/Virgin Dischi, SLCD1012, Milano, 1990). Other music productions are currently in the workings.

Acknowledgements

Many thanks to E. Bianchi, A. Camurri, G. Degli Antoni, G. Jacomini, A. Rodriguez and R. Zaccaria for their contributions to previous researches and experiences about Petri Nets and music. Particular thanks to S. Scolaro for his relevant contribution in the design and development of some features of the ScoreSynth Rel. 2.0 (graphic interface and PNs' macros, especially). Moreover, special thanks are due to the L.I.M.-Laboratorio di Informatica Musicale staff for the scientific and technical support provided all throughout our research.

Macintosh® is a registered trademark of Apple Computer, Inc.

5. References

1. G. Degli Antoni, G. Haus, "Music and Causality", **Proc. 1982 ICMC**, Venezia, Computer Music Association Ed., San Francisco, 1983, pp. 279-296.

2. S. Pope, "The Development of an Intelligent Composer's Assistant", **Proc. 1986 ICMC**, Den Haag, Computer Music Association Ed., San Francisco, 1986, 16 pp..

3. C. A. Petri, "General Net Theory", **Proc. Joint IBM & Newcastle upon Tyne Seminar on Computer Systems Design**, 1976.

4. J. L. Peterson, **Petri Net Theory and the Modeling of Systems**, Prentice Hall, New Jersey, 1981.

5. R. Valk, "Self-Modifying Nets, a Natural Extension of Petri Nets", **ICALP 1978**, LNCS, N. 62, Springer, Berlin, 1978, pp. 464-476.

6. V. E. Kotov, "An Algebra for Parallelism based on Petri Nets", **MFCS 1978**, Proc. 7th Symposium, Zakopane, Polonia, Springer, Berlin, 1978.

7. G. Haus, A. Rodriguez, "Music Description and Processing by Petri Nets", **1988 Adv. Petri Nets**, LNCS, N.340, Springer, Berlin, 1989, pp.175-199.

8. J. A. Goguen, "Complexity of Hierarchically Organized Systems and the Structure of Musical Experience", **UCLA Computer Science Dept. Quarterly**, Vol. 3, N. 4, 1975.

9. A. Bertoni, G. Haus, G. Mauri, M. Torelli, "A Mathematical Model for Analyzing and Structuring Musical Texts", **Interface**, Vol. 7, N.1, Swets & Zeitlinger B.V., Amsterdam, 1978, pp.31-43.

10. D. Harel, A. Pnueli, J. P. Schmidt, R. Sherman, "On the Formal Semantics of Statecharts", **Proc. Symposium on Logic in Computer Science**, Ithaca, New York, IEEE Computer Press, Washington, 1987, pp. 54-64.

11. S. W. Smoliar, **A Parallel Processing Model of Musical Structures**, PhD. Th. MIT, TR-242, MIT, Cambridge, MA, 1971.

12. P. Greussay, **Modeles de descriptions symboliques en analyse musicale**, M. Th. in Linguistics and Informatics, Université Paris VIII, Paris, 1973.

Appendix A: Music Objects BNFs

The following BNFs describe the syntax of MOs:

```
< MusicObject >         ::= < MusicObject > . < MetaEvent > |
                            < MusicObject > < MIDIEvent > |
                            . < MetaEvent > |
                            < MIDIEvent >
< MetaEvent >           ::= Tempo < integer > |
                            Base < integer > |
                            Signature < integer > < integer > |
                            Rest < integer > |
                            MidiEx < MIDIExBytes >
< MIDIExBytes >  ::= " ...any sequence of bytes, in hex, according to MIDI..."
< MIDIEvent >           ::=   < Header > Pgm   < integer > |
                            < Header > After   < integer > |
                            < Header > Polyp   < integer > |
                            < Header > Ctrl   < integer > < integer > |
                            < Header > Bend < integer > |
                            < Header > < NoteEvent >
< Header >       ::=  < Channel > < Quarter > < Offset >
< NoteEvent >           ::=   < Duration > < Pitch > < KeyVel >
< Pitch >               ::=   < Note+Mod > < Octave > | < KeyNumber >
< Note+Mod >            ::=   < Note > | < Note > #
< Note >                ::=    A | B | C | D | E | F | G
< Channel >             ::=   1 | ... | 16
< Quarter >      ::=  < positive integer > {...depends on song length}
< Offset >              ::=   < positive integer > {...depends on .Base}
< Duration >            ::=   < positive integer >
< KeyVel >       ::=  0 | ... | 127
< Octave >              ::=   -2 | ... | 8
< KeyNumber >           ::=    0 | ... | 127
```

Appendix B: Music Algorithms BNFs

The following BNFs describe the syntax of MAs:

```
< Algorithm >           ::= < Algorithm > < Operator > | < Operator >
< Operator >            ::= < QualOp1 > :   < RangeExpression > ,   < RangeExpression > |
                        < QualOp2 > :   < RangeExpression > , < RangeExpression > ,
                        < OpExpression > |
                        < QualOp3 > < TonalExpression > :   < RangeExpression >,
                        < RangeExpression > , < OpExpression > |
                        < QualOp3 > < TonalExpression > :   < RangeExpression >,
                        < RangeExpression > , [ < MusicObjectId > , < RangeExpression >],
                        < OpExpression > |
                        < QualOp4 > :   < RangeExpression > , < RangeExpression > ,
                        < OpExpression > |
                        < QualOp4 > :   < RangeExpression > , < RangeExpression > ,
                        [ < MusicObjectId > ,   < RangeExpression >] , < OpExpression >
< QualOp1 >             ::=   I | K | S
< QualOp2 >             ::=   M | R
< QualOp3 >             ::=   P
< QualOp4 >             ::=   C | D | L
<MusicObjectId >        ::= "... alphanumeric string ..."
< TonalExpression >     ::=   [ < Note > < Tonality > ] | [ < Note >]
< Tonality >            ::= m | - | t | p
< RangeExpression >     ::=   < RangeExpression > + < RangeTerm > |
                        < RangeExpression > - < RangeTerm > |
                        < RangeTerm >
< OpExpression >  ::=   < OpExpression > + < OpTerm > |
                        < OpExpression > - < OpTerm > |
                        < OpTerm >
< RangeTerm >           ::=   < RangeTerm > * < RangeFactor > |
                        < RangeTerm > / < RangeFactor > |
                        < RangeFactor >
< OpTerm >              ::=   < OpTerm > * < OpFactor > |
                        < OpTerm > / < OpFactor > |
                        < OpFactor >
< RangeFactor >         ::= < integer > |
                        < MetaChar >|
                        ( < RangeExpression > )
```

