The Formes System: A Musical Application of Object-Oriented Concurrent Programming

Pierre Cointe
Jean-Pierre Briot
Bernard Serpette

This paper presents the FORMES system developed at IRCAM to deal with the complexity of musical representation and used to drive a general sound synthesizer. We first focus on the original concepts induced by musical applications. Object-Oriented Programming matches a subset of the numerous requirements of Computer Music Composition, but we need to extend this metaphor towards the time component in order to describe and control the temporal structures of music. Therefore we introduce the monitor concept which supports different schedulings of hierarchical processes, and extends the class concept to the field of concurrent (musical) structures. A short tutorial of the system describes its use through some musical examples. The FORMES virtual machine implemented in LISP is also presented.

Introduction

The FORMES system is a programming environment initiated at IRCAM L'Institut de Recherche et Coordination Acoustique/Musique is "conducted" by Pierre Boulez. four years ago. It aims at music composition and synthesis (MCS) by computer. The goal is to address the complexity of musical representation and to provide for resident or guest composers - some of them with little or no computer expertise - a powerful and friendly environment for music research and production. FORMES provides a musical framework modeling the results obtained in signal processing and synthesis research into sound material organized as blocks of knowledge called *processes*. These processes must be "easily" composed to elaborate the musical score that will be executed by various synthesizers (e.g., the CHANT program [Rodet et al. 1984], the MIDI interface or the 4X real-time processor [Koechlin et al. 1985]). The implementation of FORMES uses the techniques of object-oriented systems and develops them in a context of *concurrent musical programming*. To date, a number of composers have used it for their work (e.g., J. Harvey, P. Manoury, K. Saariaho, M. Stroppa, ...). As an example, CHREODE [Barrie‘re 1984], the first piece composed with FORMES by J. B. Barrie‘re, won the "International Bourges Festival" award in 1983.

First we wish to demonstrate how exciting it is to test object-oriented methodology in a context of musical research with the support of famous composers. Then we present some examples of FORMES scores and introduce the implementation choices used to build such a complex musical system on LISP foundations. More precisely: (1) we discuss the qualities of object-oriented programming for the musical domain, and we justify our choice for developing an object level upon a LISP kernel; (2) we introduce the major FORMES classes of objects, which allow a precise control of time and a simplified data-flow: we call them *process*, *monitor, clock,* and *calculation tree*; (3) we detail a tutorial including a meta definition of the GOD process, explaining the connection between the different kinds of objects and giving various examples of FORMES predefined musical structures; (4) we define the virtual machine which receives *message-passing* expressions and macro-expands them toward optimized LISP S-expressions operating upon the internal representation associated with each class of objects. This machine is built to be immediately installed on any dynamically scoped LISP, the standard IRCAM version being the Le_Lisp system [Chailloux et al. 1984] of INRIA; (5) we conclude with a criticism of some conceptual choices and present the future developments of the FORMES system.

Object-Oriented Metaphor Matches a Subset of MCS Requirements

Requirements of MCS

We will take here the initial requirements as expressed by X. Rodet at the beginning of the project (August 1981). In MCS, the aim of a musician is to capture some musical image. A MCS program is an attempt to find and realize this image - using a model that implements our knowledge about sound production and perception - within a particular compositional context. Consequently FORMES must provide a framework to manipulate and integrate these models as building blocks or objects. These models are characterized by their contribution to the synthesis and by their temporal behavior irrespective of implementation details and synthesis techniques. Desirable properties of MCS models include the following, all of which are goals of the FORMES system [Rodet and Cointe 1984]:

Apart from a specific application, a model should not be a portrait of a particular sound or note but should be a representation of a musical process as general as possible (e.g., a model of crescendo or an attack pattern should apply to as many differents sounds as possible). A model should try to be independent of a particular synthesis technique and should refer to universal concepts as found in acoustics and psychoacoustics. Models should be applicable in any context in which they are placed. In any combination they should gracefully cooperate and interact in the universe created by the composer. Models will be much simpler if their program texts follow common composer communication conventions and presuppositions. Uniform and clear symbolism should be used to create, modify, or integrate new models. The complexity of musical processes demands that they be built from subprocesses. With such a hierarchical construction it is also possible to integrate different specific behaviors into a new and more general one.

Object-Oriented Concepts Match a Part of MCS Requirements

We have demonstrated [Cointe 1984] that an object-oriented language can be analyzed through four conceptual patterns: the *object* paradigm, the *instantiation* protocol, the *inheritance* principle, and the *message-passing* mechanism. Another reading of the previous requirements establishes clearly the correspondence between a musical model and an active object, the uniformity wish and the instantiation protocol, the simplicity of program text and the message-passing mechanism, the modularity construct and the inheritance principle. Thus the object-oriented concepts can support the description of musical models organized in a knowledge representation system [Krasner 1980] [Lieberman 1982], but they don't provide a clear answer to the compositional point of view expressed by Pierre Boulez: "The relation of the sound object as such with the musical text, its malleability with regard to the order of events creating a form, remains for me the fundamental problem" [Boulez 1983].

In effect these models have to be composed into musical and temporal structures: thus we have to extend the object-oriented domain toward time component and *real-time* algorithms. This is the main contribution of the FORMES system.

FORMES Concepts or Real-Time Scheduling of Hierarchical Processes

Experimentation with the first FORMES prototype established the need for conceptual entities called **process**, **monitor**, **genealogical tree**, and **calculation tree**. Before we embark upon the description of these entities, let us outline their respective functions. Suppose we run a *process* that is to feed a sound synthesizer: at each quantum of time, the *monitor* of the process generates a binary tree (the *calculation tree*), which is then evaluated to update inputs (called the *registers*) to the synthesizer. The evolution of the sound signal reflects the time behavior of the process.

A First Example

Let us present a simple example of a process whose goal is to produce a second octave C for 2 seconds. l s l l. (dp A_NOTE
   monitor: leaf
   each-time: ((field-to-register ’(f1)))
   env: (duration 2. f1 (pitch ’C2))) To synthesize A_NOTE with the array processor, we use the message: l. (send ’A_NOTE ’ap) Then we create (a‘ la ACT-1 [Lieberman 1986b]) another note that differs only from the previous one by its pitch: l. (send ’A_NOTE ’new ’ANOTHER_NOTE ’env: ’(f1 (pitch ’A3))) To compose A_NOTE and ANOTHER_NOTE in a melody, we define a new process whose offspring is the suite of the notes; *seq-node* indicates their sequential scheduling: l l. (dp A_MELODY
   monitor: seq-node
   sons: (A_NOTE ANOTHER_NOTE A_NOTE)) To keep the evolution of the pitch (f1), to display it later on a screen or build a score, we define the register f1 as a buffer: l. (db f1) Then to display the pitch, after playing A_MELODY: l. (send ’f1 ’screen)

Informal Description of FORMES Concepts

Now we can express the postulates used to introduce the FORMES philosophy.

Monitor and Genealogical Tree

POSTULATE-1: Each **process** is created to control in time a set of subprocesses called its offspring. To realize a precise but not stereotyped control, FORMES connects each process with a **monitor** (by reference to [Hoare 1974]) scheduling the evolution of the sons’ processes (organized as a genealogical tree) in the temporal span of their father. Following P. Boulez, a process is the FORMES abstraction for the sound object, and a monitor the FORMES abstraction for the required "malleability" of musical events. To address the complexity of the musical world, several kinds of predefined monitors are available, each one implementing a privileged musical structure. For instance: the monitor **seq-node** juxtaposes in time a sequence of subprocesses; the monitor **trans-node** performs the overlapping transition of two active processes, allowing the simulation of consonants in the singing voice; the monitor **parallel-node** concurrently schedules several musical voices; the monitor **leaf** fixes the last level in the description of a hierarchical genealogy, i.e., a limit in the macroscopic definition of a musical structure; a note can be represented by a constant pitch (i.e., a *leaf* process) or with more details (i.e., a *seq-node* process) by a sequence of an "attack," a "sustain," and a "decay."

Thus a monitor is an object that defines a set of sheduling functions describing a musical structure. As such it may be shared by different processes and defines a class of temporal control structures. Consequently the pair <monitor , genealogical tree> defines a musical representation in the FORMES symbolism. The main problem is how to connect the musical abstraction with the computer abstraction. A first step in this way uses an iconic representation (much simpler than the musical notation) suggesting the monitor’s functions. The next figure gives the icon associated with the **seq-node** monitor:

If the monitor concept presents the compositional aspect of MCS, we need to build the sound material performing the synthesis. This material (derived from signal processing) is attached to each process by the definition of rules expressing the temporal *data-flow*.

Data-Flow and Calculation Tree

"A unique feature of FORMES is the inclusion of an explicitly temporal paradigm into the language. That is, all computation in FORMES is synchronized with a dynamic calculation tree that schedules all active objects at appropriate time points" [Roads 1984].

At each quantum of a clock, the set of monitors associated with the tree of active processes builds another tree - the **calculation tree** - obtained as a particular organization of those rules whose evaluation gives the data-flow feeding the virtual synthesizer. From a synthesis point of view, the calculation tree should periodically feed the inputs (or controls) of a synthesizer by a set of new values or commands. Thus, running a process should start a sound output. However, running a process could result in any other effect like screen display, patch loading, etc... For example, the rule (field-to-register '(f1)), used by the A_NOTE process, will affect In the model of data-flow we present in this paper, a register is a LISP global variable. We could use distinct register contexts for a multiphonic model with final mix. the register f1 (fundamental frequency input of the CHANT synthesizer) with the value of the corresponding field of the process (the value of (pitch 'C2) i.e., 130.81 Hz).

POSTULATE-2: Each process maintains a set of rules (generally expressed in FORMES, LISP or C). When the process is active, these rules define its contribution to the data-flow.

GENEALOGICAL TREE --[monitors , clock]--> CALCULATION TREE

Thus a calculation tree is an object generated by running a process; it is obtained through a set of rules received from each active subprocess of its offspring (genealogical tree). The major problem is to define explicitly the interconnection of the rules given by all active processes. The first idea is to connect these rules in concordance with the hierarchy defined by the *root process*. This means *father*'s rules, then *sons*'s rules (and recursively). We call this order **pre-order** and we use a special set of rules called **each-time:** (to mean they are evaluated at each quantum of the clock when the associated process is active). We use the binary relation "<" to represent an order in the rules evaluation; for instance: *process1* < *process2* means that *process1*'s rules are evaluated before In a LISP formalism: (progn (send '*process1* 'rules) (send '*process2* 'rules)) *process2*'s rules. In fact with this order it is not possible to represent a various number of musical data-flows, so we have introduced another order called **post-order**, which uses the **each-time*:** field to connect associated rules in a reverse order: *son*'s rules preceding *father*'s rule. At the present time the calculation tree links together both kinds of rules.

Buffer: Register with Memory

A **buffer** is a memory register that keeps the evolution of its value in time. If the register f1 is defined as a *buffer* (db f1), and the message (send *a_process* 'run&bufferize) used to activate the process, the values transiting in the value-cell of the LISP variable f1 will be bufferized. (send 'f1 'screen) will then display the evolution of f1 on any alphanumeric terminal and (send 'f1 'vpr) will print it on a graphic device. The figures presented in this paper were done in that way.

Uniformity

POSTULATE-3: Each entity of the FORMES system is represented as an object. Each object belongs to a class, and to each class is associated a LISP generator: **dp** to **d**efine a **p**rocess, **dmo** a **mo**nitor, etc...

Differential Instantiation

POSTULATE-4: Each object of the system is instantiable (in the ACT-1 way [Lieberman 1986a] [Lieberman 1986b]), this instantiation using a differential method. The term *differential* [Barthes 1953] means that only the difference between the model and its instance must be explicit. No difference means a simple copy of the model. Consequently, FORMES provides two levels for creating an object, the LISP level with the generators of the virtual machine and the object level with the *new* message (as we saw in the first example). Note that a FORMES class is a bit different from SMALLTALK explicit classes [Goldberg and Robson 1983]. In FORMES, a class is rather *implicit* through the LISP generators of different types (classes) of objects, but the methods of a class of objects are also shared (cf §6.3.3.). An approach of FORMES through the class semantics, plus a micro-interpreter implementing this model, are presented in [Briot 1984].

Mapping FORMES Concepts on Object-Oriented Formalism

We give the "syntactic sugar" supporting the definition of the different classes of objects used later in this chapter. This syntax results from a comparison between the musical concepts previously presented and the traditional object-oriented representation. Traditionally, an object is defined [Birtwistle et al. 1973] [Goldberg and Kay 1976] [Hewitt and Smith 1975] [Steele and Sussman 1975] by a local environment (fields) and a set of functions (selectors+methods):

OBJECT = <FIELDS , METHODS-DICTIONARY>
ACTOR = <ACQUAINTANCES , SCRIPT>

PROCESS as an Object

The mapping of the musical idea of a process:

FORMES-PROCESS = <SYNTHESIS-RULES , MONITOR , SUBPROCESSES>

with the classical definition of an object (as above) gives the following association:

PROCESS = <P-FIELDS , P-SCRIPT>

P-SCRIPT: defines the methods (i.e., the functionalities shared by all processes); for example, the protocol of activation (*run*), the protocol of instantiation (*new*), the access to its environment (*?* & *?*<-), etc...

P-FIELDS: groups together six sets of fields:

The name of the monitor associated with the process and denoted by the field *monitor:* The description of the offspring of the process, defined as a binary tree - i.e., a list - and denoted by the field *sons:* The set of rules defining the behavior of the process for the synthesis and defining its contribution to the building of the calculation tree. These fields are denoted by the key-words *first-time:*, *last-time:*, *each-time:*, *each-time**: The fields used by the system to connect a process with the clock, the genealogical tree, and the calculation tree. They include the beginning time of the process, its duration (if possible), the address of the rules of the process in the calculation tree, and the active genealogical tree defined as the set of all active subprocesses A stop condition, which will be checked if specified in the field *exit:* The fields defined by the user with the keyword *env:* and used to maintain private information associated with the process

The first definition of a process uses the **dp** (*d*efine *p*rocess) primitive of the virtual machine. The next figure explains its syntax: center box; l s l l. (dp *process_name* *; offspring & monitor* sons: (*process_name1* ...) monitor: *monitor_name* *; rules* first-time: *implicit_progn*1 each-time: *implicit_progn*2 each-time*: *implicit_progn*3 last-time: *implicit_progn*4 *; stop condition* exit: *S_expression* *; environment* env: *init_plist1*)

Note that default values are assumed if some fields are not specified, as we saw above in our first example. We could even define a minimal leaf-process named *foo* as (dp foo) and run it.

MONITOR as an Object

A FORMES monitor is a scheduler operating upon immediate sons processes and synchronizing them together, following the description of the genealogical tree. Thus a monitor maintains the definition of a temporal control structure generally mapped to a musical structure. As an object, a monitor is defined by the pair:

MONITOR = <M-FIELDS , M-SCRIPT>

M-SCRIPT: denotes a set of methods associated with the class MONITOR. This set is returned by the monitor itself when receiving the message *selectors*.

M-FIELDS: maintains the four tasks associated with a scheduler; each of them is defined by a LISP function:

M-FIELDS = {*init: end: duration: offspring:*}

defines the start time for each son process - field: *btime* -; it also evaluates the *first-time:* rules; manages the synchronization of sons processes. Defining an event as a modification of the state of a process implies that the monitor has to rebuild the calculation tree when a new event appears; expresses - if possible - the span of the root-process (- etime btime). Different models of duration algorithms are available; for instance, the span of a sequential process can be defined as the sum of its sons' spans, or in contrast, by scaling its sons so that their sum equals its explicit duration; realizes the translation between the LISP definition of the hierarchical structure - a binary tree - and an internal representation used by the monitor to schedule in time the genealogical tree.

The primitive **dmo** is used to declare a new-monitor. It expects the following syntax: center box; l s l l. (dmo *monitor_name* init: *function_name* end: *function_name* duration: *function_name* offspring: *function_name*)

For instance the object **seq-node** is created by the definition: center box; l s l l. (dmo **seq-node** init: initn end: end-obn? duration: duren offspring: identity)

As a monitor, it recognizes the following set of messages:

(send 'seq-node 'selectors) $\Rightarrow$ (print end: init: duration: new offspring:)

CALCULATION TREE as an Object

TREE = <(*first* , *last*) , T-METHODS>

As an object, the calculation tree uses two fields to point the first and the last cons-cells of the "implicit progn" list connecting the *each-time:* rules of active processes. It recognizes the message *init* to allocate the first cons-cell, *insert* to receive rules, *delete* to suppress rules, *eval* to execute the rules and provide the data-flow, *draw* to draw an iconic sketch of itself. The primitive **dt** is used to declare a new calculation tree: center box; l l. (dt calculation tree) *; define tree*

CLOCK and BUFFER as Objects

CLOCK = <*(quantum , time)* , C-METHODS>

BUFFER = <(*last_value* , *all_values*) , B-METHODS>

As an object, a clock uses two fields, the *quantum* and the *time*. It recognizes the messages *init*, *next-tick*, and all of those returned by the message *selectors*. As an object, a buffer uses two fields, *last_value* denoting the current value of the LISP variable, and *all_values* a cons-cell whose cdr defines a list of times, and car the successive-values associated with the variable. Here is the syntax of **dc** and **db** generators: center box; l l. (dc clock) *; define clock*

(db f1) *; define buffer*

A FORMES Tutorial

We have built this tutorial in order to emphasize the *monitor* concept. Consequently we present successive examples using standard monitors to conclude with the definition of a new one using the *differential instantiation* concept.

Some FORMES Features

Before presenting and commenting on the set of FORMES examples defining this tutorial, we have to define the options chosen to specify the object-oriented level.

Message-Passing

The syntax of a transmission is a generalization of the LISP "funcall form": center box; l. (send *object selector* Arg_1 ... Arg_n)

The pair <*object* , *selector*> allows the calculation of a LISP function that is applied to the arguments Arg_i. Notice that all the arguments of the send function are evaluated, including the selector and the object.

Access to the Fields of an Object

In the context of a transmission, the fields of the object receiver are not bound to their values. This choice, imposed by the the necessity of optimizing an interpreter driving synthesizers, In musical synthesis, e.g., with the CHANT synthesizer, one could need about 10 parameters per formant, and 30 formants for very rich sounds (e.g., bell or cymbal sounds); therefore, more than 300 fields could be needed for a simple process (and several such processes may occur concurrently!). Binding implicitly this environment at every activation of the process (at every tick of the clock) would be too heavy and would slow down the interpreter. requires an explicit access to the value of a field ("a‘ la LOOPS" [Bobrow and Stefik 1983]). Then to read/write a field of a given object, we use two special transmissions: center box; l. (send *object* ’? *field*)

(send *object* ’?<- *field new-value*)

The selectors ? and ?<-, respectively, denote a "get-value" and a "put-value" function of the virtual machine.

The Pseudo-Process **fself**

FORMES uses two pseudo-objects:

the object **oself** denotes - in the scope of a transmission - the name of the current receiver; the process **fself** denotes - in the scope of process' rules (i.e., the rules *first-time:*, *each-time:*, *each-time*:*, and *last-time:*) - the name of the process itself.

fself allows us to write anonymous rules that can be shared by several objects.

Explicit access to one field can be simplified by using the LISP level: center box; l l. (send fself '? *field*) ⇒ #@*field*

(send fself '?<- '*field new-value*) ⇒ (fsetq *field new-value*)

#@ is a Le_Lisp sharp macro character, which expands into the "get-value" call of the virtual machine.

The tnorm (and 1-tnorm) Primitives

Each process maintains in its private environment two fields named *btime* and *duration*, initialized (by the monitor's function *init:*) at its activation, and respectively bound to the value of the clock at that starting-time and to the value of its potential span (calculated by the monitor's function *duration:*). If the duration is foreseen, it is possible to apply a function called **tnorm** (like *t*ime *norm*alized), which means that its value is proportional to time but goes from 0 to 1 during the span of the process received as argument; In fact, default argument of tnorm is *fself*, so we could also write (tnorm) rather than (tnorm fself). **tnorm** seems to fill the same needs as the *prototype* concept described in ARTIC [Dannenberg 1984]. The LISP definition of the **tnorm** function uses the global variable **time** denoting the current value of the clock: center box; l. (defmacro **tnorm** (process)

```
  ; tnorm(time) = (time - start-time) / duration
  `(divide (- time (send ,process '? 'btime)) (send ,process '? 'duration)))
```

Consequently, if we define the **RAMP** process as center doublebox; l s l l. (dp **RAMP**

monitor:	**leaf**	each-time: ((setq output1 (* 0.2 (**tnorm** fself))))
env:	(duration 0.1))	

at each quantum of the clock, the rule (setq output1 (* 0.2 (tnorm fself))) will be evaluated; then the value of variable output1 runs over the interval [0. 0.2] during 0.1 seconds when **RAMP** is activated.

Introduction to the Leaf Monitor

To present the simplest monitor, we chose to define the process **FIB** which generates the terms of the "FIBONACCI's suite".

Writing Fibonacci with FORMES

The definition of **FIB** uses a **leaf** monitor: this process needs no offspring. **FIB** maintains in its private environment the standard field *duration* (bound to the value 10), and three other "dummy" (e.g., bound to the "?" value) fields: *fib*$_n$, *fib*$_{n-1}$ and *aux*. center doublebox; l s l l. (dp **FIB** monitor: **leaf** first-time: ((fsetq fib$_{n-1}$ 1 fib$_n$ 1) (send 'clock '?<- 'time 1) (send 'clock '?<- 'quantum 1)) each-time: ((fsetq aux #@fib$_n$) (fsetq fib$_n$ (+

#@fib_n #@fib_{n-1})) (fsetq fib_{n-1} #@aux)) last-time: ((print #@fib_n))
env: (duration 10 fib_n '? fib_{n-1} '? aux '?))

The rule *first-time:* describes the initialization executed at each activation of the process: *fib_{n-1}* and *fib_n* receive the two first terms of the suite (fib_0 = 1 and fib_1 = 1), time (argument of the suite) is set to 1, and the quantum of the clock is set to 1 The rule *each-time:* is the classical imperative definition of the Fibonacci algorithm.

Sending the message *run* to the **FIB** process activates the "PL/1-like" following loop: center box; lI l. first-time: fib_{n-1} <- 1 ; fib_n <- 1 ; quantum <- 1 ; duration <- 10 for time = 2 by quantum to duration begin each-time: aux <- fib_n ; fib_n <- fib_n + fib_{n-1} ; fib_{n-1} <- aux end last-time: print fib_n

Then (send '**FIB** 'run) computes (fib 10) and prints the value 89. We demonstrate later (cf. §5.) that the activation of a process is controlled by a context of execution self-defined by the immanent process **GOD**. This process schedules **FIB** and supports the executive control of the previous loop.

Definition of a Trajectory for the Fundamental Frequency

This second musical example illustrates the methodology used by J. B. Barrie‘re to elaborate his piece CHREODE. The starting point was to fix a basic trajectory built on the idea of a damped sinusoid represented by this figure:

Figure 1. A damped sinusoid

This trajectory must be controlled by four parameters: a direction of sweeping, a beginning phase (ϕ), a number of periods (ω), the damping of the exponential (α).

Here is the mathematical function of time describing this trajectory:

$$\sin(2\pi\omega x - \phi) * (1 - x) \text{ sup } \alpha \qquad \text{for } x \in [0 \;\; 1]$$

and here is the LISP translation: center box; l. (de trajectory (process direction α ϕ ω)
(let ((x (if direction (1-tnorm process) (tnorm process))))
(*
(power (- 1 x) α)
(sin (- (* (* 2π x) ω) ϕ))))

The *tnorm* function allows us to control the evolution of such a trajectory; *tnorm* is coupled with a process owning fields: *direction*, α, ϕ and ω. Then we define the **RED** process, binding the parameter f1 to the given trajectory for a duration of 10 seconds: center doublebox; l s l l.

(dp **RED** monitor: **leaf** first-time: ((send ’clock ’?<- ’quantum 0.1)) each-time: ((setq f1 (trajectory fself #@direction #@α #@ϕ #@ω))) env: (duration 10 direction t α 4. ϕ π/2 ω 7.))

At each tick of the clock, the parameter f1 (fundamental frequency) receives a new value calculated on the RED trajectory.

Instantiation or Differential Perspective

Each FORMES object is a potential generator recognizing the *new* selector. The instantiation mechanism is differential, because the creation of a new object may be defined as the expression of the differences between the model and its instance. When the selector *new* is used without arguments, there is no difference between the two objects other than their names, and the instantiation is just a copy. From a musical point of view, the **RED** process defines a "form" from which it is possible to derive new ones. As an example we use three symmetries to derive three other trajectories controlled by three new processes that we call **GREEN**, **YELLOW**, and **ORANGE**: center doublebox; l. (send ’**RED** ’new ’**GREEN** ’env: ’(direction nil)) (send ’**GREEN** ’new ’**YELLOW** ’env: ’(ϕ -π/2)) (send ’**YELLOW** ’new ’**ORANGE** ’env: ’(direction t))

Seq-node Monitor

Suppose we want to describe a note process as the embedded composition of three subprocesses, successively: an attack, a sustain, and a decay. We use the **seq-node** monitor to simply juxtapose in time the **ATTACK**, **SUSTAIN**, and **DECAY** processes. center doublebox; l s. (db output1)

(defmacro *= (parameter step) ‘(setq ,parameter (* ,parameter ,step)))

l s l l. (dp **NOTE** sons: **(ATTACK SUSTAIN DECAY)** monitor: **seq-node** each-time: ((setq output1 #@amp)) env: (amp 1.))

l s l l. (dp **ATTACK** monitor: leaf each-time: ((*= output1 (tnorm))) env: (duration 0.1))

l s l l. (dp **SUSTAIN** monitor: leaf env: (duration 0.3))

l s l l. (dp **DECAY** monitor: leaf each-time: ((*= output1 (1-tnorm))) env: (duration 0.15))

Figure 2. A model of NOTE

Seq-node Scheduling

The monitor **seq-node** implements a sequence of processes. It activates the subprocesses in the left-right order given by the *sons:* (genealogical tree):

Figure 3. A 3-component envelope

Seq-node Constraints

Using the monitor **seq-node** with the process NOTE's offspring means the verification of these equalities: center box; l. T1 = (send 'NOTE '? 'btime) = (send 'ATTACK '? 'btime) T2 = (send 'ATTACK '? 'etime) = (send 'SUSTAIN '? 'btime) T3 = (send 'SUSTAIN '? 'etime) = (send 'DECAY '? 'btime) T4 = (send 'DECAY '? 'etime) = (send 'NOTE '? 'etime)

Notice that the duration of the NOTE process is not given explicitly by the field *duration*. The *seq-node* monitor's *duration:* function calculates this value by adding the durations of the sub-processes: center; c. (send 'NOTE '? 'duration) = Σ (send 'ATTACK '? 'duration) (send 'SUSTAIN '? 'duration) (send 'DECAY '? 'duration)

Seq-node Data-Flow

When running NOTE, the calculation tree is rebuilt at times T1, T2, and T3. We give the three states of this tree during the span of NOTE: center box; l l. t ∈ [T1 T2] (setq output1 #@amp) < (*= output1 (tnorm)) t ∈ [T2 T3] (setq output1 #@amp) < () t ∈ [T3 T4] (setq output1 #@amp) < (*= output1 (1-tnorm))

Remarks:

1. During the interval [T2 T3], NOTE and ATTACK's rules are present in the calculation tree, but ATTACK does not contribute to the data-flow (only NOTE does) because its rules are empty.

2. The call of the following function NOTE_data-flow could provide the same data-flow that the message *run* sent to the process NOTE: center; l. (send 'NOTE 'run) ≡ (NOTE_data-flow (send 'ATTACK '? 'duration) (send 'SUSTAIN '? 'duration) (send 'DECAY '? 'duration) (send 'NOTE '? 'amp)) center box; l. (defmacro tnorm_data-flow (btime duration) '(/ (- time ,btime) ,duration)) (defmacro 1-tnorm_data-flow (etime duration) '(/ (- ,etime time) ,duration))

(de NOTE_data-flow (Ad Sd Dd amp)
(let ((T1 0) (T2 Ad) (T3 (+ Ad Sd)) (T4 (+ Ad Sd Dd))) (for (time T1 quantum T4) (cond ((< time T2) (setq output1 amp) (*= output1 (tnorm_data-flow T0 Ad))) ((< time T3) (setq output1 amp)) ((< time T4) (setq output1 amp) (*= output1 (1-tnorm_data-flow T3 Dd))))))

This construction, in contrast to the FORMES' one, is not modular and cannot be used in another context. The difference between the two formalisms expresses the gap between a FORMES program and traditional programs written with musical languages derived from MUSIC-5.

