[Milner 1980] Milner, R., A Calculus of Communicating Systems, (Lecture Notes in Computer Science No. 92), Springer-Verlag, 1980.

[Odijk 1985] Odijk, E.A.M., *The Philips Object-Oriented Parallel Computer*, Woods, J.V. (ed.), Fifth Generation Computer Architecture (IFIP TC-10), North-Holland, 1985.

[Shaw 1981] Shaw, M. (ed.), ALPHARD: Form and Content, Springer-Verlag, 1981.

[Theriault 1983] Theriault, D.G., *Issues in the Design and Implementation of Act2*, Technical Report 728, MIT Artificial Intelligence Laboratory, June 1983.

[Weinreb and Moon 1980] Weinreb, D., D. Moon, *Flavors: Message Passing in the Lisp Machine*, Memo 602, MIT Artificial Intelligence Laboratory, 1980.

[Wirth 1982] Wirth, N., Programming in Modula-2, Springer-Verlag, 1982.

[Wulf et al. 1981] Wulf, W.A., R. Levin, S.P. Harbison, HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, 1981.

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.

1. Introduction

The FORMES system is a programming environment initiated at IRCAM[1] 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

[1] L'Institut de Recherche et Coordination Acoustique/Musique is "conducted" by Pierre Boulez.

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 [Barrière 1984], the first piece composed with FORMES by J. B. Barrière, 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.

2. Object-Oriented Metaphor Matches a Subset of MCS Requirements

2.1. 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]:

Generality: 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).

Universality: A model should try to be independent of a particular synthesis technique and should refer to universal concepts as found in acoustics and psychoacoustics.

Compatibility: 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.

Simplicity: Models will be much simpler if their program texts follow common composer communication conventions and presuppositions.

Uniformity: Uniform and clear symbolism should be used to create, modify, or integrate new models.

Modularity: 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.

2.2. 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.

3. 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.

3.1. A First Example

Let us present a simple example of a process whose goal is to produce a second octave C for 2 seconds.

```
(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:

```
(send 'A_NOTE 'ap)
```

Then we create (à la ACT-1 [Lieberman 1986b]) another note that differs only from the previous one by its pitch:

```
(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:


```
(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:

```
(db f1)
```

Then to display the pitch, after playing A_MELODY:

```
(send 'f1 'screen)
```

3.2. Informal Description of FORMES Concepts

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

3.2.1. 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.

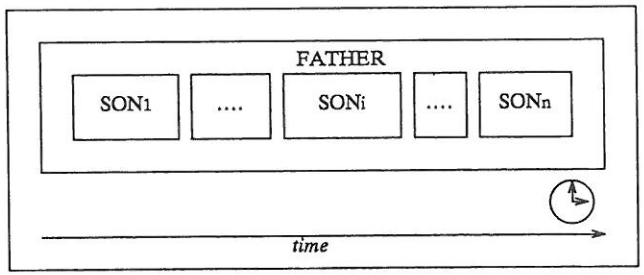
To address the complexity of the musical world, several kinds of predefined monitors are available, each one implementing a privileged musical structure. For instance:

- 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*.

3.2.2. 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.[2]

For example, the rule (field-to-register '(f1)), used by the A_NOTE process, will affect[3] 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).

[2] 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...

[3] 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.

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[4] *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.

3.2.3. 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.

3.2.4. 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 define a

[4] In a LISP formalism: (progn (send '*process1* 'rules) (send '*process2* 'rules))

process, **dmo** a monitor, etc...

3.2.5. 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].

3.3. 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>

3.3.1. 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:

monitor :	The name of the monitor associated with the process and denoted by the field *monitor:*
genealogical tree :	The description of the offspring of the process, defined as a binary tree - i.e., a list - and denoted by the field *sons:*
rules :	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*:*
sys-env :	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
exit :	A stop condition, which will be checked if specified in the field *exit:*
local-env :	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:

```
(dp process_name
  ; offspring & monitor
  sons:          (process_name1 ...)
  monitor:       monitor_name
  ; rules
  first-time:    implicit_progn1
  each-time:     implicit_progn2
  each-time*:    implicit_progn3
  last-time:     implicit_progn4
  ; 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.

3.3.2. 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:*}

init: defines the start time for each son process - field: *btime* -; it also evaluates the *first-time:* rules;

end: 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;

duration: 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;

offspring: 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:


```
(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:

```
(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) ⇒ (print end: init: duration: new offspring:)

3.3.3. 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:

```
(dt calculation tree)      ; define tree
```

3.3.4. 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:

```
(dc clock)      ; define clock

(db f1)         ; define buffer
```

4. 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.