< OpFactor >	::= < Pitch > |
	< integer > |
	< MetaChar >|
	(< OpExpression > **)**
< MetaChar >	::= ***$*** | ***!*** | ***%*** | ***?***
< Pitch >	::= < Note+Mod > < Octave > | < KeyNumber >
< Note+Mod >	::= < Note > | < Note > **#**
< Note >	::= ***A*** | ***B*** | ***C*** | ***D*** | ***E*** | ***F*** | ***G***
< Octave >	::= ***-2*** | ... | ***8***
< KeyNumber >	::= ***0*** | ... | ***127***

The three integers that form the Header specify the MIDI channel to which the event and the instant when the event must take place are referred. The first stands for the MIDI channel. The second and the third show the moment connected with the *Base* and the *Tempo* chosen.

For example, if we have chosen *Base* 24, an Header equal to 1 2 12 shows that an event must be sent on channel 1 in the middle of the second quarter, namely after a quarter plus an eighth. In the example at the end of this paper, we give an example of a complete MO coded with respect to this syntax.

These are the choices made by ScoreSynth when the values of the *Range* go beyond the upper and lower limits: if the lower limit is less than 1 then it is set to 1; if the upper limit is greater than the maximum number of notes in the MO then it is set to the maximum value allowed.

Appendix C: Macro Nets BNFs

The BNFs listed below define the syntax of modifier lists:

< ModifierList >	::= < ModifierList > < Modifier> | <Modifier>
< Modifier>	::= < PlaceModifier > | < TransModifier > | < ArcModifier >
< PlaceModifier >	::= < PlaceId > ***:*** < PlaceOpList >
< PlaceOpList >	::= < PlaceOpList > < PlaceOp > | < Place Op >
< PlaceOp >	::= < PlaceAttribute > | < PlaceSetting >
< PlaceAttribute >	::= ***play*** | ***mute*** | ***file*** | ***notfile*** | ***obj*** | ***notobj*** | ***simple*** | ***subnet***
< PlaceSetting >	::= < NewId > |
	C **=** < Capacity > |
	M **=** < Token > |
	CH **=** < Channel > |
	{ < MusicObject > ***}***
< TransModifier >	::= < TransId > ***:*** < TransOpList >
< TransOpList >	::= < TransOpList > < TransOp > | < TransOp >
< TransOp >	::= < TransAttribute > | < TransSetting >
< TransAttribute >	::= ***enabled*** | ***disabled***
< TransSetting >	::= < NewId > |
	{ < Algorithm > ***}***
< ArcModifier >	::= < PTArcOp > | < TPArcOp >
< PTArcOp >	::= < PlaceId > **->** < TransId > **=** < Multiplicity >
< TPArcOp >	::= < TransId > **->** < PlaceId > **=** < Multiplicity >
< Multiplicity >	::= ***1*** | ... | ***240***

Appendix D: The Model Execution Algorithm

The algorithm listed below is the most abstract level of the PNs execution algorithm we have defined; it is written in Pascal-like code:

```
procedure Execute;
begin
   repeat
         Blocked := NOT ( Firings ); {see the Firings function below}
         if ... no music process is active...  then No_Active_Places := TRUE
         else
              begin
                    No_Active_Places:= FALSE;
                    compute DeltaTime for the processes (one or more) that terminate as the first;
                    execute all active processes for a DeltaTime number of time units;
                    delete all terminated processes from the list of active processes;
                    increase of DeltaTime units the virtual clock variable;
              end;
   until Blocked & No_Active_Places;
end;

function Firings: boolean;{It returns TRUE if at least one transition fires}
begin
   repeat
         create the list of transitions which may have simple firings;
         create the list of transitions which may fire in the frame
         of alternative and/or conflict net structures;
         if ... there exists, as the minimum, a transition which may fire ... then
              begin
                    Firings:= TRUE;
                    while ...there exist transitions which are in alternative and/or in conflict ... do
                         begin
                               generate a pseudorandom index with uniform distribution for
                               transitions which may fire in the frame of alternative and/or
                               conflict net structures;
                               execute the transition with respect to the generated index;
                               delete all the transitions which are no more in alternative and/or
                               in conflict from the proper list;
                         end;
                    execute all transitions which may have simple firings;
              end
         else
              Firings:= FALSE;
   until ...there is no transition which may fire ... ;
end;
```

For the sake of completeness, let's take a look to the procedure that executes a transition firing:

```
procedure ExecuteTrans(TransId);
                         {TransId is the identifier code of the transition to be executed}
begin
   if ...transition has an associated algorithm ... then execute the algorithm;
   decrease the marking of input places according with the firing rule and multiplicities of arcs;
   increase the marking of output places according with the firing rule and multiplicities of arcs;
   for all output places:
        if ...place has an associated MO ... then
             insert the associated MO in the list of active processes;
end;
```