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:

```
                (send 'NOTE 'run) ≡
(NOTE_data-flow
  (send 'ATTACK '? 'duration) (send 'SUSTAIN '? 'duration)
  (send 'DECAY '? 'duration) (send 'NOTE '? 'amp))
```

```
(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.

4.3.4. 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.

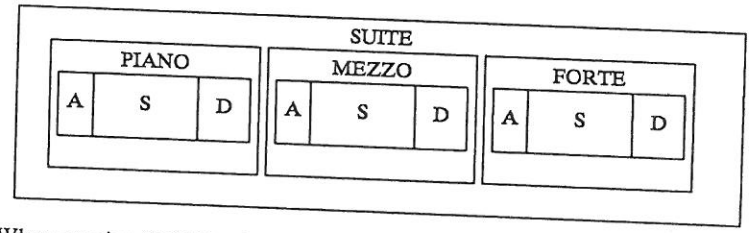
The process **SUITE** defines a sequence of three notes: **PIANISSIMO, MEZZOFORTE**, and **FORTISSIMO**:


```
(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 FORTISSIMO differ only by their respective *amp* values:

```
(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:

```
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:

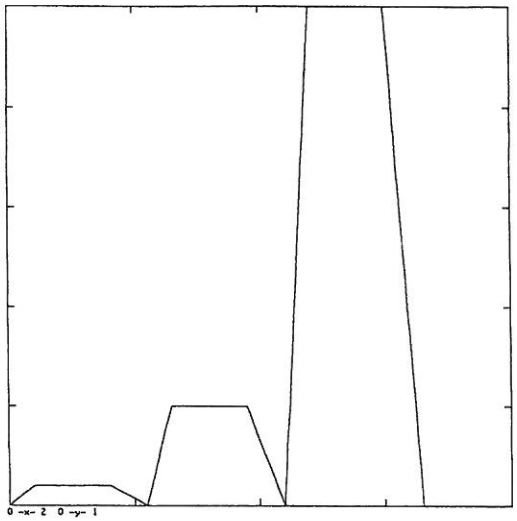


Figure 4. Running SUITE generates a sequence of envelopes

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.

4.4. 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].

4.4.1. A Pattern Matching for a Musical Sieve

The process **2-VOICES** defines a musical sieve.[7] It uses two voices for pattern matching:

[7] To study a more sophisticated example, [Cointe 1984] comments on a complex sieve realized for the 4X processor by P. Manoury.

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:

```
(db f1)

(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
(dp no-C
  monitor:     leaf
  env:         (duration 0.3) )

(send 'no-C 'new 'no-E)

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

(send 'C 'new 'E
  'env:        '(f1 130.) )
```

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

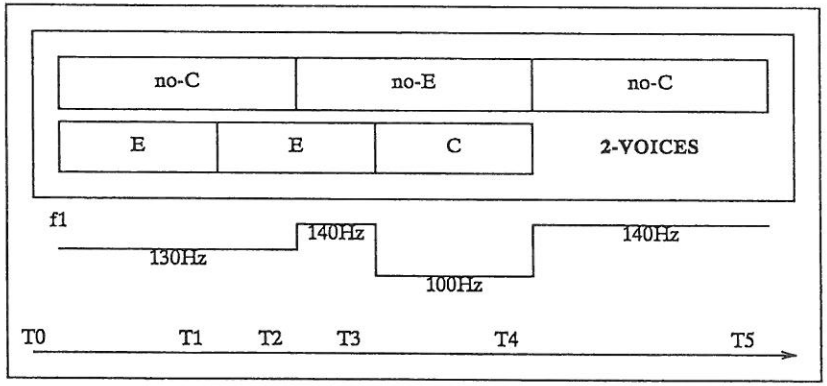
The function *unless-sieve* is a conditional progn. Its arguments are evaluated sequentially if the active process of the second voice is not time-matched by a **no**-process in the first voice:

```
(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.

4.4.2. Parallel-node Scheduling



4.4.3. 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).

4.4.4. 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):

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]

4.4.5. 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)):

```
(dp FIB1
  monitor:     leaf
  first-time:  ((fsetq fibn-1 1 fibn 2))
  each-time:   ((fsetq aux #@fibn) (fsetq fibn (+ #@fibn #@fibn-1))
               (fsetq fibn-1 #@aux))
  env:         (duration ∞ fibn '? fibn-1 '? aux '?) )

(send 'FIB1 'new 'FIB2 'first-time:  '((fsetq fibn-1 1 fibn 1)) )

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

4.5. Circ-node Monitor

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

4.5.1. 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:

```
(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*).

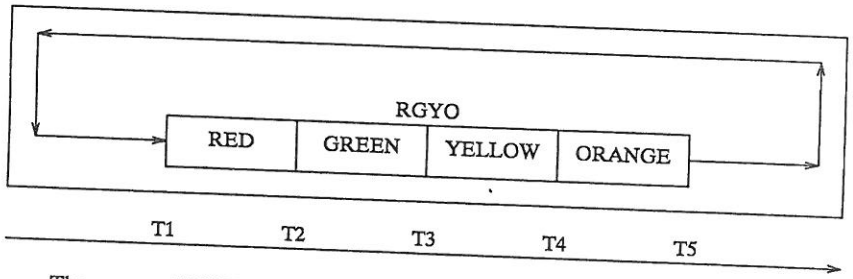
4.5.2. CHREODE or Color Compositions

"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" [Barrière 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:

```
(dp RGYO
  monitor:      circ-node
  sons:         (RED GREEN YELLOW ORANGE)
  first-time:   ((send 'clock '?<- 'quantum 0.1))
  exit:         (> time #@etime)
  env:          (duration 60.) )

(dp RED
  monitor:      leaf
  each-time:    ((setq f1 (trajectory #@father #@direction #@α #@ϕ #@ω))
                (send fself 'suspend))
  env:          (direction t α 4. ϕ π/2 ω 7.) )

(send 'RED 'new 'GREEN 'env: '(direction nil ω 15.) )
(send 'GREEN 'new 'YELLOW 'env: '(ϕ -π/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**:

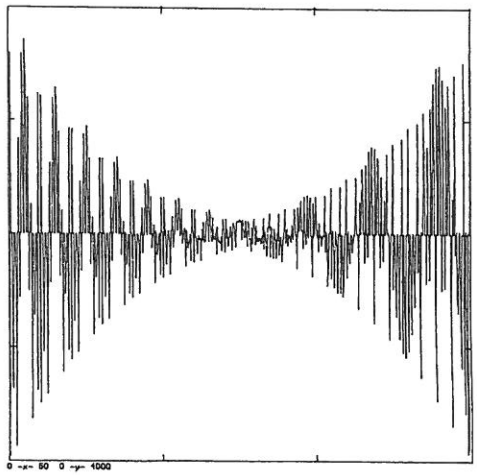


Figure 5. CHREODE, fundamental frequency: f1 (time)

5. 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):


```
(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[8] 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:

```
(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:

first-time: 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-)

[8] By the way, the seq-GOD monitor is defined as:

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

each-time*: 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-)

last-time: **GOD** (and **P**) finishing, a score is built (-3.1-) for the CHANT synthesizer.[9]

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.

6. The FORMES Virtual Machine

6.1. 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 "à la SMALLTALK" [Goldberg and Robson 1983] or an extension of LISP "à 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].

[9] 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.

6.2. 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.

6.3. 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.

6.3.1. Object as LISP Symbol

For portability reasons we have chosen to implement objects without using LISP datatypes 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 *formes type* (*class*) of the object - is put in the PTYPE (Pretty-print TYPE);

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

6.3.2. 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:

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

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).

```
(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))
```

6.3.3. 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:

```
(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).

6.3.4. 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:

-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):

```
(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.

6.4. 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:

```
(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.

6.5. 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.

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

All the generators are built on the same scheme:


```
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:

```
C-VAL [dp]   ⇒  (GOD)
OBJVAL [dp]  ⇒  (sons: monitor: first-time: each-time: ... env:)
```

Keywords allow the generator to comment at will.

6.5.2. 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.

7. Conclusion and Future Work

7.1. 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).

7.2. Future Developments

1. *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.
2. *Creation of new classes of objects.* There is no *meta-generator*, i.e., no FORMES construct supporting the definition of *dp*-like generators.

3. *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.
4. *Video animation and image processing.* We think that the FORMES abstraction is general enough to support other applications and, more particularly, video animation.
5. *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.
6. *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 Barrière, who was the first composer to use and test the FORMES environment.

We thank Jérôme Chailloux, Patrick Greussay, Jean-François 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

[Barrière 1984] Barrière, 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 degré zéro de l'écriture, (chapitre: "Langue et Parole," §1.2.4), Bibliothèque Méditations, 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 Aito, CA, 16 August 1985.

[Boulez 1983] Boulez, P., *Recherche et Création,* Le monde de la Musique, No. 54, Paris, March 1983.

[Briot 1984] Briot, J-P., *Instanciation et Héritage dans les Langages Objets,* (thèse de 3ème cycle), LITP Research Report, No 85-21, LITP - Université 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 Méthodes 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., *Implémentation et Interprétation des Langages Objets, Application aux Langages Formes, ObjVlisp et Smalltalk,* (thèse d'Etat), LITP Research Report, No. 85-55, LITP - Université 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. Barrière, *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, Séquenceurs: FORMES*, (thèse de 3ème cycle), LITP Research Report, No. 85-5, LITP - Université 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.

Concurrent Strategy Execution in Omega

Giuseppe Attardi

Omega is a description system for knowledge embedding which enables representation of knowledge in conceptual taxonomies. Reasoning on this knowledge can be carried out by a process called taxonomic reasoning, which is based on operations of traversing the lattice of descriptions. This process can be performed with a high degree of parallelism, by spreading the activities among the nodes of the lattice. Reasoning strategies expressed at the metalevel of Omega can be used to tailor deductions to specific applications. A message-passing approach is proposed to implement the deduction in Omega. An extension to Common LISP is suggested to provide the necessary message-passing primitives.

1. Introduction

Performing reasoning on a significant body of knowledge is a formidable task, which poses stringent requirements on the underlying knowledge representation system, both in terms of size of the knowledge base and in terms of performance. In the work on the Omega description system [Attardi and Simi 1981] the following ideas are explored:

- to use a description based knowledge representation system, where information is structured in conceptual taxonomies, and
- to base all reasoning on the traversal of such networks, using algorithms that can be executed with a high degree of parallelism.

An actor language is the natural choice for implementing a description system like Omega for several reasons. Omega descriptions, which represent collections of objects, are naturally implemented with actors. Concurrency in exploring the network of descriptions is suitably obtained by message passing between such descriptions. Ideally the greatest benefits of this approach could be obtained on an actor machine which could support efficiently message passing primitives. In order to experiment with the concurrent reasoning algorithms in a practical setting, I have defined an extension to the Common Lisp language to provide a minimum set of message passing primitives.