A New Level in the Musical Hierarchy: The SUITE Process

To give a significant example of calculation tree evaluation, we keep the same scheme of sequential scheduling, but we add a level of hierarchy. The process **SUITE** defines a sequence of three notes: **PIANISSIMO**, **MEZZOFORTE**, and **FORTISSIMO**: center doublebox; l s l l. (dp **SUITE** monitor: **seq-node** sons: (**PIANISSIMO MEZZOFORTE FORTISSIMO**))

Having established the NOTE model, we instantiate it three times to derive a PIANISSIMO, a MEZZOFORTE, and a FORTISSIMO. The arguments of the *new* message express the differences with the receiver; consequently NOTE, PIANISSIMO, MEZZOFORTE, and FOR-

TISSIMO differ only by their respective *amp* values: center doublebox; l l. (send '**NOTE** 'new '**PIANISSIMO** 'env: '(amp 0.04)) (send '**NOTE** 'new '**MEZZOFORTE** 'env: '(amp 0.2)) (send '**NOTE** 'new '**FORTISSIMO** 'env: '(amp 1.))

The evolution of the genealogical tree associated with an activation of the SUITE process is expressed by the new icon:

When running SUITE, nine time intervals have to be considered, each of which defines a new configuration for the calculation tree: expand box; l l. SUITE < PIANISSIMO < ATTACK time ∈ [0. 0.1] SUITE < PIANISSIMO < SUSTAIN time ∈ [0.1 0.4] SUITE < PIANISSIMO < DECAY time ∈ [0.4 0.55] SUITE < MEZZOFORTE < ATTACK time ∈ [0.55 0.65] .. .. SUITE < FORTISSIMO < DECAY time ∈ [1.5 1.65] For each note, the S-expression (setq output1 #@amp) initializes the variable *output1* to the value owned by the field *amp* (0.04 for PIANISSIMO, 0.2 for MEZZOFORTE...). This value is modified by the ATTACK and the DECAY processes but left constant by the SUSTAIN process:

Figure 4. Running SUITE generates a sequence of envelopes

Notice that the processes PIANISSIMO, MEZZOFORTE, and FORTISSIMO share the same sons: ATTACK, SUSTAIN, and DECAY. If we assign distinct durations to PIANISSIMO, MEZZOFORTE, and FORTISSIMO, they will be dynamically scaled in the sons' durations, by the function *duration:* of the *seq-node* monitor, to reflect these successive constraints from their successive active fathers.

Parallel-node Monitor

The **parallel-node** monitor is used to manage several voices in parallel. Each voice begins at the same time and is scheduled as a sequence. This monitor allows regroupment of processes not hierarchically related; this is important for many aspects of musical structure, for example, when different voices are not *synchronized* [Serpette 1984].

A Pattern Matching for a Musical Sieve

The process **2-VOICES** defines a musical sieve. To study a more sophisticated example, [Cointe 1984] comments on a complex sieve realized for the 4X processor by P. Manoury. It uses two voices for pattern matching: the first one is the pattern, the second the data. The sieve principle is to annul the process **NOTE** of the second voice when the pattern-process **no-NOTE** is simultaneously present in the first one. This given sieve operates on the fundamental frequency. The **C** (**E**) process sets the pitch to the value 100. (130.) unless the **no-C** (**no-E**) process is running in the pattern's voice. Independently, **2-VOICES** updates f1 to the value 140. at every quantum: center doublebox; l s s. (db f1)

l s s l l s. (dp **2-VOICES** monitor: **parallel-node** sons: ((**no-C** **no-E** **no-C**) (**E** **E** **C**)) each-time: ((field-to-register '(f1))) env: (f1 140.))

; patterns l s s l l s. (dp **no-C** monitor: leaf env: (duration 0.3))

l s s l l s. (send '**no-C** 'new '**no-E**)

; data l s s l l s. (dp **C** monitor: leaf each-time: ((*unless-sieve* (field-to-register '(f1)))) env: (duration 0.2 f1 100.))

l s s l l s. (send '**C** 'new '**E** 'env: '(f1 130.))

(field-to-register '(f1)) is equivalent to (setq f1 #@f1)

The function *unless-sieve* is a conditional progn. Its arguments are evaluated sequentially if the active process of the second voice is not time-matched by a **no-**process in the first voice: center box; l. (de dynamic-data () (concat 'no- fself)) (de dynamic-pattern () (caar (send #@father '? 'active-sons)))

(defmacro *unless-sieve* progn
  `(unless (eq (dynamic-data) (dynamic-pattern)) ,.progn))

Notice that each *parallel-node* process maintains in its field *active-sons* a pointer to the instantaneous genealogical tree.

Parallel-node Scheduling

Parallel-node Constraints

Each voice starts at the same time (T0) as the **2-VOICES** process **2-VOICES** finishes with the last process (no-C) of the first voice, i.e., at time T5 **2-VOICES**' duration is calculated as the duration of its longer voice (T5 - T0).

Parallel-node Data-Flow

Five intervals have to be considered, and the sieve does not operate during the [T2 T3] time when **no-E** matches **E**. The next figure expresses the *pre-order* data-flow associated with the activation of **2-VOICES** (the *post-order* data-flow is "empty" because no process of the hierarchy uses a *each-time**: rule): center box; l l. **2-VOICES** < no-$C_{(1)}$ < $E_{(1)}$ time ∈ [T0 T1] **2-VOICES** < no-$C_{(1)}$ < $E_{(2)}$ time ∈ [T1 T2] **2-VOICES** < no-E < $E_{(2)}$ time ∈ [T2 T3] **2-VOICES** < no-E < C time ∈ [T3 T4] **2-VOICES** < no-$C_{(2)}$ time ∈ [T4 T5]

FIB as a Parallel-node

We conclude the presentation of the parallel-node monitor by a new definition of the process **FIB**. This time, **FIB** activates two subprocesses built on the same model and calculating concurrently the terms (fib (- n 1)) and (fib (- n 2)): center doublebox; l s l l. (dp **FIB1** monitor: **leaf** first-time: ((fsetq fib_{n-1} 1 fib_n 2)) each-time: ((fsetq aux #@fib_n) (fsetq fib_n (+ #@fib_n #@fib_{n-1})) (fsetq fib_{n-1} #@aux)) env: (duration ∞ fib_n '? fib_{n-1} '? aux '?))

l s. (send '**FIB1** 'new '**FIB2** 'first-time: '((fsetq fib_{n-1} 1 fib_n 1)))

l l. (dp **FIB** monitor: **parallel-node** sons: ((**FIB1**) (**FIB2**)) first-time: ((send 'clock '?<- 'time 3) (send 'clock '?<- 'quantum 1)) last-time: ((print (+ (send 'FIB1 '? 'fib_n) (send 'FIB2 '? 'fib_n)))) env: (duration 10))

Circ-node Monitor

We detail this last monitor to complete the FORMES methodology and to achieve the description of one part of the Barrie‘re´s piece called CHREODE.

Instantiation of a Monitor

The idea of building a complex object by derivation from another one is applied to the monitor construct. In fact this choice means an extension of the *differential principle* to the domain of time´s musical control structure. The intuitive idea of the monitor **circ-node** is to express a rythmic pattern by repetition of the same basic sequence of subprocesses. The name of this monitor is a "LISP one" and reflects the subjacent principle of *circ*ularizing the offspring of a **seq-node** process: center doublebox; l. (send '**seq-node** 'new '**circ-node** 'offspring: '*offspring-circ*)

```
(de offspring-circ (offspring)
   (if (circlist? offspring) offspring
      (nconc offspring offspring)))
```

The only difference between a **seq-node** monitor and a **circ-node** monitor resides in the associated *offspring:* function. The function *offspring-circ* receives as argument the list *offspring* pointing out the genealogical tree and makes it circular by calling the *nconc* function (in contrast, the *offspring:* function associated to the *seq-node* monitor is *identity*).

CHREODE or Color Compositions

"At a higher level in the hierarchy, another FORMES object is charged with jumping from one curve to another at each quantum of time. Each curve has a phase and a number of periods which are either slightly different or radically opposed. This gives a very subtle and complex interplay of phase displacement and delay, successively within individual lines, and simultaneously between them all" [Barrie‘re 1984].

We have already exposed the definition of the four primary trajectories (cf. §4.2.2., §4.2.3.), each of them indexed by a color. Now we have to specify a more complex trajectory built as a "circular permutation" of the primary ones. The musical idea is to provide each quantum with a modification of the fundamental trajectory to constrain it to receive its value alternatively from each primary trajectory defined by the set:

<RED , GREEN , YELLOW , ORANGE>

This icon expresses the dynamic genealogical tree associated with the desired musical structure:

The process **RGYO** coupled with the monitor **circ-node** schedules the repetitive sequence RED to GREEN, GREEN to YELLOW, YELLOW to ORANGE, ORANGE to RED, RED to GREEN, and so on... Here is the complete FORMES program realizing the rocking trajectory: expand doublebox; l s l l. (dp **RGYO** monitor: **circ-node** sons: (RED GREEN YELLOW ORANGE) first-time: ((send ’clock ’?<- ’quantum 0.1)) exit: (> time #@etime) env: (duration 60.))

l s l l. (dp **RED** monitor: **leaf** each-time: ((setq f1 (trajectory #@father #@direction #@α #@ϕ #@ω)) (send fself ’suspend)) env: (direction t α 4. ϕ $\pi/2$ ω 7.))

l s. (send ’**RED** ’new ’**GREEN** ’env: ’(direction nil ω 15.)) (send ’**GREEN** ’new ’**YELLOW** ’env: ’(ϕ -$\pi/2$ ω 30.)) (send ’**YELLOW** ’new ’**ORANGE** ’env: ’(direction t ω 60.))

RGYO schedules the change of trajectory by giving alternatively the control to each of its sons. Because the associated genealogical tree is circular, this process cannot finish with its last son, so we have to designate the **RGYO** termination by the *exit:* stop condition. The **RGYO** end occurs when the clock outstrips its *etime*, i.e., when (> time #@etime) is true. The duration of the **RED** process is mapped with the quantum to associate the duration "T_{i+1}-T_i" with the duration between two ticks of the clock. The message *suspend* allows the suspension of the receiver process at its first (and last) tick. **GREEN**, **YELLOW**, and **ORANGE** are *differentially* derived from the **RED** definition.

The following figure gives the temporal evolution of the *f1* register after an activation of **RGYO**:

Figure 5. CHREODE, fundamental frequency: f1 (time)

GOD Process or a First Step Toward a Meta-FORMES

This section presents the activation protocol of a process **P** and explains the behavior of the only predefined process: **GOD**. When **P** receives the message *run*, the #**:object:process:run** function is applied (the *run* function is interned in the #*:object:process* package): center box; l. (send '**P** 'run) ⇒ (#:object:process:run '**P**)

(de #**:object:process:run** (oself)
 -1- (send '**GOD** 'sons: (cons oself ()))
 -2- (until (send '**GOD** 'end? 0 '#:object:process:run)
 -3- (send 'calculation-tree 'eval)))

We can observe (-1-) that **P** becomes the only son of **GOD** which controls By the way, the **seq-GOD** monitor is defined as:

(send '**seq-node** 'new '**seq-GOD** 'duration '(λ (env) 0))

its activation and builds the following data-flow:

GOD < **P** < **P**-offspring < ... < **P**-offspring* < **P*** < **GOD***

This data-flow is executed (-3-) until (-2-) **GOD** (i.e., **P**) finishes. Here is the definition of the **GOD** process that explains the activation's context of any process: center doublebox; l s l l. (dp **GOD** sons: (to-run) monitor: **seq-GOD** first-time: (-1.1- (send 'clock 'init 0.01) -1.2- (send 'calculation-tree 'init) -1.3- (mapc (λ (buffer) (send buffer 'init)) db)) each-time*: (-2.1- (send 'calculation-tree 'draw) -2.2- (mapc (λ (buffer) (send buffer 'save)) db) -2.3- (send 'clock 'next-tick)) last-time: (-3.1- (write-score 'chant)) env: ())

Three kinds of rules appear:

These rules are executed before the activation of **GOD** (and **P**). They provide the initialization of the clock: its quantum is set to 0.01 second (-1.1-), the initialization of the calculation tree (-1.2-) and of each buffer included in the set *db* (-1.3-) At each tick of the clock - when **GOD** running - these rules are the last executed by the data-flow: the calculation tree is drawn (-2.1-), the current value of any buffer is bufferized (-2.2-), and finally the clock receives the message to produce a new tick (-2.3-) **GOD** (and **P**) finishing, a score is built (-3.1-) for the CHANT synthesizer. In the context of a score-generation for a synthesizer (here CHANT). This is different for a real-time piped synthesizer (e.g., on an array processor), where data-flow is sent to it at every quantum.

Obviously such a FORMES definition of the running context means

complete uniformity and transparency; complete extensibility, because this context may be modified by every user (in fact we use a particular **GOD** for each kind of synthesizer); a more precise understanding of the interpreter architecture.

The FORMES Virtual Machine

Object-Oriented System as an Extension of a LISP Kernel

"While this does not invalidate the conjecture, it at least demonstrates that computer music is an excellent testing ground for the extensibility of general-purpose languages. And indeed, it has been the tendency of music language designers to extend established languages instead of inventing new ones out of whole cloth" [Loy and Abott 1983].

Having decided to experiment with the object-oriented methodology the implementation of our musical system, we had the choice between using an object system "a‘ la SMALLTALK" [Goldberg and Robson 1983] or an extension of LISP "a‘ la FLAVORS" [Moon and Weinreb 1980]. Our experience in SMALLTALK-76 implementation [Cointe 1983] has convinced us - from an implementation point of view - to follow the spirit of LISP rather than the SMALLTALK approach, which we consider a bit too heavy (and difficult to evolve [Briot 1985]) to elaborate an original system. Consequently, we have integrated the FORMES system in a LISP universe, maintaining a compatibility with the functional programming style [Bobrow et al. 1985].

General Principles

The virtual machine is organized as a set of LISP functions describing the internal representation of objects (i.e., associated fields), the message-passing form, the instantiation and inheritance mechanisms. To guarantee the portability of this machine, the functions are macros whose expansion provides S-expressions recognized by the aimed LISP. The only restriction concerning the portability criterion is the choice of a LISP with a dynamic scoping. The FORMES kernel is built upon the virtual machine and defines an abstract syntax using the object-oriented paradigms. It is now well known that this implementation technique allows the modification of the kernel without changing the user’s programs. This property is a powerful help in the maintenance, amelioration, and optimization of an evolving system used by nonexpert programmers.

Message-Passing

The message-passing simplifies the interface between the composer and the system, but must be implemented in an efficient way to avoid the penalization of the interpretative mechanism.

Object as LISP Symbol

For portability reasons we have chosen to implement objects without using LISP data-types other than symbol, cons-cell, and vector. Internal representation of each class of objects is mapped onto the structure of atomic symbol. This implementation model came from the ObjVlisp model [Cointe 1985]; it uses the properties of LISP atoms (the columns of LISP architecture) and allows functional objects, as we will see (but doesn’t allow anonymous objects). Each Le_Lisp symbol is represented by six fields:

LE_LISP ATOM = <C-VAL , P-LIST , OBJVAL , FVAL , TYPEFN , PTYPE>

The following section exposes the mapping of an object with a symbol, each object being built as a functional-value, a vector grouping together all the fields and a *class* factorizing the set of associated methods:

FORMES OBJECT = <*vector-fields* , *functional-value* , *f-type*>

vector-fields is put in the OBJVAL field,

the *f-type* - denoting the *f*ormes *type* (*class*) of the object - is put in the PTYPE (Pretty-print TYPE);

consequently, the C-VAL and P-LIST fields are free.

Functional Object

FORMES objects are generic functions that support a Smalltalk syntax without explicit *send*. The idea is to define each object as a macro, realizing the following expansion: center box; l. (*object selector Arg*1 ... *Arg*n) ≡ (send '*object selector Arg*1 *Arg*n)

For example, we could write (A_NOTE 'run) rather than (send 'A_NOTE 'run) to simplify the expression. We haven't used this facility in the paper to fully explain the message-passing expressions. To factorize the functional value shared by all the FORMES objects, it is kept in a global variable called *script*. The function *make-receiver* gives (in Le_Lisp) a functional value to the object *name* (i.e., bounds its two fields FVAL & TYPEFN). center box; l. (de make-receiver (name) (setfn name 'macro script))

```
(defglobal script
      '(call (rplaca call (list quote (car call))) (attach 'send call)))

(de attach (item l)
   (rplacd l (cons (car l) (cdr l)))
   (rplaca l item))
```

Methods as Packaged Functions

"All symbol names are packaged. Packaging is performed by using a new symbol property: the packagecell. Packagecells hold atoms, whose packagecell in turn can be used to determine a hierarchy of packages" [Chailloux et al. 1984].

As discussed in [Cointe and Rodet 1984], LISP provides several structures to implement a method-dictionary (as example an A-list, a P-list, or a SCHEME environment). The last version of FORMES uses the *package structure* of Le_Lisp. The two ideas are:

To associate to each class of objects a particular package, interning a different <*selector* , *method*> pair. Each selector is a LISP symbol whose functional value defines a method To use the *getfn1* primitive to look up a method given by its selector and its package. In Le_Lisp, *getfn1* is built for object-oriented languages' implementation and supports an efficient search algorithm based on a particular organization of the symbols' area [Chailloux 1986].

Consequently the definition of the *send* function is quite immediate: center box; l. (de send (-obj- -sel- . -args-)
   (apply (getfn1 (f-type -obj-) -sel-) (cons -obj- -args-)))

(send *a_process* '? *a_field*)
    ⇒   (apply (getfn1 (f-type *a_process*) '?) (list *a_process a_field*))

The function *f-type* returns the package of the receiver (#:object:process for a process). The function *getfn1* returns the symbol found in a package (the symbol #:object:process:-*sel*-). The functional value of that symbol is applied to the arguments (including the receiver itself to realize the *oself* construct).

Send as Macro

To optimize the message-passing evaluation, we define the *send* function as a macro expanding the object_level to the primitive lisp_level in two steps: center box; lI l. -1- object_level   (send '*a_process* '? *a_field*)

-2- macros_level   (#:object:process:? '*a_process a_field*)

-3- lisp_level (vref (indice *a_field* fields-keys) (objval '*a_process*))

The following definition of the *send* macro is available when the receiver (-obj-) and the selector (-sel-) are constants, i.e., quoted (in the other case the macro-expansion is incorrect): center box; l. (defmacro send (-obj- -sel- . -args-)
 `(,(getfn1 (f-type (cadr -obj-)) (cadr -sel-)) ,-obj- ,.-args-)

(send '*a_process* '? *a_field*) ⇒ (#:object:process:? '*a_process* *a_field*)

In fact, the *send* function expands in different calls, whether the receiver and selector are constants or not; in the first case, macro-expansion is the deepest.

Inheritance Mechanism

The inheritance mechanism is rather primary and addresses only the methods (actually we have no inheritance of the fields). This mechanism is also supported by the package construct defining a binary inheritance tree. Consequently when a method is not found in the package of the receiver, the search continues in the previous package, and so on, until the root (the void package) is found. (All classes are defined as subclasses of the *object* class, as an example a process knows the #*:object:process* package itself defined as a subpackage of #*:object*, thus the methods interned in the *object* package (ex: the method *selectors*) are recognized by all the FORMES objects.) The *getfn* construct of Le_Lisp generalizes the *getfn1* primitive: center box; c. (getfn1 *D_package selector*) ⇒ (getfn *D_package selector E_package*)

Until a method denoted by the symbol *selector* is found in the current package - *D_package* denotes the departure package - the search progresses in the previous one. *E_package* marks the end of the search.

Instantiation Mechanism

In order to define a new object, the user can choose between using a LISP way and a special primitive of the virtual machine (**dp** for a process) or using an ACT-1 way by sending the *new* message to an existing object of the same class.

The LISP Generators: dp, dmo, dt, dc, & db

All the generators are built on the same scheme: center box; l c l. C-VAL [d<class>] ⇒ list grouping together all objects of the class OBJVAL [d<class>] ⇒ list of keys recognized by the d<class> generator VALFN [d<class>] ⇒ the LISP code of the generator.

As an example: center box; l c l. C-VAL [dp] ⇒ (GOD) OBJVAL [dp] ⇒ (sons: monitor: first-time: each-time: ... env:)

Keywords allow the generator to comment at will.

The Generic Message: new

With every class is associated a *new* method contained in the associated package. This generic function recognizes the same keywords as the LISP generator.

Conclusion and Future Work

Representations of Musical Knowledge as Libraries of Processes

It is now recognized that the encapsulation mechanism of object-oriented languages - grouping together data and procedures - is a powerful aid in the development of knowledge representation systems. Accordingly, a database can continually memorize the processes resulting from analysis of musical performances or from scientists' and composers' imagination, building a kind *of living memory* of musical research. Thus in the FORMES universe, the main activity of the composer-programmer is to modify models, to derive new models from pre-existing ones, and to continuously compose any of them (for checking or for musical production).

Future Developments

A better formalization of the data-flow. At the present time, all the FORMES examples use these two orders (pre and post) to connect together the rules of active processes. This scheme is still too limited to allow a great number of musical structures to be *easily* and *intuitively* described. *Creation of new classes of objects*. There is no *meta-generator*, i.e., no FORMES construct supporting the definition of *dp*-like generators. *Iconic specification of a monitor*. The definition of a monitor is quite complex, and we have to work on an abstract representation allowing the specification and the automatic generation of such a musical scheduler. *Video animation and image processing*. We think that the FORMES abstraction is general enough to support other applications and, more particularly, video animation. *A definition of the virtual machine for COMMONLOOPS*. To follow the evolution of LISP toward lexical scoping [Steele 1984] - and to guarantee the portability of FORMES system - we have to transform our virtual machine. *Real-time implementation*. FORMES' first goal was to address the complexity of music representation. If the composers agree to its musical framework, then according to [Loy and Abott 1983], we consider that its architecture may be reconsidered to drive a real-time signal processing control. Acknowledgments We particularly thank Xavier Rodet - director of the CHANT-FORMES project - for his essential and active contribution to the conception of FORMES, and Jean-Baptiste Barrie‘re, who was the first composer to use and test the FORMES environment. We thank Je’roˆme Chailloux, Patrick Greussay, Jean-Franc;ois Perrot, and Harald Wertz for their sustained interest and support. We thank Pierre Boulez, Jacques Duthen, Tod Machover, Philippe Manoury, Yves Potard, Marco Stroppa, Jan Vandenheede, and every one at IRCAM for their encouragement and comments concerning musical aspects of the FORMES system. References

[Barrie‘re 1984] Barrie‘re, J. B., Chreode-1: the pathway to new music with computer, Musical Thought at IRCAM, Contemporary Music review, Vol. 1, No. 1, N. Osborne (ed.), T. Machover (issue ed.), August 1984, pp. 181-202.

[Barthes 1953] Barthes, R., Le degre’ ze’ro de l’e’criture, (chapitre: "Langue et Parole," §1.2.4), Bibliothe‘que Me’ditations, Editions Gonthier, Paris, 1953.

[Birtwistle et al. 1973] Birtwistle, G., O-J. Dahl, B. Myhrhaug, K. Nygaard, Simula Begin, Petrocelli/Charter, New York, 1973.

[Bobrow and Stefik 1983] Bobrow, D., M. Stefik, The Loops Manual, Xerox PARC, Palo Alto, CA, December 1983.

[Bobrow et al. 1985] Bobrow, D. G., K. Kahn, G. Kiczakes, L. Masinter, M. Stefik, F. Zdybel, CommonLoops: Merging Common Lisp and Object-Oriented Programming, IJCAI85 Draft, ISL-85-8, Xerox PARC, Palo Alto, CA, 16 August 1985.

[Boulez 1983] Boulez, P., Recherche et Cre’ation, Le monde de la Musique, No. 54, Paris, March 1983.

[Briot 1984] Briot, J-P., Instanciation et He’ritage dans les Langages Objets, (the‘se de 3e‘me cycle), LITP Research Report, No 85-21, LITP - Universite’ Paris-VI, Paris, 15 December 1984.

[Briot 1985] Briot, J-P., Metaclasses in Object-Oriented Languages, 5th AFCET Congress on Form Recognition and Artificial Intelligence, Grenoble, Savoie, 27-29 November 1985, Vol. 2, pp. 755-

764.

[Chailloux et al. 1984] Chailloux, J., M. Devin, J-M. Hullot, Le_Lisp a Portable and Efficient Lisp System, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 5-8 August 1984, pp. 113-123.

[Chailloux 1986] Chailloux, J., Recherche et Invocation des Me'thodes dans les Langages Objets, (draft), 3rd AFCET Workshop on Object-Oriented Programming, Centre Georges Pompidou, Paris, 8-10 January 1986.

[Cointe 1983] Cointe, P., A Vlisp Implementation of Smalltalk-76, Integrated Interactive Computing Systems, P. Degano and E. Sandewall (ed.), North-Holland, Amsterdam - New York - Oxford, 1983, pp. 89-102.

[Cointe 1984] Cointe, P., Imple'mentation et Interpre'tation des Langages Objets, Application aux Langages Formes, ObjVlisp et Smalltalk, (the‘se d'Etat), LITP Research Report, No. 85-55, LITP - Universite' Paris-VI - IRCAM, Paris, 17 December 1984.

[Cointe 1985] Cointe, P., The OBJVLISP Model: A Departure Platform in the Experimentation of Object-Oriented Formalisms, 5th AFCET Congress on Form Recognition and Artificial Intelligence, Grenoble, Savoie, 27-29 November 1985, Vol. 2, pp. 737-754.

[Cointe and Rodet 1984] Cointe, P., X. Rodet, Formes: an Object & Time Oriented System for Music Composition and Synthesis, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 5-8 August 1984, pp. 85-95.

[Dannenberg 1984] Dannenberg, R. B., Artic: A Functional Language for Real-Time Control, Conference Record of the 1984 ACM symposium on Lisp and Functional Programming, Austin, Texas, 5-8 August 1984, pp. 96-103.

[Goldberg and Kay 1976] Goldberg, A., A. Kay, Smalltalk-72 Instruction Manual, SSL 76-6, Xerox PARC, Palo Alto, CA, March 1976.

[Goldberg and Robson 1983] Goldberg, A., D. Robson, Smalltalk-80 The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

[Hewitt and Smith 1975] Hewitt, C. E., B. Smith, A Plasma Primer, AI Laboratory, MIT, MA, September 1975.

[Hoare 1974] Hoare, C.A.R., Monitors: An Operating System Structuring Concept, CACM, Vol. 17, No. 10, October 1974.

[Koechlin et al. 1985] Koechlin, O., et al., La Station de Travail Musicale 4X, IRCAM Research Report, No. 39, Centre Georges Pompidou, Paris, 1985.

[Krasner 1980] Krasner, G., Machine Tongues VIII: The Design of a Smalltalk Music System, Computer Music Journal, Vol. 4, No. 4, MIT Press, MA, Winter 1982, pp. 4-14.

[Lieberman 1982] Lieberman, H., Machine tongues IX: Object-Oriented Programming, Computer Music Journal, Vol. 6, No. 3, MIT Press, MA, Fall 1982, pp. 8-21.

[Lieberman 1986a] Lieberman, H., Delegation and Inheritance, Two mechanisms for Sharing Knowledge in Object-Oriented Systems, 3rd AFCET Workshop on Object-Oriented Programming, J. Bezivin and P. Cointe (ed.), Globule+Bigre, No 48, Paris, Janvier 86.

[Lieberman 1986b] Lieberman, H., Concurrent Object-Oriented Programming in Act 1, in Concurrent Object-Oriented Programming, A. Yonezawa and M. Tokoro (ed.), MIT Press, MA, September 1986.

[Loy and Abott 1983] Loy, G., A. C. Abott, Programming Languages for Computer Music, (draft), U.C. San Diego, CA, March 1983.

[Moon and Weinreb 1980] Moon, D. A., D. Weinreb, Flavors: Message Passing In The Lisp Machine, AI Memo, No. 602, MIT, MA, November 1980.

[Roads 1984] Roads, C., Research in Music and Artificial Intelligence, adapted from an invited lecture presented at the annual meeting of the Italian Computer Society, Padova, Italy (October 1982), MIT, MA, March 1984.

[Rodet et al. 1984] Rodet, X., Y. Potard, J.B. Barrie‘re, The CHANT Project: From Synthesis of the Singing Voice to Synthesis in General, Computer Music Journal, Vol. 8, No. 3, MIT Press, MA, Fall 1984.

[Rodet and Cointe 1984] Rodet, X., P. Cointe, Formes: Composition and Scheduling of Processes, Computer Music Journal, Vol. 8, No. 3, MIT Press, MA, Fall 1984, pp. 32-50.

[Serpette 1984] Serpette, B., Contextes, Processus, Objets, Se’quenceurs: FORMES, (the‘se de 3e‘me cycle), LITP Research Report, No. 85-5, LITP - Universite’ Paris-VI, Paris, 30 October 1984.

[Steele and Sussman 1975] Steele, G. L., G. J. Sussman, SCHEME: An Interpreter for Extended λ-Calculus, AI Memo, No. 349, MIT, MA, December 1975.

[Steele 1984] Steele, G. L., Common Lisp - The Language, Digital Press (DEC), Burlington, MA, 1984.