Composition de processus de transformation temps réel dans Elody

Stéphane Letz, Dominique Fober, Yann Orlarey
Grame 9, rue du Garet 69001 LYON
[letz, fober, orlarey]@grame.fr

***Résumé**:Elody est au départ un environnement pour la composition musicale permettant la description et la manipulation algorithmique de structures musicales et de procédés compositionnels en temps différé. Pour permettre la définition de processus de transformation temps réel, nous avons développé une extension du langage en lui ajoutant un nouvel élément primitif :* le flot d'entrée temps réel. *Cet objet peut être manipulé et transformé comme les objets musicaux temps différé avant même d'être complètement connu. L'évaluation d'une expression temps réel produit une séquence de commandes qui pilotent un moteur de rendu. Celui-ci transforme alors le flot d'entrée en un flot de sortie. Dans la suite de l'article, nous présentons cette approche et son application à la transformation de flots MIDI et audio.*

1 Les langages de composition musicale pour le temps réel

Pour la programmation des pièces interactives, il est nécessaire de disposer de langages de composition musicale adaptés à la spécification de processus d'interaction ou de transformation temps réel. Dans le cas de systèmes interactifs MIDI, on dispose d'outils de différents types :
- des langages spécialisés comme Key Kit [Thompson 1997] ou des librairies comme MidiShare [Fober & al. 1995] utilisables avec des langages généralistes (C/C++, Java...). Ces environnements donnent accès à des primitives de traitement des événements : par exemple les *alarmes*, des fonctions définies par l'application et exécutées lorsque les événements sont reçus, et des *tâches* pour décrire l'organisation temporelle des traitements en utilisant les services d'un ordonnanceur.
- des environnements qui permettent de manipuler des notions de plus haut niveau comme le langage Max [Puckette 1991]. Le programme est alors un graphe d'opérateurs effectuant en continu les opérations spécifiées sur les flots de valeurs reçues par l'application.

De manière générale, l'utilisateur décrit explicitement l'ordre des calculs et doit tenir compte de l'aspect temporel des données qui sont traitées. On utilise alors essentiellement des langages impératifs à état et effet de bord.
Bien que très généralistes dans la mesure où ils permettent de décrire tout type d'interaction, ces environnements ne sont pas toujours d'accès facile. Le plus souvent, l'utilisateur doit spécifier le problème en terme de description d'un système à état. De plus, la construction d'abstractions de plus haut niveau n'est pas toujours possible : par exemple dans un environnement graphique comme Max, la définition de patches simples est rapide car le processus et son interface utilisateur sont directement associés. Pourtant la construction de patches plus complexes peut être difficile en raison de la difficulté à *combiner les processus* plus simples dans des contextes d'utilisation différents, en particulier lorsque les programmes ont des états ou des effets de bords.

Dans le domaine des langages temps différé, la situation est assez différente : les modèles de programmation utilisés (objet, contraintes, fonctionnels) sont connus, leurs propriétés et leur sémantique bien définis. En particulier dans le cas des langages fonctionnels, on a une forte composabilité des expressions : la propriété de *transparence référentielle* stipule qu'une expression close dénote toujours la même valeur et ceci dans des contextes différents. De plus les fonctions sont des objets de première classe qui peuvent être librement combinés comme des données. Pourtant, la construction de programmes qui doivent interagir avec l'extérieur reste délicate. Les notions d'état, d'effets de bord, et en réalité toutes les notions habituellement utilisées pour accéder aux entrées/sorties n'existent pas directement. Ainsi le problème de l'intégration de notion d'entrée/sortie et d'état, ceci tout en gardant les bonnes propriétes du langage, reste un problème ouvert.

2 Entrée/sortie dans les langages fonctionnels

La façon standard de considérer les entrées/sorties dans un langage fonctionnel est de traiter ces opérations comme des fonctions qui *transforment l'état courant du système* en un *nouvel état* et qui produisent éventuellement un *résultat*. Ce type de fonction est couramment appelé une *action*. L'évaluation d'une expression qui agit sur l'environnement extérieur donne comme valeur une action qui peut être une *action simple* ou une *action complexe* combinaison d'actions plus simples. Les opérations de combinaison de ces actions garantissent leur exécution séquentielle et le passage des valeurs résultat entre les actions. Ainsi, on opère une distinction entre les calculs qui rendent des *valeurs simples* et les calculs qui *rendent une valeur* mais produisent aussi *un effet de bord*. Les opérations primitives de manipulation de ces actions sont "regroupées" dans le concept de *monad* [Wadler 1992]. Un monad est défini par un triplet (M, UnitM, BindM), où M est un constructeur de type, UnitM transforme une valeur en une action qui rend cette valeur sans autre effet, BindM permet de combiner séquentiellement des actions.
Une autre approche est celle utilisée dans le langage fonctionnel *Clean* [Eekelen, Plasmeijer 1998] avec la notion de *type unique*. L'état global du système est alors modélisé par une *structure unique* et le système de typage du langage garantit son utilisation séquentielle et l'impossibilité de la dupliquer.
L'avantage de ces méthodes est de retrouver pour la programmation d'opérations sur les entrées/sorties une écriture proche de celle utilisée dans un langage impératif classique, tout en bénéficiant des avantages de l'approche fonctionnelle. Nous nous sommes inspirés de ce cadre général pour ajouter dans Elody la possibilité de décrire des processus de transformations temps réel, tout en gardant l'approche de programmation homogène utilisée dans l'environnement. Le modèle général d'actions a été simplifié en considérant une action comme une fonction de transformation dont l'effet de bord sera de modifier le flot temps réel, mais qui ne produit pas de valeur utilisée par la suite dans l'évaluation.

3 Extension pour le temps réel dans Elody

3.1 Le flot temps réel

L'introduction du temps réel se fait par l'ajout d'un nouvel élément primitif dans le langage : le *flot temps réel*. C'est une séquence, infinie dans l'absolu, qui va contenir l'ensemble des événements reçus en temps réel par l'application. Cette séquence n'est connue qu'à partir du moment ou l'expression qui l'utilise est évaluée et réduite, mais il est possible de la manipuler comme si elle existait déjà, presque comme un objet temps différé. Cette abstraction de l'entrée discrète sous la forme d'un flot est d'une grande puissance. Elle ressemble en cela à la vision fichier qui est utilisée de manière homogène pour la manipulation des entrées/sorties sous Unix. D'une certaine manière, l'utilisateur manipule une *promesse de séquence* comme si les événements qu'elle va contenir étaient connus. C'est la responsabilité du système de traduire les opérations décrites par l'utilisateur sur cette séquence inconnue en opérations effectives qui seront réalisées en temps réel, lors du rendu de l'expression. Ainsi une spécification de haut niveaux pourra être traduite en une séquence de commandes élémentaires organisées dans le temps qui vont piloter un *moteur de rendu*. L'utilisateur décrit une spécification en terme d'organisation et de transformations dans le temps de portions du flot temps réel, le moteur de rendu exécutera les opérations bas niveaux à effets de bord qui réaliseront cette spécification.

3.2 Intégration dans le langage.

La technique utilisée procède en 3 étapes : l'*évaluation d'une expression* donne comme valeur une structure de donnée, en réalité un arbre. Les feuilles de cet arbre décrivent des actions, au sens défini au paragraphe précédent, et dans notre cas, des transformations qui vont être effectuées sur le flot temps réel. La phase de rendu qui rend perceptible une valeur est ici décomposée en 2 opérations :

- une fonction de *compilation* qui compile une valeur en une séquence de commandes ordonnées dans le temps.
- une opération de *transformation*, réalisées par le *moteur de rendu* qui agit sur le flot temps réel, à la réception des *commandes*, pour produire un flot résultat.

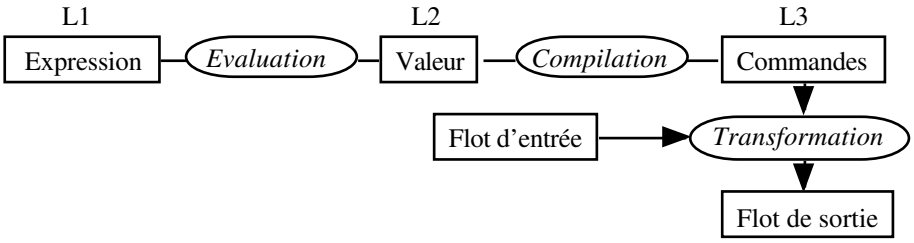

fig 1: schéma des traitements

Dans le cas général décrit précédement, les actions effectuent un effet de bord et produisent un résultat qui est réinjecté dans l'évaluateur et utilisé pour poursuive le calcul en cours. Ainsi évaluation et rendu doivent être mélangés. Nous simplifions le cas général de la manière suivante :

- les actions décrites sont des transformations, donc des actions à effet de bord, qui ne produisent pas de résultat au sens de l'évaluation. Ainsi le résultat de l'évaluation n'est pas réinjecté dans l'évaluateur.
- on ne manipule pas d'object infini. Une expression peut être évaluée en une seule fois pour produire une valeur qui est ensuite utilisée par les opérations de compilation/transformation. Ainsi, il n'y a pas d'interaction entre l'évaluation et les opérations de compilation/transformation. Cette réduction du problème permet aussi de rendre la communication entre un évaluateur non temps réel et un rendu temps réel plus simple. En pratique l'évaluation est faite en Java, les transformations temps réel en C.

4 Sémantique du langage

La figure 1 montre la suite des traitements internes qui, partant d'une expression construite par l'utilisateur, conduisent à son écoute. Le langage L1 est celui des *expressions* définies par l'utilisateur. Après une opération d'*évaluation,* on obtient un terme du langage des *valeurs* L2. Après *compilation* d'une valeur, on obtient un terme du langage des *commandes* L3. Les commandes pilotent le moteur de *transformation* qui agit sur le flot temps réel d'entrée pour produire un flot temps réel de sortie.

4.1 Evaluation des expressions

Nous présentons la sémantique de l'extension temps réel sur un sous ensemble du langage déja présenté dans [Orlarey & al.1997] et ceci sur un flot MIDI. C'est un langage sans abstraction et application avec un nombre réduit d'opérateurs primitifs qui se concentre sur l'aspect temps réel. Le langage $L1$ des expressions comporte les éléments suivants :

$sil[d]$	un silence de durée d
$input$	le flot temps réel de durée infinie
$seq[e,f]$	la séquence de 2 expressions e et f
$mix[e,f]$	le mixage (superposition temporelle) de 2 expressions e et f
$beg[e,d]$	le début de e pris sur une durée d
$rst[e,d]$	le reste de e privé de son début sur une durée d
$trp[e,n]$	la transposition de e de n unités
$xpd[e,c]$	la dilatation/compression de e d'un coefficient c

Le langage $L2$ des valeurs comprend les éléments suivants :

$sil[d]$	un silence de durée d
$input[d,t_{in},c,n]$	le flot temps réel pris sur une durée d, à partir d'une date t_{in}, avec un coefficient de compression/dilatation c, un facteur de transposition n.
$seq[e,f]$	la séquence de 2 expressions e et f
$mix[e,f]$	le mixage (superposition temporelle) de 2 expressions e et f

La fonction d'évaluation $E: L1 \Rightarrow L2$ qui traduit une expression en une valeur est ici très simplifiée par rapport à celle de [Orlarey & al.1997], elle n'utilise pas d'environnement.

$$E(input) \Rightarrow input[\infty, 0, 1, 0]$$
$$E(sil[d]) \Rightarrow sil[d]$$
$$E(seq[e, f]) \Rightarrow seq[E(e), E(f)]$$
$$E(mix[e, f]) \Rightarrow mix[E(e), E(f)]$$
$$E(beg[e, d]) \Rightarrow B(E(e), d)$$
$$E(rst[e, d]) \Rightarrow R(E(e), d)$$
$$E(trp[e, n]) \Rightarrow T(E(e), n)$$
$$E(xpd[e, c]) \Rightarrow X(E(e), c)$$

La fonction auxiliaire *Begin* découpe le début d'une valeur sur une durée d : $B: L2 \times Int \Rightarrow L2$. Elle prend en paramètre une valeur et un entier et rend une valeur.

$$B(input[d, t_{in}, c, n], d_1) \Rightarrow input[\min(d, d_1/c), t_{in}, c, n]^1$$
$$B(sil[d], d_1) \Rightarrow sil[\min(d, d_1)]$$
$$B(seq[v_1, v_2], d) \Rightarrow \begin{cases} B(v_1, d) & si \quad d <= D(v_1) \\ seq(v_1, B(v_2, d - D(v_1))) & si \quad d > D(v_1) \end{cases}$$
$$B(mix[v_1, v_2], d) \Rightarrow mix[B(v_1, d), B(v_2, d)]$$

La fonction auxiliaire *Rest* supprime le début d'une valeur sur une durée d : $R: L2 \times Int \Rightarrow L2$

$$R(input[d, t_{in}, c, n], d_1) \Rightarrow input[\max(d - d_1/c, 0), t_{in} + d_1/c, c, n]^2$$
$$R(sil[d], d_1) \Rightarrow sil[\max(d - d_1, 0)]$$
$$R(seq[v_1, v_2], d) \Rightarrow \begin{cases} R(v_1, d - D(v_1)) & si \quad d >= D(v_1) \\ seq(R(v_1, d), v_2) & si \quad d < D(v_1) \end{cases}$$
$$R(mix[v_1, v_2], d) \Rightarrow mix[R(v_1, d), R(v_2, d)]$$

La fonction auxiliaire *Dur* rend la durée d'une valeur exprimée dans le flot de sortie : $D: L2 \Rightarrow Int$

$$D(input[d, t_{in}, c, n]) \Rightarrow d * c$$
$$D(sil[d]) \Rightarrow d$$
$$D(seq[e, f]) \Rightarrow D(e) + D(f)$$
$$D(mix[e, f]) \Rightarrow \max(D(e), D(f))$$

La fonction auxiliaire *Transp* transpose une valeur de n unités : $T: L2 \times Int \Rightarrow L2$

$$T(input[d, t_{in}, c, n], n_1) \Rightarrow input[d, t_{in}, c, n + n_1]$$
$$T(sil[d], n) \Rightarrow sil[d]$$
$$T(seq[v_1, v_2], n) \Rightarrow seq[T(v_1, n), T(v_2, n)]$$
$$T(mix[v_1, v_2], n) \Rightarrow mix[T(v_1, n), T(v_2, n)]$$

La fonction auxiliaire *Expand* dilate une valeur par un coefficient c : $X: L2 \times Float \Rightarrow L2$

$$X(input[d, t_{in}, c, n], c_1) \Rightarrow input[d, t_{in}, c * c_1, n]$$

---

[1] la durée de l'opération *Begin* spécifiée ici est exprimée dans le flot de sortie, alors que la durée spécifiée dans *input[...]* est exprimée dans le flot d'entrée. Ces 2 durées sont réliées par le coefficient de dilatation/compression.

[2] la durée de l'opération *Rest* spécifiée ici est exprimée dans le flot de sortie, alors que la durée spécifiée dans *input[...]* est exprimée dans le flot d'entrée. Ces 2 durées sont réliées par le coefficient de dilatation/compression.

$X(sil[d],c) \Rightarrow sil[d * c]$
$X(seq[v_1, v_2],c) \Rightarrow seq[X(v_1,c), X(v_2,c)]$
$X(mix[v_1, v_2],c) \Rightarrow mix[X(v_1,c), X(v_2,c)]$

4.2 Compilation d'une valeur

Une valeur obtenue après évaluation d'une expression est *compilée* en une séquence de commandes qui représentent chacune une transformation à appliquer sur tous les événements reçus pendant une certaine durée. Pour une transformation donnée on a le schéma suivant :

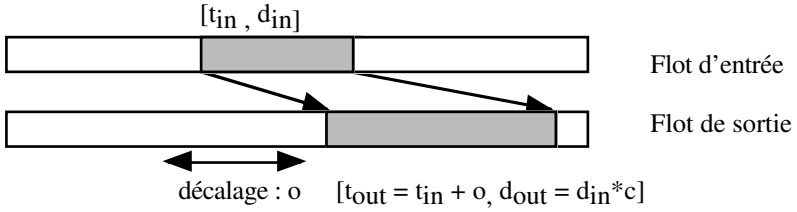

fig 2 : effet de dilatation et de décalage temporel d'une commande sur une portion du flot temps réel

Informellement, la transformation agit sur une *portion du flot d'entrée* (caractérisée par une date t_{in} et une durée d_{in}) et produit une *portion du flot de sortie* après une *déformation du temps* (caratérisée par un décalage *o* et un coefficient de dilatation/compression *c*) et la transformation des autres paramètres (hauteurs, vélocité...). Chaque transformation est codée sous la forme d'une commande: $cmd[d, t_{in}, o, c, n]$. La fonction de compilation prend en paramètre une *valeur*, la *date courante dans le flot de sortie* et produit une *séquence de commandes* et une *nouvelle date dans le flot de sortie* :

$Rn: L2 \times t_{out} \Rightarrow \{cmd\} \times t_{out}$
$Rn(input[d, t_{in}, c, n], t_{out}) \Rightarrow \{cmd[d, t_{in}, t_{out} - t_{in}, c, n]\} \times t_{out} + d * c$
$Rn(sil[d], t_{out}) \Rightarrow \{\ \} \times t_{out} + d$
$Rn(seq[v_1, v_2], t_{out}) \Rightarrow res_1 \bullet Rn(v_2, t_{out1}) \quad avec \quad res_1 = Rn(v_1, t_{out}) \Rightarrow \{.....\} \times t_{out1}$

La date finale t_{out1} du rendu de la valeur v_1 est passée en paramètre du rendu de la valeur v_2.
$Rn(mix[v_1, v_2], t_{out}) \Rightarrow Rn(v_1, t_{out}) \bullet Rn(v_2, t_{out})$

On utilise une opération de *mixage* de 2 séquences S et S' notée S • S' qui produit une séquence résultat en "mélangeant" les commandes issues de chaque séquence argument par ordre de date. En pratique chaque commande est codée par un couple de deux événements MIDI début et fin de transformation.

Exemple de compilation

Soit l'expression suivante qui décrit la mise en séquence du flot temps réel coupé sur 5 s avec lui-même : $seq[input[5000,0,1,0], input[5000,0,1,0]]$.
Après compilation on obient le résultat : $\{cmd[5000,0,1,0], cmd[5000,5000,1,0]\}$. Ainsi 2 *valeurs identiques* donneront 2 fonctions de transformations *différentes*.

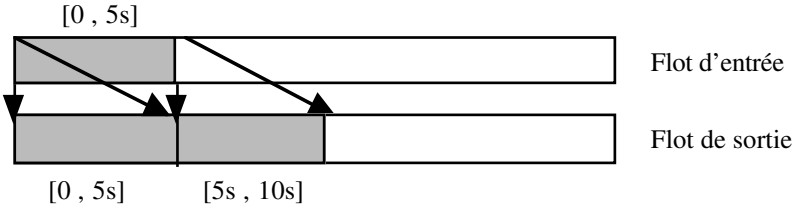

fig 3 : effet de la transformation : $\{cmd[5000,0,1,0], cmd[5000,5000,1,0]\}$

4.3 Transformation temps réel

Le résultat de la compilation est une séquence de commandes codées sous la forme de couples d'événement associés stokés dans une séquence. Celle-ci est jouée et les événements de commande pilotent le moteur de rendu. C'est une machine à état qui contient à un moment donné une *liste de commandes* en cours. Pour un événement entrant, toutes les transformations en cours sont appliquées en *parallèle*, et tous les événements produits sont envoyés vers la sortie.

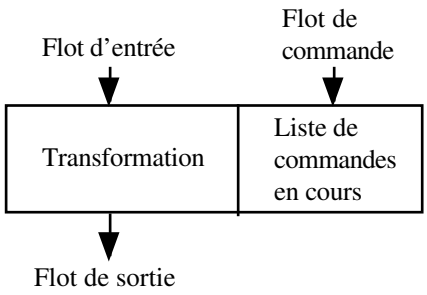

fig 4 : le moteur de rendu

L'état de la machine est constitué par la liste de commandes en cours à une date donnée : $Trans := [< trans >^*]$

Deux opérations modifient l'état courant de la machine :

$addCmd: cmd \times Trans \Rightarrow Trans$: ajoute une commande dans la liste des commandes en cours.
$remCmd: cmd \times Trans \Rightarrow Trans$: supprime une commande dans la liste des commandes en cours.
$handleCmd: cmd \times Trans \Rightarrow Trans$: à la reception d'une commande issue du flot de commande, utilise $addCmd$ ou $remCmd$ pour ajouter ou supprimer une commande dans la liste des commandes en cours.

A un instant donné, un événement reçu est transformé par une fonction : $Transform: ev \Rightarrow ev$ qui agit sur sa date et sa hauteur de la manière suivante :

$t_{out} = \max((t_{in} - t_{fun}) * c + o + t_{fun}, t_{cur})$: transformation de la date ou t_{in} est la date de l'événement dans le flot d'entrée, c le coefficient de dilatation/compression, o le décalage temporel, t_{fun} la date de l'événement de transformation, t_{cur} la date courante. [La date de l'événement dans le flot de sortie peut être inférieure à la date courante, c'est à dire être *dans le passé,* dans ce cas il sera joué à la date courante].

$p_{out} = p_{in} + n$: transformation de la hauteur

L'événement reçu est transformé par chaque fonction contenue dans la liste des transformations en cours, les événements résultat sont ordonnés par date dans un flot de sortie. C'est le rôle de la fonction $transformList: ev \times Trans \Rightarrow Output$

$Merge: Output_1 \times Output_2 \Rightarrow Output$: produit un flot obtenu en mélangeant les événements des 2 flots arguments par ordre de date.

Au final, l'ensemble du comportement de la machine à état peut être décrit par l'opération : $Tr: Input \times Cmd \times Trans \times Output \Rightarrow Output$ où on a :

$Input := [< in >^*]$	liste d'évenements MIDI reçus
$Cmd := [< cmd >^*]$	liste d'événements de commande
$Output := [< out >^*]$	liste d'événements MIDI produits
$Trans := [< trans >^*]$	liste des transformations en cours à une date donnée.

$$Tr(< null >, Com, Trans, Ouput) \Rightarrow Output$$

$$Tr(< ev::in >, < null >, Trans, Ouput)$$
$$\Rightarrow Merge(TransformList(ev, Trans), Tr(< in >, < null >, Trans, Output))$$

$$Tr(< ev::in >, < ev_1::cmd >, Trans, Ouput)$$
$$\Rightarrow \begin{cases} Tr(< in >, < ev_1::cmd >, Trans, Merge(transformList(ev, Trans), Output)) & si \quad Date(ev) <= Date(ev_1) \\ Tr(< ev::in >, < cmd >, handleCmd(ev_1, Trans), Output) & si \quad Date(ev) > Date(ev_1) \end{cases}$$

5 Exemples de transformations temps réel

La composition de processus de transformation temps réel devient très proche de la composition habituelle de structures temps différé dans Elody. Les capacités spécifiques du langage, en particulier la définition de *partitions de programmes*, c'est à dire de processus organisés dans le temps, existent aussi pour les processus d'interaction temps réel. La métaphore de la partition, très largement utilisée pour la composition en temps différé devient utilisable pour le temps réel. La possibilité de définir des transformations ou interactions temps réel à ce haut niveau d'abstraction libère l'utilisateur du souci de l'ordonnancement temporel des opérations primitives de transformation avec effet de bord. Les processus de transformation temps réel deviennent des objets de première classe librement composables en des opérations plus complexes. Les exemples suivants sont réalisés avec une implémentation agissant sur un flot temps réel MIDI.

5.1 Constructions temporelles : “thru”, délai, écho

Le processus le plus simple que l'on peut décrire est un “thru”, redirection de l'entrée vers la sortie. Il est défini par l’expression suivante : $input[\infty, 0, 1, 0]$

Si le flot temps réel est maintenant décalé dans le temps de 1 seconde, la sortie produite à partir de la date 0 est constituée d'un silence de 1 seconde suivit de l'entrée capturée à partir de la date 0 : on obtient donc un processus qui renvoie chaque événement reçu avec un delai de 1 seconde : $seq[sil[1000], input[\infty, 0, 1, 0]]$

Plusieurs instances du flot temps réel peuvent être traitées individuellement et mixées : l'expression suivante décrit un *écho* où chaque événement reçu et renvoyé immédiatement, avec un delai de 1 seconde et avec un delai de 2 secondes.

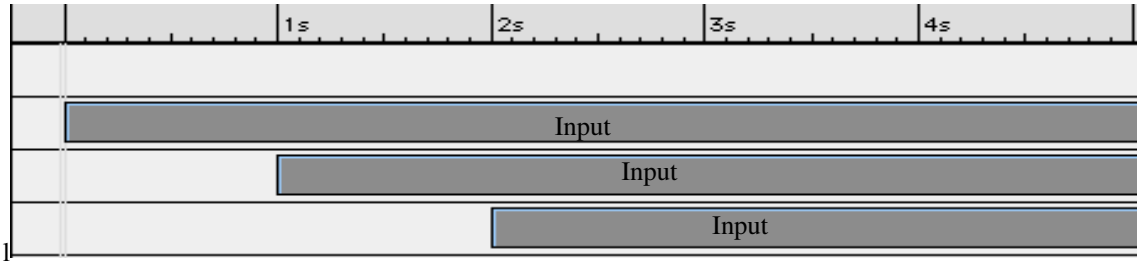

fig 5 : un triple écho temps réel

5.2 Découpage temporel

Comme pour un objet temps différé, il est possible de couper une portion du flot temps réel : dans l'exemple suivant, on coupe les 5 premières secondes du flot temps réel et le résultat obtenu est répété 2 fois. Après évaluation, le processus résultant effectue un “thru” sur les 5 premières secondes, et répète ensuite entre les dates 5 et 10 s ce qui a été reçu entre les dates 0 et 5 s.

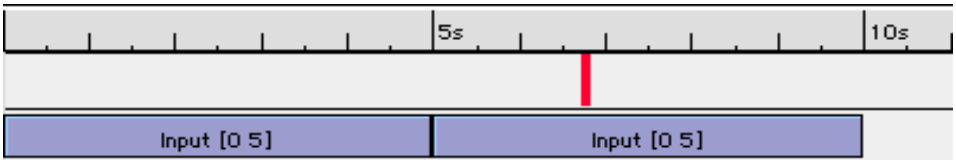

fig 6 : répétition 2 fois d'une portion de 5 s du flot temps réel

5.3 Compression/dilatation

Le flot temps réel peut être compressé ou dilaté. Il est alors possible de définir des processus qui déforment le temps et dont le fonctionnement est incompatible avec le temps réel : par exemple la compression d'un facteur 2 du flot temps réel. Pour tout processus de ce type, la convention est d'envoyer les événements au plus tôt, c'est à dire que tous les événements dont la date après transformation appartient au passé seront joués à la date courante.

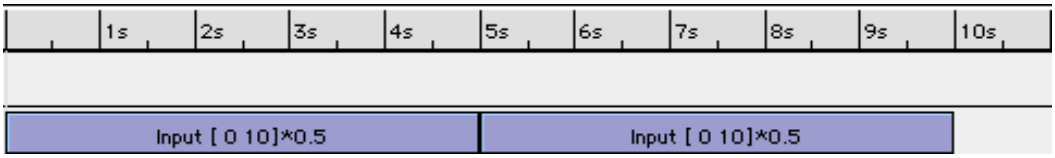

fig 7 : répétition 2 fois d'une portion de 5 s du flot temps réel compressé 2 fois

5.4 Transformations placées dans le temps

Il est possible dans Elody de construire des partitions de programmes, c'est à dire des abstractions organisées dans le temps en Séquence ou Mix et qui s'appliquent sur un argument [Orlarey & al.1997]. Comme en temps différé, les *partitions de programmes* peuvent être appliquées sur le flot temps réel de façon à spécifier (par exemple) une séquence de transformations effectuées sur des portions successives de l'entrée temps réel. Dans l'exemple suivant, 2 transformations différentes sont appliquées sur le flot temps réel : une fonction *canon* entre les dates 0 et 5 s et une fonction *écho* entre les dates 5 et 10 s.

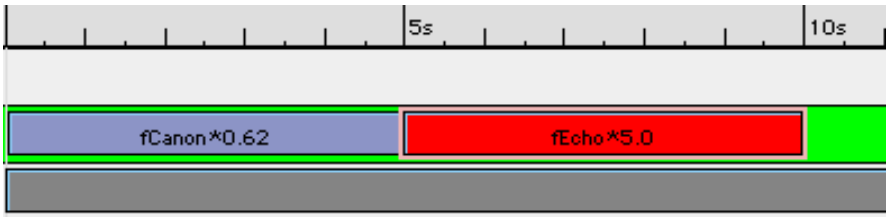

fig 8 : application d'une séquence de fonctions sur le flot temps réel

5.5 Mixage temps réel, temps différé

Comme la composition temps réel devient très semblable à la composition temps différé, il est très facile d'utiliser en temps réel des opérations préalablement définies sur des structures temps différé (et vice-versa). Il est possible aussi d'effectuer des transformations qui agissent sur une expression qui "mélangent" des parties temps réel et temps différé. Dans l'exemple suivant une séquence d'abstractions est appliquée au mixage de l'entrée temps réel et d'une séquence temps différé *seq1*.

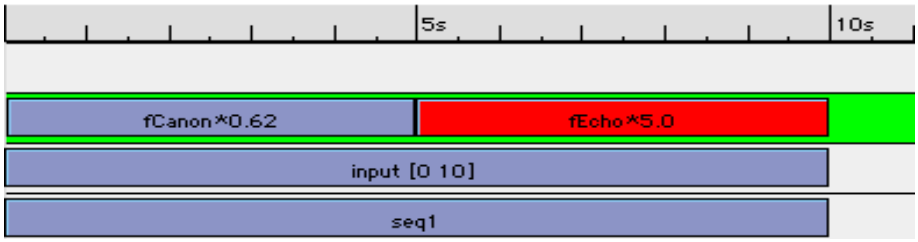

fig 9 : application d'une séquence de fonctions sur le mixage d'un objet temps différé et du flot temps réel

5.6 Composition à plusieurs niveaux

Un objet résultant d'un premier niveau de composition peut devenir le matériaux de base de niveaux de construction plus élaborés. Dans l'exemple suivant on a :

- premier niveau : application d'une séquence de transformations sur le flot temps réel, ici une fonction *canon* appliquée sur des portions successives de 4 secondes.

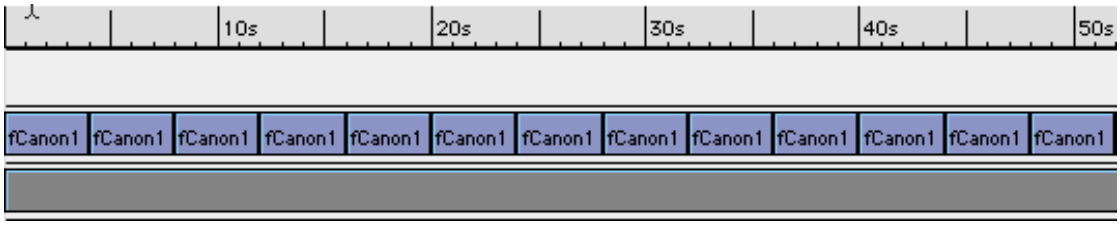

fig 10 : application d'une séquence de fonctions *canon* sur le flot temps réel

- deuxième niveau : le flot résultant de la première transformation est découpé en portions de 16 secondes, nommées respectivement A, B et C. Ces éléments sont utilisés comme matériaux de base d'un 2° niveau de composition suivant le shéma de construction temporelle suivant :

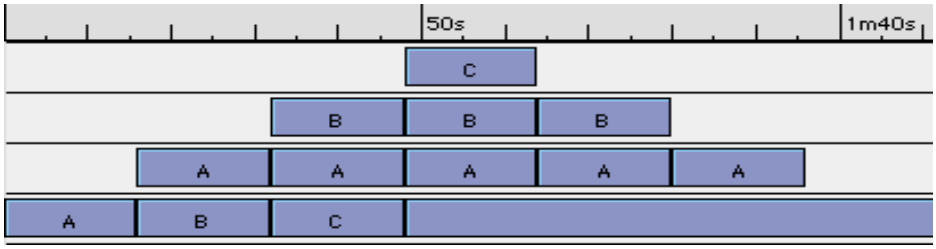

fig 11 : construction à partir du flot résultat de la fig 10

Tout élément déja transformé, objet de première classe, peut être librement réutilisé et composé dans des constructions plus complexes. Les fonctions définies en temps différé peuvent être utilisées sur des objets temps réel et vice-versa. Il est ainsi possible de construire très rapidement des programmes d'interactions très complexes par *combinaison* et *réutilisation* de processus plus simples.

6 Version Audio

Nous avons implémenté une version audio du mécanisme précédemment décrit sur le flot Midi. L'entrée temps réel est dans ce cas le flot des échantillons reçus par l'application qui pourra être manipulé comme le flot MIDI. Les opérations sur le flot audio temps réel sont traduites après évaluation et compilation en une séquence de commandes qui comprennent les actions suivantes :

$rec[Id, t_{in}, d]$: enregistrement d'une portion du flot temps réel à partir de la date t_{in}, sur une durée

d. Cette opération est séparée en 2 événements :

$recOn[Id, t_1]$: début d'enregistrement

$recOff[Id, t_2]$: fin d'enregistrement avec $t_2 = t_1 + d$

play$[Id, t_{in}, d]$: lecture d'une portion du flot temps réel préalablement enregistrée à partir de la date t_{in}, sur une durée *d*. Cette opération est séparée en 2 événements :

$playOn[Id, t_1, < effets* >]$: début de lecture d'une portion du flot temps réel avec application d'une liste d'effets.

$playOff[Id, t_2]$: fin de lecture avec $t_2 = t_1 + d$

$effet := pitch[m, p] \mid chorus[m_1, m_2] \mid reverb[m]$

Les commandes sont traduites en événements MIDI System Exclusif qui sont insérés dans une séquence, joués et envoyés au moteur de rendu audio. Nous avons implementé une version simple qui contient les effets décrits précédemment. De même que pour la version MIDI, le moteur de rendu reçoit et exécute des commandes et transforme le flot audio d'entrée en un flot audio de sortie.

7 Conclusion

Nous avons présenté une extension du langage Elody qui permet la définition de processus de transformation temps réel. Une nouvelle primitive est ajoutée au langage : le flot temps réel. Il peut être manipulé avec les opérateurs existant comme un objet temps différé. Les transformations temps réel deviennent des objets de première classe librement composables en des processus plus complexes. Elles sont compilées en commandes simples, ordonnées dans le temps, qui pilotent un moteur de rendu. Celui-ci transforme un flot d'entrée en un flot de sortie. Ce travail constitue une étape dans la description de processus interactifs avec un langage purement fonctionnel. Nous pensons ajouter par la suite les primitives nécessaires pour décrire des processus *temps réel réactifs* c'est à dire dont l'activation est liée à l'occurence d'événements extérieurs, ceci en gardant les facilités actuelles de composition des processus. Il sera alors possible de décrire une classe plus large de programmes interactifs.

Références

[Eekelen, Plasmeijer 1998] Marco Van Eekelen, Rinus Plasmeijer. *Concurrent Clean language report.* High Level Software Tools B.V and University of Nijmegen 1998.

[Elliot, Hudak] Conal Elliott, Paul Hudak. *Functional Reactive Animation.* In the proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97).

[Fober & al. 1995] Fober D., Letz S., Orlarey Y. - *MidiShare, un système d'exploitation musical pour la communication et la collaboration* - Actes des Journées d'Informatique Musicale JIM95, Paris, pp.91-100.

[Orlarey & al.1997] Y Orlarey, D Fober, S Letz. *L'environnement de composition musicale Elody* . Actes des 4° Journées d'Informatique Musicale JIM 97 Lyon pp 122- 136.

[Puckette 1991] Miller Puckette. *Combining Event and Signal Processing in MAX Graphical Programming Environment.* Computer Music Journal Vol 15, number 3 1991 The MIT Press.

[Thompson 1997] Tim Thompson *Key Kit Midi Language* Manuel d'utilisation 1997.

[Wadler 1992] Philip Wadler *Comprehending monads*. Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming,2:461-493, 1992.