Résumé

Dans le cadre de la musique dite *mixte*, le langage d'*Antescofo* permet de spécifier conjointement parties électroniques et parties instrumentales. L'ordinateur, peut ainsi au moment de la performance, être considéré comme un musicien qui interagit avec les autres instrumentistes. En ce sens, une partition *Antescofo* permet de spécifier un système réactif particulier : une performance musicale. Nous étudions ici le langage d'*Antescofo* à la lumière des langages synchrones classiques, comme *Lucid Synchrone* ou *Reactive ML*, qui sont spécialement conçus pour la spécification de systèmes réactifs.

Antescofo is a software for composition and performance of realtime computer music and composed of a coupled score follower and specification language for composition and performance of mixed live electronics pieces. During a performance, the computer interacts with other musicians. Therefore, an *Antescofo* score becomes a specification of a reactive system : a musical performance. Synchronous languages are precisly designed to specify reactive systems. We study here the link between the language of *Antescofo* and classical synchronous languages such as *Lucid Synchrone* or *Reactive ML*.

Table des matières

1 Introduction **5**

2 Le langage d'*Antescofo* **9**
2.1 Présentation . . . 9
2.2 Le langage original . . . 9
2.3 Indépendances des attributs . . . 11
2.4 Analyse de causalité . . . 15
2.5 Formalisation et ensemble de types . . . 17

3 Un interprète *Antescofo* en Reactive ML **19**
3.1 Le langage *Reactive ML* . . . 19
3.2 Fonctionnement de l'interprète *Antescofo* . . . 21
3.3 Discussion . . . 26

4 L'expérience *Lucid Synchrone* **27**
4.1 Le langage *Lucid Synchrone* . . . 27
4.2 Normalisation d'une partition *Antescofo* . . . 29
4.3 Fonctionnement de l'interprète . . . 31
4.4 Discussion . . . 36

5 Évaluation des interprètes **39**
5.1 Partitions et performances . . . 39
5.2 Couplage avec Max/MSP . . . 39
5.3 Évaluation . . . 40

6 Conclusion **43**

1 Introduction

Dès le milieu du $XX^{\text{ème}}$ siècle, les compositeurs commencent à s'intéresser à la notion de musique dite *mixte*. Avec l'évolution des techniques d'enregistrement il devient, en effet, possible d'associer des instrumentistes qui jouent en direct à des sons pré-enregistrés.

Bruno Maderna, par exemple, composera dès 1951, *musica su due dimensioni* pour flûte percussion et bande. Pour la première fois, on assiste dans cette pièce à un véritable dialogue entre l'intrumentiste et la bande magnétique [1].

Cette forme de dialogue sera souvent critiquée par les interprètes en raison du caractère immuable et figé des sons enregistrés. L'ordinateur, en autorisant une véritable forme d'interaction entre les musiciens et les sons électroniques va donc, petit à petit bouleverser ce genre émergent. C'est dans ce contexte qu'est né le projet *Antescofo* [2].

Antescofo

On définit traditionnellement le suivi de partition comme l'alignement automatique, et en temps réel, d'un flux audio joué par un ou des musiciens sur une partition musicale symbolique. Le suivi de partition permet, de plus, de gérer les erreurs et oublis éventuels des musiciens. Un suiveur de partition peut ensuite être couplé à un système réactif capable de déclencher des événements sonores. L'ordinateur devient alors un musicien capable d'écouter les autres instrumentistes pour se synchroniser [3].

C'est grâce à l'émergence de tels systèmes qu'il est de nos jours possible de véritablement interpréter de la musique mixte. Sans système de suivi, les instrumentistes sont contraints de se synchroniser sur une bande sonore fixe, imperméable à toute tentative d'interprétation.

À l'heure actuelle, *Antescofo* [4] est considéré comme l'état de l'art de ce domaine. Son système de suivi repose sur une estimation couplée de la position au sein de la partition et du tempo (ou vitesse d'exécution) de la performance. Ainsi, lors de la performance, l'ordinateur dispose à tout moment de deux informations :

- la position de l'événement détecté,
- le tempo estimé lors de la détection.

Le tempo est estimé en fonction des instants de détection des événements instrumentaux. C'est un processus complexe qui est, en soi, un domaine de recherche à part entière [5]. Cette estimation permet ensuite d'anticiper la détection suivante. *Antescofo* est donc un système de suivi *anticipatif* (d'où le nom *Antescofo*, pour *Anticipatory Score Follower*).

Parallèlement, *Antescofo* est équipé d'un séquenceur qui récupère les informations de position et de tempo. Celui-ci est donc capable de déclencher des actions électroniques en suivant, à l'instar d'un véritable musicien, la position et le tempo de l'interprète. L'ordinateur peut ainsi être considéré comme un instrumentiste à part entière.

L'originalité et l'efficacité du système font que, de nos jours, *Antescofo* est utilisé régulièrement par des compositeurs tels que Pierre Boulez, Marco Stroppa ou Philippe Manoury.

Systèmes réactifs

On peut distinguer trois types de sytèmes informatiques :

- Les sytèmes transformationnels n'ont pas de contrainte temporelle. Les données nécessaires au traitement sont disponibles dès le lancement de l'application. Les logiciels de traitement de bases de données ou de calcul scientifique sont des exemples de systèmes transformationnels.

- Les sytèmes interactifs sont soumis à des contraintes temporelles un peu plus fortes. Les instants de production respectent des valeurs statistiques. Les résultats dépendent de données produites par l'environnement. On peut par exemple citer les logiciels de bureautique.

- Les systèmes réactifs sont des systèmes qui sont en interaction permanente avec leur environnement. Aux sollicitations de leur environnement, les systèmes réactifs répondent par des réactions. En retour, ces réactions peuvent modifier l'environnement [6]. En ce sens, un instrumentiste évoluant au sein d'un ensemble de musiciens peut être considéré comme un système réactif.

Le langage d'*Antescofo* [7] permet de spécifier conjointement la performance humaine et la description des événements électroniques pris en charge par le séquenceur. Les parties électroniques peuvent ainsi être considérées de la même manière qu'une partie instrumentale classique. *Antescofo* est donc un langage spécialisé conçu pour spécifier un système réactif particulier : une performance musicale.

Les langages synchrones

Les langages synchrones tels que *Lustre* [8], *Esterel* [9, 10] ou *Signal* [11], sont apparus il y a une vingtaine d'années, précisément pour faciliter la programmation de systèmes réactifs. Ils permettent, en particulier, d'intégrer la notion de temps directement dans le langage de programmation.

Le concept de réaction est fondamental pour les systèmes embarqués. Il est par exemple nécessaire qu'un avion, une voiture ou un train soit extrêmement réactifs à son environnement. Pour cette raison les langages synchrones ont eu un grand succès pour la programmation de systèmes embarqués. Pour plus de précisions sur les diverses applications de ces langages, on pourra se reporter à [12].

Le langage d'*Antescofo* permet lui aussi de spécifier un système réactif. Il est donc légitime de se demander, d'une part, quelle place occupe le langage d'*Antescofo* au sein des divers paradigmes de programmation synchrone. Et d'autre part, ce que que peuvent apporter les langages synchrones à un système tel qu'*Antescofo*.

Sémantique par traduction

L'objectif de cette étude est de confronter les structures de contrôle du langage d'*Antescofo* aux divers paradigmes de programmation synchrone. La traduction de ces structures dans un formalisme synchrone, permettrait en particulier de donner une véritable sémantique par traduction au langage. Cela suppose cependant, que la sémantique du langage synchrone cible soit bien définie.

Dans ce cadre, nous nous sommes principalement intéressés à deux langages synchrones correspondant à deux paradigmes bien distincts :

- *Reactive ML* [13], est un langage réactif dans le style d'*Esterel*. Il permet de décrire des suites d'instructions à exécuter successivement ou en parrallèle.

- *Lucid Synchrone* [14] est un langage flot de données plus proche de *Lustre*. Il repose sur la notion de flot de données qui s'écoulent dans le temps.

Ces deux langages ont une sémantique claire et connue et présentent, en outre, l'avantage d'être des langages fonctionnels de haut niveau.

Ce rapport est organisé de la manière suivante :

- La section 2 décrit les différentes structures de contrôle du langage d'*Antescofo*.
- La section 3 détaille la traduction de ces structures dans le langage *Reactive ML*.
- Dans la section 4, nous abordons les problèmes posés par une approche *flot de données*.
- Enfin, dans une dernière partie, nous présentons une ébauche de procédure d'évaluation.

2 Le langage d'*Antescofo*

2.1 Présentation

Le langage d'*Antescofo* permet de spécifier conjointement la performance humaine et la description des actions électroniques de manière raisonnablement expressive (on pourra en trouver une description détaillée dans [15]). Ce formalisme permet de minimiser la différence de traitement entre parties électroniques et parties instrumentales, ce qui facilite la conception de la pièce pour le compositeur.

La figure 1 montre un exemple simple de partition et sa formalisation dans le langage d'*Antescofo*. Sur cet exemple, deux événements sont attendus de la part des instrumentistes, ils sont indiqués par le mot clef : `NOTE`. Deux actions électroniques, `a11`, `a12`, `a21` et `a22` sont attachées à chacun de ces événements. Ces actions sont groupées au sein de structures indiquées par le mot clef `Group`. Enfin, les délais devant les actions sont spécifiés relativement au tempo courant. Ainsi, $\frac{1}{2}$ signifie un délai d'un demi temps.

Notons qu'*Antescofo* fonctionne par envoi de message, les actions électroniques n'ont donc pas de durée. Elle sont caractérisées par un délai qui correspond au temps d'attente avant l'envoi du message.