4.1. 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.

4.1.1. Message-Passing

The syntax of a transmission is a generalization of the LISP "funcall form":

```
(send object selector Arg1 ... Argn)
```

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

4.1.2. 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,[5] requires an explicit access to the value of a field ("à la LOOPS" [Bobrow and Stefik 1983]).

[5] 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.

Then to read/write a field of a given object, we use two special transmissions:

```
(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.

4.1.3. 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:

```
(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.

4.1.4. 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;[6] **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:

```
(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

```
(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.

4.2. 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".

4.2.1. 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: *fibn*, *fibn-1* and *aux*.

[6] In fact, default argument of tnorm is *fself*, so we could also write (tnorm) rather than (tnorm fself).


```
(dp FIB
   monitor:     leaf
   first-time:  ((fsetq fibn-1 1 fibn 1)
                 (send 'clock '?<- 'time 1) (send 'clock '?<- 'quantum 1))
   each-time:   ((fsetq aux #@fibn) (fsetq fibn (+ #@fibn #@fibn-1))
                 (fsetq fibn-1 #@aux))
   last-time:   ((print #@fibn))
   env:         (duration 10 fibn '? fibn-1 '? aux '?) )
```

- The rule *first-time:* describes the initialization executed at each activation of the process: *fibn-1* and *fibn* receive the two first terms of the suite (fib0 = 1 and fib1 = 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:

```
first-time:   fibn-1 <- 1 ; fibn <- 1 ; quantum <- 1 ; duration <- 10
              for time = 2 by quantum to duration
               begin
each-time:       aux <- fibn ; fibn <- fibn + fibn-1 ; fibn-1 <- aux
              end
last-time:    print fibn
```

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.

4.2.2. Definition of a Trajectory for the Fundamental Frequency

This second musical example illustrates the methodology used by J. B. Barrière 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:

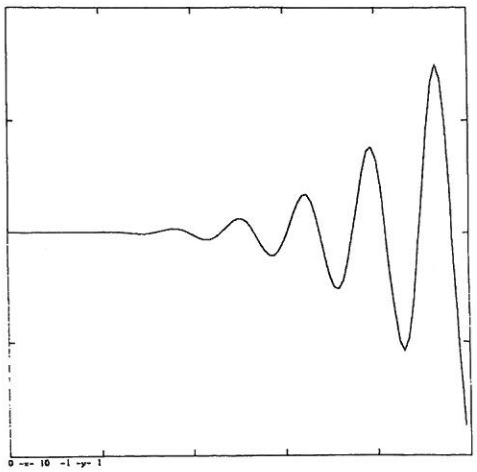


Figure 1. A damped sinusoid

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:

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

and here is the LISP translation:

```
(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:

```
(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.

4.2.3. 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**:

```
(send 'RED 'new 'GREEN 'env:  '(direction nil) )
(send 'GREEN 'new 'YELLOW 'env:  '(ϕ -π/2) )
(send 'YELLOW 'new 'ORANGE 'env:  '(direction t) )
```

4.3. 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.

```
(db output1)

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

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

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

(dp SUSTAIN
   monitor:     leaf
   env:         (duration 0.3) )

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

Figure 2. A model of NOTE

4.3.1. 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):

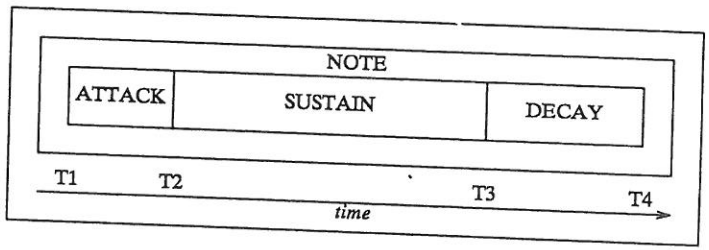


Figure 3. A 3-component envelope

4.3.2. Seq-node Constraints

Using the monitor **seq-node** with the process NOTE's offspring means the verification of these equalities:

```
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 subprocesses:

$$\text{(send 'NOTE '? 'duration)} = \Sigma\ \text{(send 'ATTACK '? 'duration)}\ \text{(send 'SUSTAIN '? 'duration)}\ \text{(send 'DECAY '? 'duration)}$$

4.3.3. 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:

```
t∈ [T1  T2]    (setq output1 #@amp) < (*= output1 (tnorm))
t∈ [T2  T3]    (setq output1 #@amp) < ()
t∈ [T3  T4]    (setq output1 #@amp) < (*= output1 (1-tnorm))
```

Remarks: