An extendible tool for CAC

Hans Lunell
Roya College of Music
Stockholm  S-115 91 SWEDEN
E-mail: lunell@softsys.se

ABSTRACT: This paper describes an easily extendible tool for computer aided composition. It consists of a general flow-oriented surface language interfacing a data base of algorithmic components. Although the tool currently is built up to support computer aided composition on the macro level, it would be easy to adapt it to the sound processing level. It is implemented in C++ on Macintosh.

Introduction

Composers use very different compositional procedures in their work. A tool for composers hence must be flexible, extendible and moldable to suit the individual composer. Preferably the composer himself should be able to adapt the tool. But this has to be made as simple as possible, as the composer is not a programmer. This project grew out of such a situation. For a course in algorithmic composition a framework was needed in which the students could work and experiment with their own compositional models. Thus, the goal was not to implement a complete environment, rather to create a skeleton based on principles simple enough not to be constraining, but easily extendible.

The tool and its underlying model

The solution to the problem sketched above was to combine a simple interpreted surface language, a static data base of pre-compiled algorithms, a predefined set of data types and a dynamic data base of variables.

Data types

Basic data types are set, list and tuple, corresponding to the classes Set, List and Tuple. They all have integers as elements and are derived from the base class IntArray. They differ in behaviour in the following way. Lists are ordered with respect to the elements. There is a first, a next and a last element. They are possible to append, prepend, concatenate, etc. In contrast, tuples are ordered according to the element values. In this case there is a largest and smallest element, etc. Sets are unordered in both respects. They are just collections of integers without any duplicates. Type conversions reshape the values accordingly. For example, by converting a list into a set duplicates are removed. Another conversion into a tuple sorts the values in increasing order.

In addition, there are the data types integer and string, but they use of less interest from the principal point of view. Integers and strings only exist as constants in argument lists.

Variable data base

Variables are created as they appear in the surface language and introduced in a conventional way in a data base, or rather in symbol table. Variables are dynamically typed and either set, list or tuple.

Algorithm data base

Pre-compiled algorithms used to transform, generate, combine and make other operations on variables and constants constitute the data base. These are called by the interpreter in the normal C++ fashion. Even without any extensions the data base contains components to perform base procedures such as reading, writing, retrieving, storing and playing musical material (e.g., MIDI data), as well as typical algorithmic procedures on music material.

ICMC PROCEEDINGS 1995 353
The basic mechanism in the language is transformation of one or several sets, lists and/or tuples into one set, list or tuple. Syntactically, one or several variables or constants are fed into an algorithmic module, which generates a result. For example:

\[ d \rightarrow \text{reverse}(\cdot) \rightarrow \cdot, \]

would read the list \( d \), reverse it and place the result in \( \cdot \) (after having created \( \cdot \), if necessary). Another example is

\[ \text{generate}(200) \rightarrow \text{filter}(\cdot) \rightarrow \cdot \rightarrow \text{play}(\cdot), \]

which would generate 200 integers, feed them through a filter, store the result in the variable \( \cdot \) and play the result interpreting \( \cdot \) as a list of pitches.

Constants, lists, sets and tuples are recognized by the type of parentheses surrounding the values. For example, \( (12, 34, 56) \) is a set, \( [12, 34, 56] \) in a list and \( \{12, 34, 56\} \) is a tuple. Duplicates are neglected in a set and the order of the elements is irrelevant in a tuple. Hence, \( (12, 34, 56) \) and \( \{56, 34, 12\} \) are the same constants as above, respectively.

**Extension mechanism**

The capabilities of the tool are extended by adding new algorithmic modules. Such a module is simply a C++ function written according to certain rules. These are as follows: Arguments should be the classes Set, List and Tuple, besides the built-in types int and char*. All arguments but the last should be passed by const reference, as they are input parameters. The last argument must always be the result parameter, be passed by reference and have the type Set, List or Tuple.

To introduce the new function in the tool, it has to be compiled and linked to the rest of the algorithmic modules, but also inserted in the static algorithm data base to enable the interpreter to find and call the function. This is done by adding a C++ source line to the load module with information on the name of the function, the number and types of arguments and a function pointer.

**Conclusion**

The surface language is enough simple to be grasped very quickly, in less than one hour. At the same time it seems well adapted to the way many composers think and work (at least instrumental composers). It is flexible and easy to use for exploratory purposes.

The extension mechanism is also simple to grasp and once this is done, it is very easy to mold the tool to ones own needs. For uninitiated programmers (e.g., the composer starting to learn) the dividing-line between the flow-oriented surface and the functions helps to separate tasks into different levels of abstraction. The lack of control-flow in the surface language isolates control-flow problems to the functions and imposes a natural modularization principle on more complex problems.

It would be a simple matter to exchange the textual interface for a graphic, but the gain is questionable. This probably makes the interface between the tool and the algorithmic modules less transparent and the tool would consequently lose portability. It would probably also change the way it is used to more complex networks of transformation, but decrease the amount of experimental work.

**Acknowledgement**

The students contributed to the ideas of the project and the Board for Artistic Development financed it.