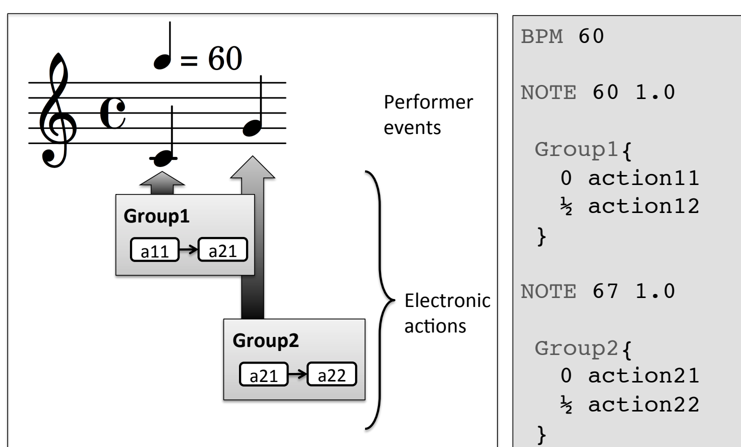


Figure 1 – Extrait d'une partition et de sa description dans le langage d'*Antescofo*. Les actions électroniques sont indiquées sur fond gris.

2.2 Le langage original

Nous considérons ici, un sous ensemble du langage tel qu'il est décrit dans [7]. Dans ce cadre, une partition est une alternance d'événements et d'actions électroniques spécifiée par la grammaire suivante.

$$
\begin{aligned}
\textit{score} &::= \varepsilon \mid \textit{event score} \mid (d\ \textit{action})\ \textit{score} \\
\textit{event} &::= (e\ c) \\
\textit{group} &::= \texttt{group}\ \ell\ \textit{synchro error}\ (d\ \textit{action})^{+} \\
\textit{action} &::= a \mid \textit{group} \\
\textit{synchro} &::= \texttt{loose} \mid \texttt{tight} \\
\textit{error} &::= \texttt{local} \mid \texttt{global}
\end{aligned}
$$

- La séquence vide est notée ϵ.
- Un événement $(e\ c)$ désigne un événement instrumental de durée c.
- Une action a désigne une action atomique, c'est-à-dire un simple envoi de message. Les structures de groupe permettent de composer des actions.
- Un délai d devant une action indique un délai relatif au tempo. C'est la durée qu'il faut attendre avant d'exécuter l'action.
- Les actions composées (`group`) sont identifiées par un label ℓ. Les attributs *synchro* et *error* permettent de spécifier une stratégie de synchronisation et une procédure de rattrapage d'erreur.

Les structures de contrôle du langage original que nous avons laissé de côté peuvent être simulées à l'aide des constructions précédentes.

On peut remarquer que rien n'interdit de définir un groupe à l'intérieur d'un autre groupe. Le langage offre donc la possibilité de regrouper les actions électroniques dans une structure hiérarchique arbitrairement complexe. Ainsi dans tout ce qui suit, lorsqu'on parle des actions contenues dans un groupe, il est possible que certaines d'entre elles soient également des groupes.

Synchronisation

Les structures de groupe permettent de regrouper un ensemble d'actions. Il est alors possible de caractériser la manière dont cet ensemble se synchronise avec les musiciens. Il existe à l'heure actuelle deux comportements distincts : `loose` et `tight`.

- `loose` correspond au cas le plus simple, les actions se synchronisent sur la vitesse estimée, ou tempo, des instrumentistes. Le seul événement important est l'événement déclencheur auquel est attaché le groupe.
- `tight` est un peu plus subtil. Au moment ou le groupe est lancé, le système calcule à quel événement est associé chacune des actions qui le compose. Celles-ci ne seront lancées qu'une fois l'événement correspondant détecté. Tous les événements instrumentaux ont donc la même importance.

Dans le cas d'une performance parfaite à tempo constant, il n'y a pas de différence entre les deux comportements. En revanche, si les instrumentistes ont une interprétation un peu plus souple, il se peut que les événements instrumentaux arrivent plus tôt, ou plus tard que ce qui est prévu par le tempo courant. Dans ce cas on observe une réelle différence de comportement entre un groupe `loose` et un groupe `tight`.

Le premier n'est touché par ces variations de vitesse que dans une moindre mesure du fait de l'inertie de l'estimation du tempo. Le second en revanche restera au plus près de l'interprétation des musiciens. En particulier, la simultanéité des actions et des événements sera préservée. Ce comportement est donc particulièrement bien adapté dans un contexte d'accompagnement automatique.

Rattrapage d'erreur

Lors d'une performance, il peut arriver qu'un instrumentiste manque un événement. Pire, il peut arriver qu'un groupe soit justement attaché à cet événement. Il existe, pour gérer ce genre de situation, deux attributs de groupe : `local` et `global`.

- Un groupe `local` correspond par exemple à un motif d'accompagnement. Si un événement n'est pas détecté les actions correspondantes ne sont pas déclenchées.

- Un groupe `global` est utile pour les actions qui sont capitales pour le bon déroulement de l'œuvre. Si un événement n'est pas détecté, les actions correspondantes sont lancées avec un délai nul lors de la détection de l'événement suivant.

Les quatre comportements

Les attributs de synchronisation et de rattrapage d'erreur peuvent être arbitrairement combinés. Il se dégage donc quatre comportements spécifiés de la manière suivante [7] :

- `loose-local` : si l'événement associé au début du groupe e_1 n'est pas détecté, alors aucune des actions du groupe n'est lancée.
- `loose-global` : si l'événement associé au début du groupe e_1 n'est pas détecté, alors le groupe est lancé, avec un délai nul, dès la détection d'un événement postérieur e_2. Les rapports temporels entre les actions sont conservés.

Un groupe `loose` peut donc être vu comme une entité atomique. En cas d'erreur, c'est l'ensemble du corps du groupe qui est affecté. Soit l'ensemble du groupe est *translaté* sur l'événement suivant, soit il est tout simplement ignoré. Enfin, seule une erreur sur l'événement déclencheur affecte le comportement du groupe.

- `tight-local` : si un événement associé au groupe e_1 n'est pas détecté ; alors les actions associées à cet événement sont ignorées ; les actions du groupe sont lancées comme prévu initialement si leur position est après l'événement détecté suivant : e_2.
- `tight-global` : si un événement associé au groupe e_1 n'est pas détecté ; alors les actions associées à cet événement sont lancées avec un délai nul dès que l'événement suivant e_2 est détecté. Les actions du groupe situées après l'événement e_2 sont lancées comme prévu initialement.

À l'inverse du groupe `loose`, une erreur de l'instrumentiste n'affecte qu'un sous-ensemble du corps d'un groupe `tight` : celui attaché à l'événement manqué. On peut également remarquer que lors de l'exécution d'un groupe `tight`, le futur ne dépend pas des erreurs du passé. Si l'instrumentiste se trompe, la fin de la performance sera la même que si l'instrumentiste avait joué parfaitement du début à la fin.

2.3 Indépendances des attributs

Les quatre comportements précédents présentent l'avantage de regrouper de manière concise de nombreuses réalités musicales. Le langage tel qu'il est décrit dans la section 2.2 a ainsi été utilisé avec succès à l'occasion de nombreux concerts.

Pour autant on peut regretter que certains attributs couvrent d'un même nom deux comportements distincts. Par exemple le sens de l'attribut `global` change selon la stratégie de synchronisation adoptée.

Dans cette section nous essayons de spécifier de manière indépendante la gestion des erreurs des instrumentistes d'une part, et les stratégies de synchronisation d'autre part. Ceci pourra peut être faciliter la conception de nouvelles stratégies de synchronisation ou de gestion des erreurs dans le futur.

Synchronisation et réécriture

Pour caractériser les deux types de synchronisation, nous introduisons la notion de point de contrôle. Les points de contrôle sont les événements instrumentaux sur lesquels se synchronise un groupe. Un

groupe `loose` ne possède, par exemple, qu'un unique point de contrôle : l'événement déclencheur. À l'inverse, si le groupe est `tight`, l'ensemble des événements instrumentaux qui doivent être joués simultanément deviennent des points de contrôles.

On peut alors remarquer qu'un groupe `tight` peut être décomposé en un ensemble de groupes `loose`. Tout se passe comme si, lors du lancement d'un groupe `tight`, le système associait un groupe `loose` à chacun des points de contrôle. Ces groupes représentent l'ensemble des actions comprises entre deux événements instrumentaux consécutifs. Ils ne se chevauchent donc pas. Le processus de réécriture est illustré par la figure 2.

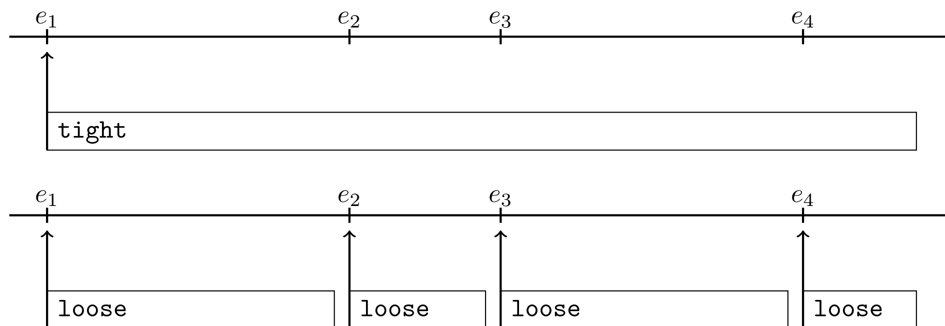


FIGURE 2 – Conversion d'un groupe `tight` en un ensemble de groupes `loose`.

La stratégie `tight` peut donc être vue comme un raccourci syntaxique pour un ensemble de groupes `loose`. Il ne reste alors plus qu'à définir les stratégies de gestion des erreurs dans le cas de la stratégie `loose`. L'indépendance des deux types d'attributs est ainsi assurée par construction. Nous verrons par la suite que cette vision permet de retrouver l'ensemble des comportements désirés.

La structure de groupe `loose` devient ainsi l'atome de base. On peut dès lors très facilement imaginer de nouvelles stratégies de synchronisation. On pourrait, par exemple, synchroniser un groupe sur les temps forts, ou sur tous les do #. Synchroniser revient à répartir des points de contrôle à l'intérieur d'un groupe. En d'autres termes, une stratégie de synchronisation est une manière de découper un groupe en un ensemble de groupes `loose`. Cette vision de la synchronisation rend donc le langage plus modulable.

`global` et `local`

Prenons, pour commencer, le sens de `local` et `global` tel qu'il est défini par les comportements `loose-local` et `loose-global`.

- `local` : si l'événement associé au début du groupe e_1 n'est pas détecté, alors aucune des actions du groupe n'est lancée.
- `global` : si l'événement associé au début du groupe e_1 n'est pas détecté, alors le groupe est lancé dès la détection d'un événement postérieur e_2. Les rapports temporels entre les actions sont conservés.

On retrouve, de manière évidente les comportements `loose local` et `loose global`. De même le comportement `tight-local` induit par la réécriture en ensemble de groupes `loose` correspond bien à celui attendu. Si un point de contrôle est manqué, les actions qui lui sont associées sont tout simplement ignorées.

En revanche, on comprend ici que l'attribut `global` ne prend pas le même sens selon qu'il est attaché au comportement `tight-global`, ou `loose-global`. Si on conserve le sens de `global` tel qu'il est défini ci-dessus, les groupes caractérisés par les deux attributs `tight` et `global` auraient un comportement inédit.

Ainsi, si un point de contrôle est manqué, le groupe d'actions qui lui est associé sera lancé, avec un délai nul, dès la détection du prochain événement. Les rapports temporels entre les actions au sein de ce groupe seront conservés. Ainsi, en cas d'erreur, les groupes `loose` issus de la réécriture d'un groupe `tight` peuvent être amenés à se superposer. L'ordre entre les actions au sein du groupe n'est donc plus garanti. La figure 3 illustre ce comportement.

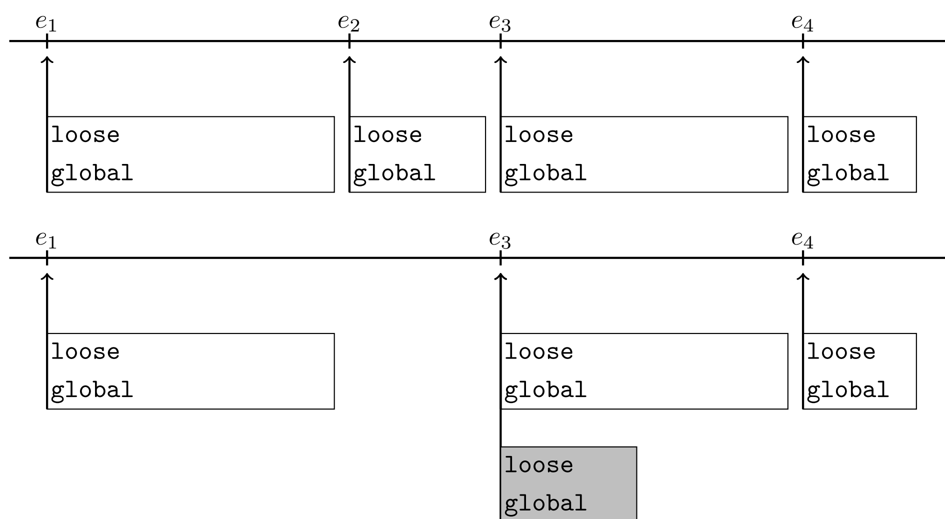


Figure 3 – Illustration de l'attribut `global`. En haut, une partition. En bas, l'exécution de cette partition dans le cas où le deuxième événement est manqué. Le même comportement serait observé si la partition ne contenait qu'un unique groupe `tight global`.

Il n'est pas évident que ce comportement corresponde à une réalité musicale. Cependant, c'est une conséquence logique des spécifications du système.

`causal` et `partial`

Intéressons nous maintenant au sens de l'attribut `global` tel qu'il est défini par le comportement `tight-global`. Encore une fois, la notion de point de contrôle permet de décrire aisément ce comportement : Si un point de contrôle est manqué, les actions qui lui sont associées sont lancées avec un délai nul dès la détection du point de contrôle suivant.

Ce comportement s'étend assez naturellement aux groupes `loose`. Si le point de contrôle est manqué, on découpe le groupe en deux. Les actions normalement antérieures à l'événement détecté sont immédiatement lancées avec un délai nul. Les actions postérieures à cet événement forment alors un nouveau groupe `loose` qui est, lui aussi, immédiatement lancé. En d'autres termes, le point de contrôle de ce nouveau groupe est l'événement détecté qui suit immédiatement l'erreur. La figure 4 illustre ce comportement.

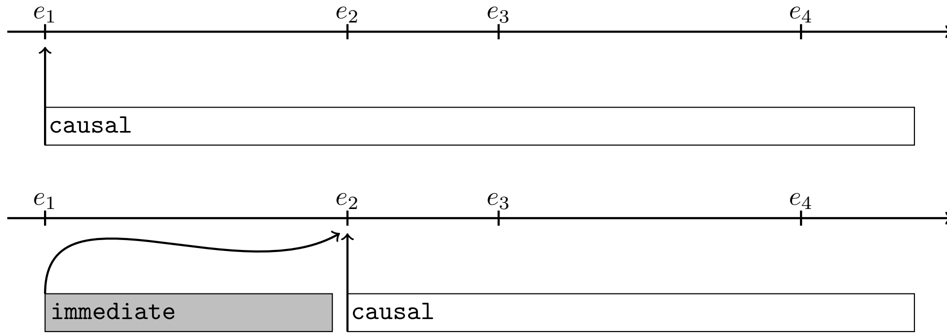


Figure 4 – Illustration de l'attribut `causal` dans le cas d'un groupe `loose`. En haut, une partition. En bas, l'exécution de cette partition dans le cas où le premier événement est manqué. Le mot clef `immediate` indique que toutes les actions contenues dans le bloc sont lancées avec un délai nul.

Si maintenant, on considère un groupe `tight` comme un ensemble de groupes `loose`, on retrouve bien le comportement `tight-global`. En effet, dans ce cadre, les groupes `loose` sont toujours strictement compris entre deux événements. Ainsi si le point de contrôle est manqué, il n'y a pas d'action postérieure à l'événement détecté. Les actions attachées au point de contrôle sont donc, tout simplement, lancées avec un délai nul.

Nous verrons dans la section 2.4 que ce comportement permet de préserver un ordre partiel sur les actions. Pour cette raison, on nomme l'attribut correspondant `causal`.

Le comportement d'un groupe `causal` fait apparaitre, presque naturellement, un dernier attribut. Il peut en effet être commode, en cas d'erreur, d'ignorer les actions qui précédent normalement l'événement détecté. C'est en quelque sorte un intermédiaire entre les comportements `causal` et `local`. On le nomme `partial` car il indique, en cas d'erreur, une exécution partielle du contenu d'un groupe. Notons que dans le cas d'un groupe `tight`, `local` et `partial` sont équivalents du fait de l'absence de chevauchement entre les groupes issus de la réécriture.

Les comportements induits par les attributs `causal` et `partial` partagent une propriété remarquable : *le futur ne dépend pas des erreurs du passé*. En effet le fait de manquer un point de contrôle ne change pas le comportement du système pour la fin de performance. En d'autres termes ces attributs certifient que la performance retombera toujours sur ses pieds. Ce sont, en ce sens, de véritables attributs de rattrapage d'erreur.

A l'inverse, les comportements spécifiés par les attributs `global` et `local`, peuvent, en cas d'erreur, considérablement altérer la fin de la performance. Un groupe peut ainsi se retrouver associé à un événement imprévu ou bien être tout simplement ignoré. Le résultat n'a alors plus grand rapport avec celui d'une performance idéale.

Nous sommes donc passés de quatre attributs possibles à six. Quatre attributs de gestion des erreurs et deux de synchronisation. Cela étant, les attributs de gestion d'erreur sont maintenant indépendants de la stratégie de synchronisation adoptée. Par ailleurs, il est très probable que les attributs `causal` et `partial` correspondent à une véritable réalité musicale.

Il y a donc maintenant huit comportements possibles. La figure 5 résume ces diverses possibilités.

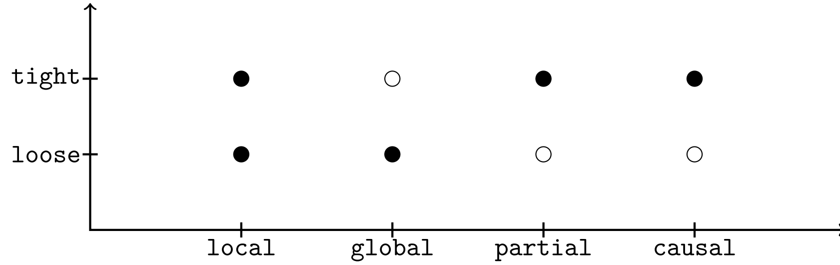


Figure 5 – Illustration des différents comportements possibles.
Les cercles noirs correspondent aux comportements déjà présents dans le langage original.

2.4 Analyse de causalité

Nous nous intéressons ici à la stratégie de rattrapage d'erreur `causal`. Le but de cette section est de montrer formellement que cette stratégie préserve un ordre partiel sur les actions. On montre ainsi que sous certaines conditions, on peut assurer au compositeur qu'une action a sera toujours exécutée avant une action b quelles que soient les erreurs commises par l'interprète.

Cette propriété est fondamentale en programmation synchrone. En effet, il peut arriver qu'un événement futur dépende du comportement d'un événement passé. Il est alors nécessaire de garantir, que, lors de l'exécution, le premier événement précédera toujours le second. On dit alors qu'il y a une relation de causalité entre ces événements.

Notations

On représente une partition par un couple $(\mathcal{A}, \mathcal{E})$. $\mathcal{A}$ représente l'ensemble des actions électroniques et $\mathcal{E}$ l'ensemble des événements instrumentaux. Dans ce qui suit, on appellera plus génériquement *événement* un élément de l'ensemble $\mathcal{A} \cup \mathcal{E}$.

Quitte à passer par une étape de réécriture, on ne considère ici que des groupes `loose`.

Une action a appartenant à un groupe `causal` peut être caractérisée par un couple $(e, d) \in \mathcal{E} \times \mathbb{R}$ où e correspond à l'événement déclencheur du groupe et d au délai de l'action au sein du groupe. Ainsi, $a : (e, d) \in \mathcal{A}$ signifie $s(a) = s(e) + d$.

Pour *Antescofo*, les événements n'ont pas de durée. On associe donc à un événement instrumental la date de sa détection, et à une action électronique la date de son exécution.

Soit $s : \mathcal{A} \cup \mathcal{E} \longrightarrow \mathbb{R}^+$ la fonction de datation de la partition. Cette fonction associe à chaque événement la date qu'il aurait lors d'une performance idéale. De la même manière on pose $p : \mathcal{A} \cup \mathcal{E} \longrightarrow \mathbb{R}^+$ la fonction de datation de la performance. Ces deux fonctions ne renvoient pas la date physique, mais une date relative au tempo. Nous n'avons pas besoin ici de prendre en compte les variations de tempo.

Enfin, soit $(e_1, e_2) \in \mathcal{E}^2$, $e_1 \longrightarrow e_2$ signifie que e_2 est l'événement qui succède immédiatement e_1 lors de la performance.

Attribut `causal` et ordre partiel

Nous supposons, que les événements instrumentaux qui ne sont pas manqués sont joués en rythme. En d'autres termes, $\forall e \in \mathcal{E}$, si e n'est pas manqué, alors $p(e) = s(e)$. On suppose en outre que l'instrumentiste

joue le premier et le dernier événement de la partition. Sans changer l'esprit de la preuve, ceci nous permet d'éviter les cas particuliers causés par les bornes de la partition. Ceux-ci ne présentent, par ailleurs, pas de difficulté notable.

On peut dans ce cadre formaliser la stratégie `causal` de la façon suivante :

Définition 1. Soit $a : (e, d) \in \mathcal{A}$ une action, appartenant à un groupe `causal`.
Si $\exists (e_1, e_2) \in \mathcal{E}^2$ tel que :

$$\begin{cases} e_1 \longrightarrow e_2 \\ s(e_1) < s(e) < s(e2) \\ s(a) = s(e) + d < s(e2) \end{cases}$$

alors $p(a) = s(e_2)$,
sinon $p(a) = s(a)$

Les deux conditions $e_1 \longrightarrow e_2$ et $s(e_1) < s(e) < s(e2)$ indiquent que l'événement e est manqué. Si tel est le cas, deux cas se présentent. Si l'action considérée est postérieure à l'événement détecté, elle conserve la même date. Dans le cas contraire, l'action est lancée immédiatement et prend donc la date de cet événement.

On déduit immédiatement de cette définition les deux propriétés suivantes :

Propriété 1. $\forall a \in \mathcal{A},\ p(a) \geq s(a)$

Propriété 2. $\forall a \in \mathcal{A},\ p(a) > s(a)$ *si, et seulement si,* $\exists (e_1, e_2) \in \mathcal{E}^2$ *tel que :*

$$\begin{cases} e_1 \longrightarrow e_2 \\ s(e_1) < s(e) < s(e_2) \\ s(a) = s(e) + d < s(e_2) \end{cases}$$

On définit par ailleurs un ordre partiel sur les actions :

Définition 2. Soient $a_1 : (e_1, d_1)$ et $a_2 : (e_2, d_2)$ deux actions.

$$a_1 \prec a_2 \Leftrightarrow \begin{cases} s(e_1) \leq s(e_2) \\ s(e_1) + d_1 < s(e_2) + d_2 \end{cases}$$

La relation de causalité se traduit enfin par le théorème suivant :

Théorème 1 (Causalité)**.** *Soit* $(a_1, a_2) \in \mathcal{A}^2$ *deux actions appartenant à des groupes de stratégie* `causal`*.*

$$a_1 \prec a_2 \Rightarrow p(a_1) \leq p(a_2)$$

Démonstration. Si a_1 et a_2 appartiennent au même groupe c'est immédiat.

Supposons qu'il existe $(e_1, e_2) \in \mathcal{E}^2$ et $(d_1, d_2) \in \mathbb{R}^2$ tels que :

$$\begin{cases} s(e_1) < s(e_2) \\ a_1 : (e_1, d_1) \\ a_2 : (e_2, d_2) \end{cases}$$

D'après les propriétés 1 et 2, on peut distinguer deux cas :

- Si $p(a_1) = s(a_1)$ on a :

$$p(a_1) = s(a_1) = s(e_1) + d_1 < s(e_2) + d_2 = s(a_2) \leq p(a_2)$$

- Si $p(a_1) > s(a_1)$ alors d'après la propriété 2, $\exists (e_1^1, e_2^1) \in \mathcal{E}^2$ tel que :

$$\begin{cases} e_1^1 \longrightarrow e_2^1 \\ s(e_1^1) < s(e_1) < s(e_2^1) \\ s(a_1) = s(e_1) + d_1 < s(e_2^1) \end{cases}$$

  On peut ici encore distinguer deux cas :
  - Si $s(e_2^1) \leq s(a_2)$ on a :

$$p(a_1) = s(e_2^1) \leq s(a_2) \leq p(a_2)$$

  - Si $s(e_2^1) > s(a_2)$ on a :

$$\begin{cases} e_1^1 \longrightarrow e_2^1 \\ s(e_1^1) < s(e_2) < s(e_2^1) \\ s(a_2) = s(e_2) + d_2 < s(e_2^1) \end{cases}$$

    donc par définition de p, $p(a_2) = s(e_2^1) = p(a_1)$

□

Il est facile d'étendre cette propriété de causalité au cas ou a_2 appartient à un groupe de stratégie `global`. En effet, il est évident qu'une action appartenant à un un groupe g sera toujours exécutée plus tôt si g est `causal` que s'il est `global`.

L'outil *ascoVerif*

Toutes ces considérations nous ont permis de développer un outil de vérification relativement simple. On donne en entrée, deux actions a_1 et a_2. Une première analyse temporelle permet d'associer à chacune de ces actions, un événement déclencheur et un délai. En particulier, cette analyse commence par réécrire l'ensemble des groupes `tight` sous forme de groupes `loose`. On associe en outre à chacune des actions l'attribut de gestion des erreurs du groupe qui les contient. Par convention, les actions directement rattachées à un événement sont considérées comme `local`.

Il existe trois cas, qui ne permettent pas de conclure.

- Si une des actions est `local` ou `partial`, il peut arriver qu'elle soit manquée.
- Si a_1 est `global`, on ne peut prévoir, en cas d'erreur, sa date d'exécution.
- Si $a_1 \not\prec a_2$, la causalité ne peut être garantie.

Enfin si $a_1 \prec a_2$, que a_1 est `causal` et a_2 `causal` ou `global`, on peut assurer que a_1 sera toujours exécutée avant a_2.

ascoVerif prend donc en entrée deux actions et détermine si la propriété de causalité peut être garantie pour ces deux actions. Dans le cas contraire, il indique pourquoi cette propriété ne peut être assurée.

2.5 Formalisation et ensemble de types

Les langages que nous avons étudiés : *Reactive ML* et *Lucid Synchrone* sont tous deux des langages dit *à la ML*. Il est donc aisé de formaliser la structure de la partition par un ensemble de type. Cette formalisation découle tout naturellement de la grammaire présentée dans la section 2.2

```
type delay = float
type label = int
type tempo = float

(** Atomic action for Asco *)
type message = unit -> unit
(** Synchronization strategies *)
type sync = Tight | Loose
(** Error handling strategies *)
type err = Local | Global | Partial | Causal

(** Basic action *)
type action =
   { action_delay : delay;
     action_body : message; }

(** Asco group *)
type group =
     { group_delay : delay;
       group_synchro : sync;
       group_error : err;
       group_body : asco_event list; }

(** Generic asco event *)
and asco_event = Empty | Group of group | Action of action

(** Electronic score event *)
type score_event =
   { event : label;
     body : asco_event list; }

(** Instrumental score *)
type instr_score = delay list
(** Electronic score *)
type score = (score_event list)*(instr_score)
```

- `instr_score` contient les dates, relativement au tempo, des événements instrumentaux attendus.
- `score_event` contient l'ensemble des actions électroniques associées à un événement instrumental. Il est caractérisé par le `label` de l'événement auquel il se rattache et contient une liste de `asco_event`
- `asco_event` est le type générique pour toute action électronique. Il peut représenter une action atomique ou un groupe qui peut, à son tour, contenir des actions etc.
- Les actions atomiques sont caractérisées par un délai et un message.
- Un groupe est caractérisé par un délai, un attribut d'erreur, un attribut de synchronisation et la liste des `asco_event` qu'il englobe.

3 Un interprète *Antescofo* en Reactive ML

Nous décrivons dans cette section l'implémentation d'un interprète du langage d'*Antescofo* en *Reactive ML*. L'objectif de cette implémentation est double. Elle permet d'une part de répondre à la question : Les langages synchrones sont-ils adaptés aux différentes structures du langage d'*Antescofo* ? D'autre part, RML est un langage qui possède une sémantique claire et bien définie [13]. L'implémentation de l'interprète permet donc de donner une sémantique par traduction aux différentes notions du langage d'*Antescofo* : groupe, synchronisation, gestion des erreurs...

3.1 Le langage *Reactive ML*

Reactive ML [16] est un langage synchrone conçu comme une extension du langage fonctionnel OCaml. Ce langage allie ainsi la puissance du formalisme fonctionnel avec l'expressivité temporelle des langages synchrones. Des langages fonctionnels, RML hérite l'inférence de type, la récursivité et l'ordre supérieur. À cela s'ajoute les processus, fonctions *du temps*, et les signaux qui permettent la communication entre processus.

La notion d'instant

Reactive ML s'appuie sur la notion d'instant ou pas de temps. Un processus décrit simplement une suite d'instructions à effectuer à chaque instant. Au cours de l'exécution, tous les processus sont synchrones : ils partagent le même temps. En d'autres termes, les processus s'attendent mutuellement avant de passer, ensemble, à l'instant suivant. On peut ainsi considérer que les calculs réalisés au sein d'un même pas d'exécution sont instantanés. C'est l'hypothèse synchrone.

L'utilisation de cette notion d'instant se fait par l'intermédiaire du mot-clef `pause`. Il permet d'indiquer à un processus d'attendre l'instant suivant pour continuer. L'exemple suivant [1] permet de mieux cerner cette notion.

```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre

let rec process iter_largeur f a =
  pause;
  match a with
    | Vide -> ()
    | Noeud(x,g,d) ->
      f x;
      (run (iter_largeur f g) || run (iter_largeur f d));
```

Le type `'a arbre` permet de définir un arbre binaire générique (`'a` signifie que l'étiquette d'un nœud peut être de n'importe quel type). On considère ici un parcours d'arbre en largeur d'abord. Le processus `iter_largeur` applique donc la fonction `f`, étage par étage, à chacun des nœuds de l'arbre. La difficulté vient du fait que la structure d'arbre ne fait pas de lien explicite entre deux nœuds *frères*. Dans un langage de programmation classique il faut retrouver ce lien ce qui nécessite un peu d'astuce.

1. Cours de RML de Louis Mandel, MPRI

En revanche en RML l'écriture est transparente. On commence par attendre que tous les nœuds de l'étage en cours soient évalués. C'est le sens du mot `pause`. On peut ensuite s'intéresser à l'étage suivant. S'il est vide, on ne fait rien. Sinon on peut évaluer le nœud courant puis parcourir en largeur, et en parallèle, le fils droit et le fils gauche. On utilise pour signifier cela le mot `run` et le symbole `||`.

Lors de la compilation, l'utilisateur peut décider de donner une épaisseur fixe à ces instants. On définit ainsi une horloge de base qui sert de référence à tous les processus. Il devient alors très facile d'utiliser cette horloge pour évaluer des durées. Par exemple, le processus suivant se contente d'attendre une durée `dur` en seconde.

```
let clock_step = 0.001

let process wait_abs dur =
  let d = int_of_float (dur/.clock_step) in
    for i = d downto 1 do pause; done;;
```

Ici, `clock_step` est une variable globale qui définit la durée d'un pas de temps. Il faut ensuite passer cette constante en argument au compilateur pour obtenir le comportement voulu.

Les signaux

Les signaux sont des canaux de communication pour les processus. Ceux-ci sont caractérisés par une valeur par défaut et une fonction d'accumulation. Par exemple, `signal s default 0 gather (+)` définit un signal `s` qui vaut 0 par défaut. À un instant donné, la valeur du signal est la somme des valeurs émises sur le signal `s` au cours de l'instant précédent. Ceci permet d'éviter les problèmes de lecture et d'écriture concurentes.

Un fois ces signaux définis, un processus peut soit émettre une valeur, soit récupérer la valeur du signal à la fin de l'instant précédent. Pour émettre une valeur `value` sur un signal `s`, il suffit d'écrire `emit s value`. À l'opposé, il existe plusieurs manières de récupérer la valeur du signal :

- `pre ?s` : renvoie la valeur du signal à l'instant précédent ou la valeur par défaut si aucune valeur n'a été émise sur ce signal.
- `last ?s` : renvoie la dernière valeur émise sur le signal `s`.

Enfin, il existe plusieurs structures qui permettent de contrôler un processus à l'aide de signaux :

- `await s; p` : attend une émission sur `s` pour exécuter `p`.
- `await s (v) when (v == value) in p` : attend que `s` prenne la valeur `value` pour exécuter `p`.
- `do body until s -> other done` : exécute `body` tant qu'il n'y a pas d'émission sur le signal `s`. Dans le cas contraire, c'est `other` qui est exécuté.

La simplicité de ces diverses structures de contrôle alliée à leur grande expressivité fait de *Reactive ML* un langage parfaitement adapté à la simulation de systèmes réactifs complexes. Il a été utilisé dans ce but pour la simulation de grands reseaux de contact [17]. Pour plus de précision sur la sémantique du langage on pourra se reporter à la thèse de Louis Mandel [18].

3.2 Fonctionnement de l'interprète *Antescofo*

Le système réactif dispose de quatre entrées :

- Un signal pour les événements instrumentaux détectés : `ext_event`,
- Un signal pour le tempo : `bpm`,
- La partition électronique traduite grâce à notre système de type (cf. section 2.5),
- La partition instrumentale qui donne la date attendue pour chacun des événements en fonction du tempo.

Nous considérons ici le langage tel qu'il est décrit dans la section 2.3.

Canaux de communication

Nous avons fait le choix de séparer au maximum les signaux de communication interne de ceux nécessaires pour dialoguer avec l'extérieur. La séparation de ces deux univers, interne et extérieur, se fait grâce au processus `follow`. Celui-ci permet de séparer le signal extérieur `ext_event` en deux signaux de communication interne : `event` et `missed_event`. Le premier correspond au dernier événement détecté, et le second contient la liste des événements manqués entre le nouvel événement et celui qui le précédait.

```
let process follow ext_event event missed_event =
  loop
    await ext_event (ev) in
    let previous_ev = last ?event in
    emit event (ev);
    let n = ev - previous_ev - 1 in
    if n>0 then
      begin
        for i=previous_ev + 1 to ev -1 do
          emit missed_event i;
        done
      end
  end
```

Nous avons ainsi transformé les deux signaux d'entrée, `ext_event` et `bpm` en trois : `event`, `missed_event`, et `bpm`. Ceci nous permettra plus tard de donner une véritable sémantique aux diverses stratégies de synchronisation et de gestion des erreurs.

Suivi de tempo

Le but d'antescofo est d'envoyer des messages de contrôle pour accompagner un instrumentiste. La base de l'interprète est donc un processus ayant le comportement suivant : *Attendre un delai* `d` *puis envoyer le message* `m`. Mais, sur une partition, les délais sont spécifiés relativement au tempo de l'instrumentiste. Le système doit donc être capable de suivre ces variations de tempo et donc de dilater ou contracter, si besoin est, ces délais.

Ce travail de suivi de tempo est réalisé par le processus `wait_rel`. Celui-ci attend une durée `dur` relativement à un tempo `bpm` en battement par minute. Une durée de 1 temps au tempo 120 équivaut ainsi à une demi seconde.

```
let process wait_rel dur bpm =
  signal prop default (1,1) gather (fun x y -> x) in
  let rec process aux prop =
    let b = last ?bpm in
    let (p,q) = last ?prop in
    let abs_dur = dur *. (60. /. b) in
    let nb_steps = int_of_float (abs_dur /. clock_step)-1 in
    let n = (p * nb_steps) / q in
    do
      for i = n downto 1 do
        emit prop (i*p-1,n*q);
        pause
      done;
    until bpm
      -> run (aux prop)
    done
  in run (aux prop);;
```

Le tempo est ici représenté par un signal. La valeur du tempo peut ainsi être modifiée au cours de l'attente. Dès lors le processus doit être capable de recalculer le temps qu'il lui reste à attendre. Il est donc nécessaire de garder en mémoire la proportion de la durée qu'il reste à attendre : p/q. On utilise pour cela le signal `prop` qui contient un couple : (p, q). On commence par calculer le nombre d'instant qu'il faut attendre en fonction du tempo courant. Ensuite, tant que le tempo ne change pas, on se contente de mettre à jour la proportion de la note qu'il reste à attendre grâce au signal `prop`. Dans le cas contraire, on recalcule la durée qu'il reste à attendre en terme de nombre de pas et on recommence.

Processus d'exécution

À chaque type de structure, `group loose`, `group tight` ou `action`, est associé deux processus : `exec_detected` et `exec_missed` qui ne dépendent que du tempo. Le premier est lancé si l'événement déclencheur est bien détecté, le second est utilisé si cet événement est manqué.

Ainsi, pour exécuter une action électronique `ae` associée à un événement instrumental `ev`, il suffit d'écouter en parallèle les deux canaux de communication interne. Si `ev` est reçu sur canal `event`, on lance le processus `exec_detected`. S'il est reçu sur le canal `missed_event`, on lance le processus `exec_missed`. On peut remarquer que RML nous permet d'écrire ce comportement de façon remarquablement transparente et concise.

```
let process exec ae ev =
  (await event (e) when (e == ev) in
  exec_detected ae)
  ||
  (await missed_event (mev) when (List.mem ev mev) in
  exec_missed ae)
```

La principale difficulté est due au fait que les différentes structures peuvent être arbitrairement imbriquées. Les processus d'exécution doivent donc être définis de manière mutuellement récursive.

1. Si `ae` est une action :
   - `exec_detected` attend le delai associé à l'action puis envoie le message correspondant.
   - `exec_missed` ne fait rien. Par convention, les actions directement attachées à un événement instrumental sont considérées comme locales.
2. Si `ae` est un groupe `loose` :
   - `exec_detected` attend le délai associé au groupe puis appelle `exec_detected` sur les actions qui composent le corps du groupe. Ainsi, les actions du corps du groupe ne se synchronisent que sur le tempo.
   - `exec_missed` dépend de la stratégie de gestion des erreurs adoptée.
     Si le groupe est `local` on ne fait rien.

     Si le groupe est `global`, on appelle `exec_detected` sur les actions qui composent le corps du groupe sans attendre le délai associé au groupe. L'ensemble est donc rattaché à l'événement détecté suivant et conserve intact les rapports temporels entre les actions.

     Si le groupe n'est ni `global`, ni `local`, on sépare le corps du groupe en deux catégories d'actions. Les actions normalement antérieures à l'événement détecté : `previous_actions`, et celles qui lui sont postérieures : `next_actions`. On soustrait alors la date idéale de l'événement détecté aux délais des actions de `next_actions`. Ainsi, ces actions se produiront à la date prévue par la partition.

     Si le groupe est `partial`, on ignore les actions de `previous_actions`, et on appelle `exec_detected` sur les éléments de `next_actions`. Les actions qu'on ne peut rattraper sont donc ignorées, les autres se produiront à la date prévue.

     Si le groupe est `causal`, on commence par fixer à zéro le délai des actions de `previous_actions`. On appelle ensuite `exec_detected` sur l'ensemble des éléments du corps du groupe. De cette manière les actions qu'on ne peut rattraper sont lancées immediatement. Les autres seront exécutées à la date prévue par la partition.
3. Si `ae` est un groupe `tight` :
   - `exec_detected` attend le délai associé au groupe puis calcule pour chaque élément du corps du groupe l'événement instrumental auquel il doit se rattacher. Le délai de chaque action est mis à jour en conséquence. On appelle ensuite `exec` sur chacun de ces éléments. Ainsi le corps du groupe se synchronise sur le tempo, et sur les événements instrumentaux.
   - `exec_missed` calcule pour chaque élément du corps du groupe l'événement instrumental auquel il doit se rattacher. Le délai de chaque action est mis à jour en conséquence. On peut alors séparer le corps du groupe en deux catégories. D'un côté les actions qui auraient déjà dû se produire : `previous_actions`, et de l'autre celles associées à un événement instrumental futur : `next_actions`.

Si le groupe est `local` ou `partial`, on ignore `previous_actions` et on appelle `exec` sur les éléments de `next_actions`. De cette manière, les actions associées à un événement instrumental manqué sont ignorées, les autres attendent la détection de leur événement déclencheur pour s'exécuter

Si le groupe est `global`, on appelle `exec_detected` sur les éléments de `previous_actions`, et `exec` sur les éléments de `next_actions`. Ainsi, les actions associées à un événement instrumental manqué sont immédiatement lancées en conservant intact leurs rapports temporels. Les autres attendent la détection de leur événement déclencheur pour s'exécuter.

Si le groupe est `causal`, on commence par fixer à zéro le délai des éléments de `previous_actions`. On appelle ensuite `exec_detected` sur ces éléments, et `exec` sur les actions de `next_actions`. Les actions associées à un événement instrumental manqué sont donc immédiatement lancées, les autres attendent la détection de leur événement déclencheur pour s'exécuter.

Enfin, pour exécuter une partition, on appelle le processus `exec` sur l'ensemble des actions qui sont directement attachées à un événement instrumental. En d'autres termes, il suffit d'appeler le processus `exec` sur les actions qui forment le corps de chacun des `score_event` (cf. section 2.5).

La sémantique de chacune des structures est donc donnée par deux processus : `exec_detected` et `exec_missed`. En outre, les processus `exec_missed` permettent de donner un sens aux stratégies de gestion des erreurs. RML nous permet donc de traduire de manière quasiment transparente les spécifications décrites dans la section 2.3. Nous avons donc pourvu le langage d'*Antescofo* d'une véritable sémantique par traduction.

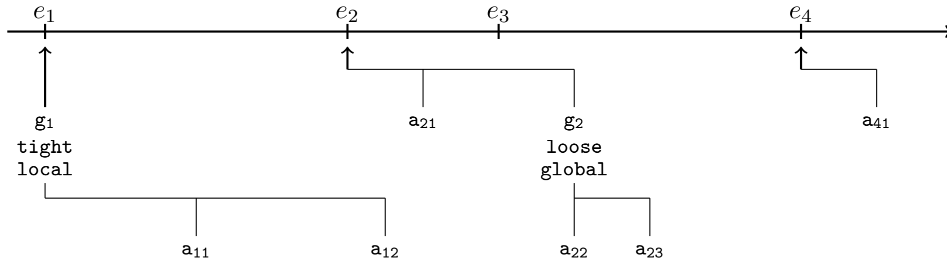
Arbre de processus

Un des grands avantages du langage RML est qu'il permet de gérer facilement la création dynamique de processus. Le programme décide au moment de son exécution quels processus doivent être exécutés. Cet aspect dynamique nous permet de gérer facilement les variations de tempo et la gestion des erreurs.

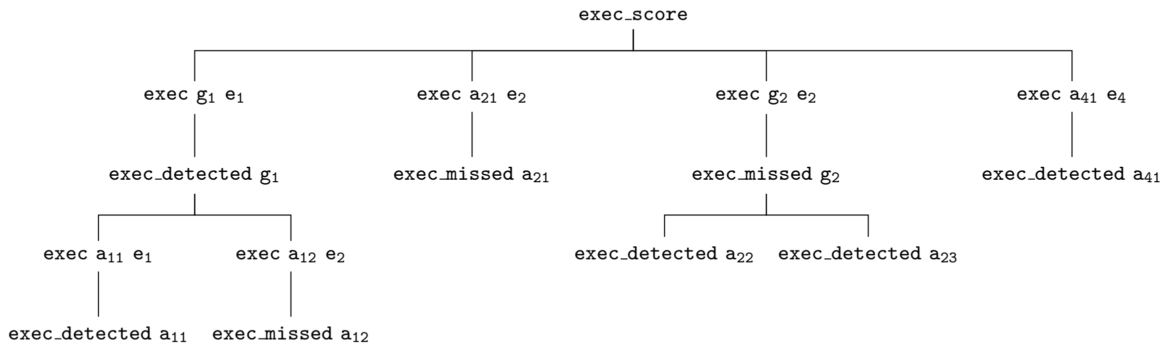
Prenons par exemple l'exécution d'une partition *Antescofo*. Au début de l'exécution on ne lance que des processus `exec`. Ensuite, en fonction des événements détectés, et de la nature des groupes et actions rencontrés, l'interprète décide s'il faut lancer des processus `exec`, `exec_detected` ou `exec_missed`. L'exécution d'une partition se traduit donc par un arbre de processus qui reflète à la fois la structure de la partition et la performance de l'instrumentiste. La figure 6 illustre l'exécution d'une partition simple.

Simulateur

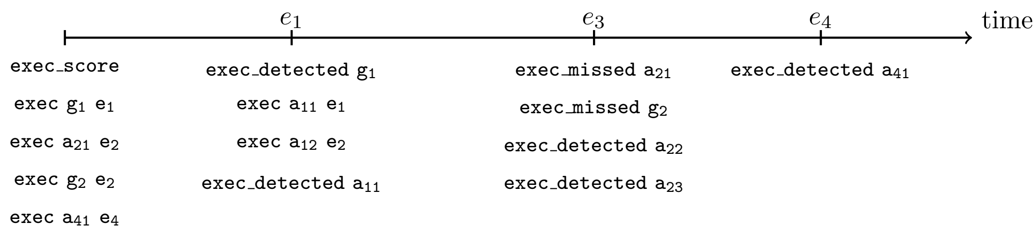
Dans le cadre de cette étude, nous ne nous intéressons qu'à la partie séquenceur d'*Antescofo*. Pour évaluer les performances de notre interprète, il faut donc, d'une manière ou d'une autre, simuler le comportement de la machine d'écoute. Nous avons donc intégré à l'interprète *Antescofo* un processus de simulation de performance. Celui-ci prend en entrée une liste de triplets : (ev, d, t) qui signifient : *Après d secondes, envoyer ev sur le canal* `ext_event` *et t sur le canal* `bpm`. Ce comportement se traduit de manière transparente en RML.



(a) Exemple de partition.



(b) Arbre des processus créés lors de la performance.



(c) Instants d'activation des différents processus au cours de la performance.

FIGURE 6 – Illustration de la création dynamique de processus lors de l'exécution d'une partition. On suppose ici que l'événement e_2 est manqué.

```
let rec process run_performance ext_event bpm perf =
  match perf with
    [] -> ()
  | (i,d,b)::t ->
      begin
        run (wait_abs d);
        emit ext_event i;
        emit bpm b;
      end
      ||
      run (run_performance ext_event bpm t)
```

3.3 Discussion

Reactive ML est un langage impératif. Il est donc bien adapté à une situation qui nécessite beaucoup de contrôle. L'aspect synchrone permet, en outre, de gérer facilement la programmation *temporelle*. On peut ainsi traduire de manière quasiment transparente des instructions telles que : *Attendre le signal s puis exécuter le processus p*. Ceci nous à permis de traduire naturellement les spécifications du langage d'*Antescofo*.

Ce langage permet en outre, la création dynamique de processus. Ainsi, lors de la détection d'un événement le système lance de manière récursive les fonctions d'exécution associées aux différentes actions. Lors de la performance, un *arbre de processus* reflétant la structure hiérarchique de la partition est ainsi créé dynamiquement. Cet aspect dynamique est fondamental pour un système tel qu'*Antescofo*. Cela permet de spécifier les délais par des expressions ou des variables qui ne sont calculées qu'au dernier moment. Ces valeurs peuvent donc dépendre de flux de données asynchrones tels que le tempo, les événements instrumentaux ou une pédale activée par le musicien. En cela notre système qui s'appuie sur la puissance expressive de RML est même en avance par rapport à l'*Antescofo* original.

Reactive ML était donc particulièrement bien adapté à notre problème.

Cela étant cet avantage peut, à terme, poser un problème de performance. Le système nécessite en effet la création de beaucoup de processus qui peuvent, dans le cas d'une partition conséquente, faire exploser le temps de calcul. En effet, RML ne donne pas de garantie sur la durée maximale d'un pas d'exécution, c'est une politique dite de *best effort*, le système fait ce qu'il peut. Il peut ainsi arriver qu'un des pas d'exécution requiert beaucoup plus de calcul, et donc de temps, que les précédents. Cela peut s'avérer gênant dans le cas d'une performance temps réel.

Notre système n'est donc pas appelé à remplacer *Antescofo*. Cependant, l'utilisation de *Reactive ML* nous a permis d'implémenter un système réactif semblable à *Antescofo* de manière élégante et concise (le coeur de l'interprète ne compte que 400 lignes de code). Notre système peut donc servir d'outil de vérification, ou de prototypage pour les constructions futures. Nous l'avons ainsi utilisé avec succès pour prototyper les stratégies de gestion d'erreur `partial` et `causal` qui n'existent pas dans le langage original (cf. section 2.3).

4 L'expérience *Lucid Synchrone*

Dans cette partie, nous explorons le paradigme de programmation synchrone dit *flot de données*. Ce paradigme offre plus de garanties sur l'éxécution d'un programme. En contrepartie, les aspets dynamiques sont bien plus difficiles à gérer. Il n'est donc plus question de traduire directement les spécifications des diverses structures du langage d'*Antescofo*. Pour contourner ce problème, nous avons choisi de ramener la partition *Antescofo* à sa plus simple expression.

Dans cette partie, après une courte présentation du langage, nous aborderons le problème de la normalisation d'une partition *Antescofo*. Puis nous détaillerons le fonctionnement de l'interprète *Lucid Synchrone* qui permet d'exécuter ces formes normales.

4.1 Le langage *Lucid Synchrone*

Lucid Synchrone [14] est un langage synchrone flot de données. Comme *Reactive ML*, *Lucid synchrone* (ou Lucy) est un langage *à la ML*, on retrouve donc les notions d'ordre supérieur, d'inférence de type et de récursivité. En outre ce langage permet à l'utilisateur de construire des programmes sous forme de combinaisons d'automates ce qui facilite grandement l'implémentation de certaines fonctions.

RML est un langage synchrone conçu comme une extension d'OCaml. On part donc d'un langage fonctionnel auquel on ajoute des constructions synchrones. À l'inverse, Lucy est un langage synchrone conçu dans le style de *Lustre* [19] qui intègre les notions d'ordre supérieur et de récursivité.

Notion de flot

À l'origine les langages flot de données s'inspirent d'un modèle de programmation synchrone, développé par Kahn [20], basé sur le formalisme utilisé par les ingénieurs de contrôle automatique. Un programme est appelé *nœud*. Il est déclaré avec des paramètres d'entrée et de sortie, et le corps du nœud définit la relation fonctionnelle qui existe entre les entrées et les sorties.

En d'autres termes un nœud transforme une suite infinie d'entrées en une suite infinie de sorties. Ces suites infinies sont nommées flots. Un programme peut donc être vu comme un bloc fixe qui transforme des flots de données.

Les flots sont les atomes de base du langage. La variable `x`, par exemple désigne le flot $x_0, x_1, x_2, \ldots$. De même 1 désigne le flot constant $1, 1, 1, \ldots$. Cette conception suppose un léger ajustement du système de typage des langages fonctionnels tel que OCaml. Ainsi, `int` ne désigne plus un entier mais une suite infinie d'entiers. En conséquence, les fonctions de base : $+, *, /, \ldots$ deviennent des applications point à point.

Il est également possible de manipuler les flots de données dans le temps en introduisant des délais unitaires. Ainsi `pre x` renvoie le flot `x` retardé d'un instant. Notons que `pre x` ne peut être défini au premier instant, il prend alors la valeur *nil*. Pour contourner ce problème, on peut utiliser l'opérateur `fby`. `x fby y` vaut x_0 au premier instant et y_{n-1} à l'instant n.

x	x_0	x_1	x_2	...
y	y_0	y_1	y_2	...
x * y	$x_0 * y_0$	$x_1 * y_1$	$x_2 * y_2$	...
pre x	nil	x_0	x_1	...
x fby y	x_0	y_0	y_1	...

Les horloges

À chaque flot est associé une horloge. Cette horloge est un booléen qui indique à chaque instant si le flot est défini ou non. Ceci permet de gérer des flots qui ne sont pas tous échantillonnés au même rythme. En conséquence, un programme doit vérifier un ensemble de contraintes. Il est par exemple impossible d'additionner deux flots définis sur des horloges différentes. En *Lucid Synchrone*, cette analyse est réalisée lors de la compilation. C'est ce qu'on appelle le *calcul d'horloge*.

Deux opérateurs permettent de faire communiquer des flots qui n'ont pas la même horloge :

- `when` est un échantillonneur qui permet à des processus rapides de communiquer avec des processus lents. Ainsi `x when b` est le flot qui prend la valeur de `x` quand le booléen `b` est vrai. L'horloge de ce flot est donc `b`.
- `merge`, à l'opposé, permet à des processus lents de communiquer avec des processus rapides en mélangeant deux flots suivant un flot booléen. Par exemple `merge x y b` est un flot qui prend la valeur de `x` quand le booléen `b` est vrai, et la valeur de `y` dans le cas contraire.

b	t	f	f	t	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
x when b	x_0			x_3	...
merge x y b	x_0	y_1	y_2	x_3	...

Dans notre cas, il est plus facile de considérer que tous les flots se basent sur une même horloge. On peut ainsi déclarer l'épaisseur d'un pas de temps comme une variable globale. Ceci nous permet d'exprimer très facilement les durées. Pour obtenir le comportement attendu, il faut ensuite passer cette variable en argument au compilateur.

Les signaux

Il y a, en *Lucid Synchrone*, une notion analogue aux signaux de RML. Pour émettre une valeur `value` sur un signal `s`, il suffit d'écrire `emit s = value`. On peut également tester la présence d'un signal : `?s`, ou tester sa valeur lorsqu'il est présent à l'aide de l'opérateur `present`.

L'exemple suivant illustre l'utilisation des signaux. `example x y` émet la valeur du signal `x` si `x` est présent, et la valeur du signal `y` si `x` est absent et `y` présent.

```
let node example x y = o where
  present
     x(v) -> do emit o = v done
   | y(v) -> do emit o = v done
  end
```

Les automates

Il est également possible de programmer sous forme d'automate. Un automate est un ensemble d'états et de transitions. Ces dernières peuvent être de deux types :

- Une transition est qualifiée de *faible* si, au cours d'un pas de temps, on commence par évaluer le contenu de l'état avant de tester la transition. Elle est alors introduite par le mot `until`.
- À l'inverse, une transition *forte* est testée avant l'évaluation du contenu de l'état. Si une telle transition est empruntée, c'est le contenu de l'état cible qui sera évalué au cours du pas de temps. De telles transitions sont introduites par le mot `unless`.

Les automates sont probablement la structure de contrôle la plus agréable pour l'utilisateur. On peut ainsi très facilement traduire le comportement suivant : *Attendre que la condition `cond` soit vérifiée puis émettre `value`.*

```
let node wait_emit cond = o where
  rec automaton
    | Wait ->
        do
        until cond then Exec
    | Send ->
        do o = emit value
        done
  end
```

Pour plus de précisions sur le langage, on pourra se reférer à [21].

4.2 Normalisation d'une partition *Antescofo*

Le langage d'*Antescofo* est volontairement expressif. Il permet de capturer l'architecture d'une œuvre musicale dans toute sa complexité. Cette complexité est nécessaire au compositeur lors de la phase de création. Pour autant, il n'est pas forcément utile de la conserver au moment de l'exécution.

Le comportement du système est en effet assez simple. Il s'agit d'attendre des événements instrumentaux particuliers pour ensuite envoyer une succession de messages de contrôle. Il est donc assez naturel de regrouper les actions atomiques en fonction de leur événement déclencheur. On obtient ainsi une forme normale très simple de la partition.

La normalisation s'effectue en trois étapes :
- Propagation des attributs d'erreur
- Répartition des actions
- Aplatissement

Propagation des attributs d'erreur

La forme normale ne doit contenir que des actions atomiques. Il est donc nécessaire de propager les attributs de rattrapage d'erreur des groupes aux actions atomiques. Par exemple, les actions attachées à un groupe `causal` deviennent `causal`.

On peut remarquer que cette propagation n'est pas possible dans le cas des attributs `global` et `local`. En effet, en cas d'erreur, la totalité d'un groupe caractérisé par l'un de ces attributs est affectée. Pour propager ces attributs aux actions atomiques, il faudrait donc conserver un lien entre les actions d'un même groupe. Dans le cadre de cette étude nous avons donc choisi d'écarter ces deux attributs. Les deux autres, en revanche, se propagent facilement aux actions atomiques.

Ainsi, si un événement est manqué, les actions atomiques qui lui sont associées sont rattachées à l'événement détecté suivant. Les actions qui sont normalement antérieures à cet événement sont lancées immédiatement si elles sont déclarées `causal`, simplement ignorées si elles sont déclarées `partial`. Les autres seront lancées à la date prévue par la partition.

Enfin, si un groupe g_1 est contenu dans un groupe g_2, les actions associés à g_1 prennent l'attribut qui caractérise g_2. On peut en effet remarquer que l'attribut de gestion d'erreur des groupes qui ne sont pas directement rattachés à un événement instrumental n'a pas vraiment de sens. En cas d'erreur, si le groupe g_2 est `causal`, g_1 sera exécuté normalement avec un délai nul. Si g_2 est `partial`, g_1 sera soit exécuté normalement, soit simplement ignoré selon sa position par rapport à l'événement détecté. En outre, les actions atomiques directement attachées à un événement qui étaient, par convention, `local`, sont déclarées `partial`.

Répartition des actions atomiques

La notion de groupe `tight` permet au compositeur d'associer implicitement des actions à un ensemble d'événements instrumentaux. Ainsi, au moment de l'exécution d'un groupe `tight`, on commence par calculer à quel événement devrait se rattacher chacune des actions du groupe. Cela correspond au processus de réécriture présenté dans la section 2.3. L'objectif de la normalisation est de réaliser cette analyse statiquement, avant l'exécution.

Notons que cette répartition statique interdit certains comportements. Ainsi, définir un groupe g_1 `tight` dans un groupe g_2 `loose` n'a plus vraiment de sens. En effet, dans ce cas, les événements associés aux actions de g_1 dépendent énormément des variations de tempo de l'instrumentiste qui précède le lancement du groupe. Nous considérerons donc que tous les groupes contenus dans un groupe `loose` sont, eux aussi, `loose`.

