A CONSTRAINT BASED LOGIC PROGRAM FOR GENERATING POLYPHONIES.

Francis COURTOT,
IRCAM, Department of Musical Research,
31 rue Saint-Merri, 75004 Paris, France.
e-mail: courtot@ircam.fr, use@ircam.fr/courtot.

ABSTRACT: This paper describes a program written in PrologII currently under development at IRCAM. The primary goal of this program is to provide a model which tries to represent the way a composer structures his knowledge, when confronted with a computer aided composition environment. A secondary goal is to provide a computer aided composition environment for Philippe Hurel, a French composer currently working on a commission from the Ensemble Intercontemporain.

1. FORMAL BASIS.

The program is organised according to three levels of representation. These are, in ascending order of complexity: types, methods, and concepts. We firstly present each of these levels, and then deal with an example taken from P. Hurel's application.

1.1 Types.

The first level of representation concerns what a composer begins to manipulate in the way of "interesting objects". For representing such objects, the first level in our system defines types. They include user defined or pre-defined types like pitches, chords, ... There are two different kind of types: the first ones, referred to as primitive, are the most simple. They represent very basic elements such as alteration, pitch-name and the like. The second kind of type are the composed types, constructed from others, either primitive or composed, like pitches, durations, intervals...

The way the types are structured internally relies on two distinct structures.

The first one is a tree, referred to as a signature, whose leaves are terms with four components, and whose nodes are type constructors. Its root is a maximal element, a fictive type called ANY. The leaves store four features of a primitive type, namely the domain (possible values for a term of this type), cutting (a characterisation of the intervals between each element of the domain), circularity (a boolean stating whether the last element of the domain is followed by the first or not), invertibility (each element has an inverse, or not). There are five type constructors, organised in 3 categories:

- list-like constructors: ho (horizontal, simultaneous collection of elements) and ve (vertical, successive collection of elements),
- taxonomic constructors: nup (composition of n types to form a new one), prod (same as nup, but with syntactic constraints on the new type),
- union: ou (simple union of types (a term belonging to any of the types composed),

The second way to structure the types is a conceptual graph. There are 5 type constructors, aimed at representing the relations between types. This graph is used to undertake the semantics of the language defined by the types above, and allows the system to automatically construct useful new types. The associating operators are the following:

- Type: associate two types and state that the later formalises the durations of the former.
- Der: associate two types and state that the later is a "codulation" of the former; this operator can be useful to represent the motion between a type formalising elements such as vibrato, trill, and the like, with pitches.
- Inc: state that a type is the one that calculates the intervals of another.
- Equal: allows polymorphic objects, that is multiple representation of the same musical entity.
- Sub-type: no particular relations between types

These type characteristics, constructors, and associators, as defined above, lead to the construction of a type base, accessible to the user who can modify it as well.

1.2 Methods.

The second level of representation defines methods (actually predicates) which can be applied to the objects defined with the types above. These methods can be classified in five categories, according to their behaviour.

The first one covers the object generation (using an algorithm designed by the user), referred to as a theory. The second kind of methods are constraints. Generally, the theory used by a composer is very general, that is, the instances generated by the theory are not all used by the composer when he works. We then use constraints in order to reduce a priori the amount of search for particular objects. These constraints are based on the types defined above, and on the third level of representation, that is concepts.

The third category of methods is the one that transforms objects into others, including "transmutations" between types associated with the Equal operator defined above, intervalic calculation, and the like. The
fourth category of methods include graphic editors and output as well. The last category groups miscellaneous methods, such as some routines, that allow the user to read or write in a file...  

1.3 Concepts.

The third level of representation actually makes the link between types, objects, and methods. We refer to this level as a conceptual level, because it is seen as a more general structure for musical objects. The user can define a set of characteristics (here called a concept) that enables a composer to examine an object from different perspectives. These characteristics are important, since they define the possibilities a composer has to establish the constraints. These constraints most likely represent a formal aspect of the piece he is working on. In our representation, a concept is a predicate whose arguments are all these characteristics, that is a structure that states the relationships between each characteristics. Actually, depending on the level of formalisation the composer deals with, a concept can be a single predicate or a program, such as the networks used by P. Hurel (see below).

1.4 An example.

For P. Hurel, the theory lies in a recursive structure of symbols called a pattern. The eight letters, _a b c d e f g_ form such a pattern. When repeated 5 times, the same structure (called the accent pattern and represented in upper case) reappears every 5 symbols: _A b c d e f g - a b c d e f g - A b c d e f g - a b c d e f g_. Figure 1 shows a part of the signature used for P. Hurel. In this figure, the greatest element has not been represented.

![Diagram of polyphony](image)

**Figure 1.** A part of the conceptual graph is represented in figure 2, where only have been represented the types related to another with an associate different from sub-type.

![Diagram of polyphony](image)

**Figure 2.**

The concept corresponding to a pattern must define the number of repetitions, the number of symbols used to obtain the accent pattern, and the accent pattern itself:

```
<table>
<thead>
<tr>
<th>pitches</th>
<th>Temp</th>
<th>symbolic_durations</th>
<th>Equal</th>
<th>numeric_durations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

II THE SYSTEM.

The system has a built-in interface that deals with two problems. The first one corresponds to the graphical editors. The second problem corresponds to the ability a composer has to write Prolog II programs.

II.1 The graphical interface.

This interface allows one to graphically construct Prolog programs simply by constructing a network with predicate boxes chosen from a dynamic menu. Here is an example of such a network, redefined by P. Hurel, and corresponding to a single voice for a polyphony.
Each box represents a predicate, chosen in the menu Predicates, in which the allowed predicates are classified according to the categories we have mentioned above. When the user clicks on the "?" part of the box, a window displaying commentaries is shown, stating the semantics of the predicate, and the meaning of every argument. The little boxes within a predicate box are arguments. Whether they are white or black, they have a value or not.

The connections represent equality constraints, that is unification. It is very easy to establish and disable such constraints: simply click in an argument box and drag the mouse to another. There is a type verification that prohibits incorrect unification.

In order to give a value to any argument, the user graphically selects the argument by double-clicking in it, and the system calls an appropriate editor. The first one is a general, alpha-numeric one. There are also a pitch editor, and a chord editor, as well as a duration editor. Furthermore, some editors specially conceived for P. Hurd are available too, like a pattern editor, numeric editors and the like. When no editor specific to the type of the argument selected is available, then the general editor is called.

It is possible to store the Prolog goal corresponding to the network, save the network itself, and run and interrupt the program thus constructed. The order in which the predicates are evaluated is done automatically by the system, according an order on the predicate categories mentioned above.

It is also very easy to store or recall a set of predicates, and to add new predicates. In this respect, it is also easy to edit predicates and transform them.

The graphical result of figure 3 is shown in figure 4.

II.2 Abstraction.

The easiest way to construct a new predicate is to make a network and abstract it. For such an operation, the user simply designates the new arguments, and the system generalises (simply turning these arguments into variables) the predicate definition. Then it adds it to the menu, with a name given by the user. The "?" part of a predicate box corresponding to an abstraction can display the network as it has been constructed by the user.

CONCLUSION

The model that is at hand in this project is designed to help the composer as much as possible in an automatic way. In this respect, the type formalisation of a composer's syntax is helped through a collection of dialogues and heuristics, that allows him to build a type base as well as to provide new types constructed by the system. The next step we want to experiment, is to include a powerful machine learning system aimed at the automatic programming of some predicates, from examples of their behaviour. In particular, the constraints a composer use are an interesting kind of predicate to be learned.

Using a system of this type, we hope to achieve a better understanding of musical formalisation.