Pour chaque événement instrumental on créé deux ensembles d'actions.

- `loose_actions` correspond aux actions atomiques et aux groupes `loose` directement associés à l'événement.
- `tight_actions` regroupe le corps des groupes `tight` directement associés à l'événement. Cet ensemble peut donc contenir tout type d'action : groupes `tight`, groupes `loose` ou actions atomiques.

Les actions de `loose_actions` sont directement rattachées à leur événement déclencheur. En revanche, il convient de répartir les éléments de `tight_actions`. On procède donc par induction sur la structure des actions.

- Si l'action est atomique, on l'ajoute à l'ensemble `loose_actions` associé à l'événément qui lui est le plus proche.
- De même si l'action est un groupe `loose`, on l'ajoute à l'ensemble `loose_actions` associé à l'événément qui lui est le plus proche.
- Enfin, si l'action est un groupe `tight`, on applique cette procédure à l'ensemble des actions contenues dans le groupe.

Après itération, on a ainsi attaché à chaque événement instrumental une liste d'actions, `loose_actions`, qui ne contient que des actions atomiques et des groupes `loose`. Toutes les actions électroniques sont donc rattachées directement à leur événement déclencheur.

Aplatissement

Enfin, pour obtenir la forme normale, il suffit de transformer les listes `loose_actions` en listes d'actions atomiques. Il s'agit donc d'aplatir les groupes `loose`. On procède encore une fois par induction sur la structure des actions.

- Si l'action est atomique, on l'ajoute directement à la liste finale.
- Si l'action est un groupe `loose`, on applique cette procédure à l'ensemble des actions associées au groupe.

La figure 7 résume les différentes étapes de la normalisation sur un exemple simple.

4.3 Fonctionnement de l'interprète

Contrairement à l'interprète RML, décrit dans la section 3, nous considérons ici la partition sous sa forme normale. Il n'est donc plus ici question de processus d'exécution. On considère plutôt la partitition comme un ensemble de processus qui attendent la détection d'un événement pour lancer une séquence d'actions atomiques. Il nous faut donc traduire la mise en séquence d'une part, et l'attente en parallèle de plusieurs processus d'autre part.

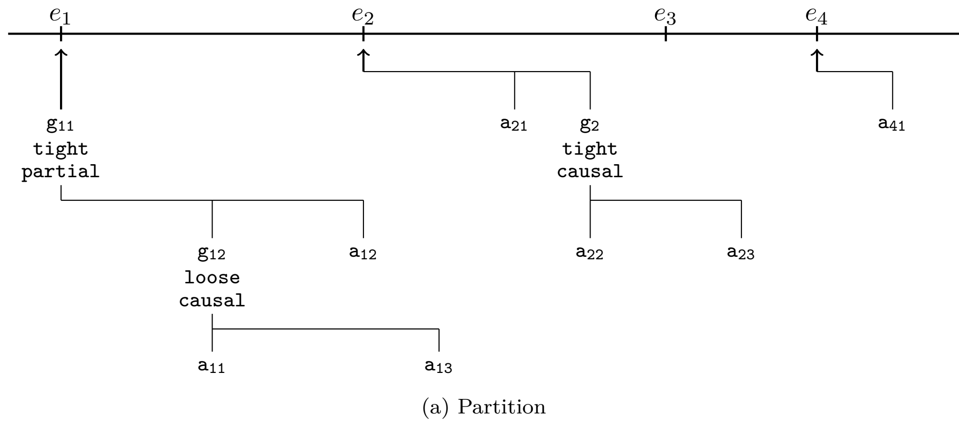
Pour exprimer cela en terme de flots de données, nous avons choisi de représenter chaque processus temporel par un flot booléen. Ce flot indique si le processus est, ou non, en cours d'exécution.

L'architecture globale de l'interprète reste cependant sensiblement la même que celle de la version RML. En particulier, nous ne reviendrons pas sur le découplage des canaux de communication qui est réalisé de la même façon dans les deux langages. Nous disposons donc en entrée de trois canaux de communications :

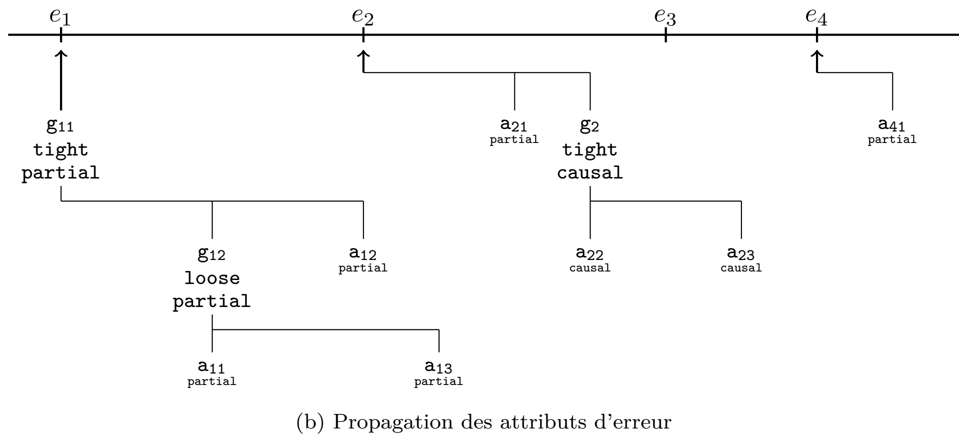
- `event` pour les événements détectés,
- `missed_event` pour les événements manqués,
- `bpm`, pour le tempo estimé.

Suivi de tempo

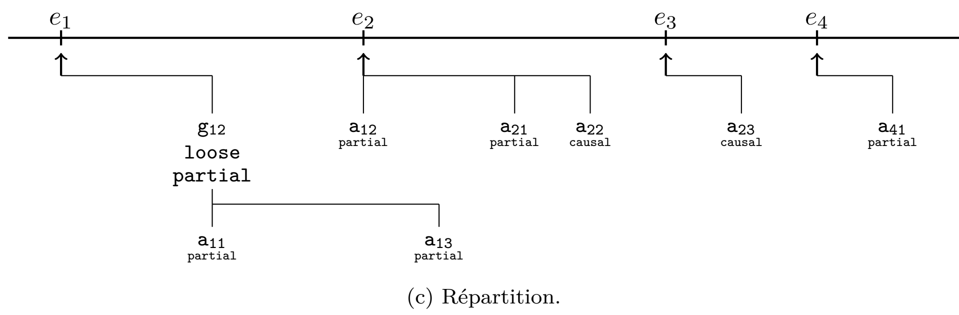
Le comportement de base d'une action *Antescofo* est le suivant : *Attendre un délai* `d` *relativement au tempo* `bpm` *puis envoyer un message* `m`. Ce travail de suivi de tempo est réalisé par le nœud `send`. Ce nœud repose sur la même idée que le processus `wait_rel` de l'interprète RML (cf. section 3.2). Si le tempo vient à changer, on recalcule le nombre de pas qu'il reste à attendre en fonction du nouveau tempo et de la proportion de la durée qu'il reste à attendre.



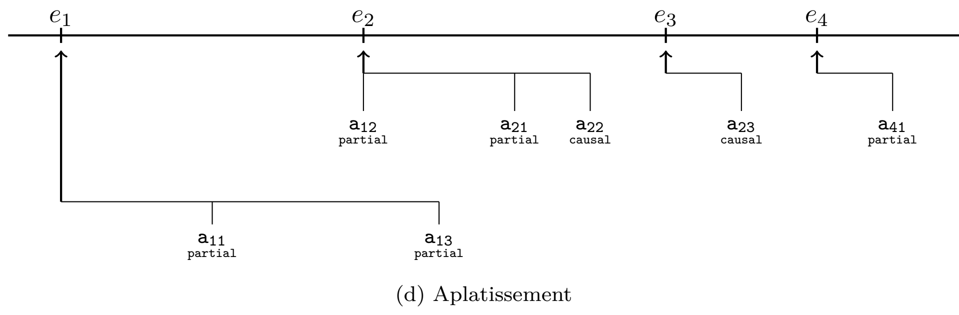
(a) Partition



(b) Propagation des attributs d'erreur



(c) Répartition.



(d) Aplatissement

FIGURE 7 – Normalisation d'une partition simple.

```
let node send (delay,message) bpm () = o where
  automaton
    Init ->
      do o = false
      unless (delay = 0.0) then Send
      unless (delay <> 0.0) then Wait(1,1)
  | Wait(p,q) ->
      let abs_dur = delay *. (60. /. bpm) in
      let nb_steps = int_of_float (abs_dur /. clock_step) in
      let n = (p * nb_steps) / q in
      let rec cpt = n -> pre cpt - 1 in
      do o = false
      until (bpm <> (bpm fby bpm)) then Wait(p * cpt, q * n)
      until (cpt = 1) then Send
  | Send ->
      do send_message message ()
      and o = true then Stop
  | Stop -> do o = true done
  end
```

- L'état initial permet d'écarter le cas où le délai à attendre est nul. Dans ce cas, on passe immédiatement dans l'état `Send`.

- L'état `Wait` est paramétré par un couple `(p,q)` qui représente la proportion de la durée qu'il reste à attendre. Tant qu'on se trouve dans cet état, l'action est en cours d'exécution, la sortie `o` prend donc la valeur *false* à chaque pas. Lorsque le tempo change, on retourne dans l'état `Wait` en actualisant le couple `(p,q)`. Dans le cas contraire, le compteur `cpt` est décrémenté à chaque pas. Lorsque celui-ci arrive à 1, la durée est écoulée, il est temps d'envoyer le message. On passe donc dans l'état `Send`.

- Lorsqu'on arrive dans l'état `Send`, l'exécution de l'action est terminée `o` prend donc la valeur *true*. L'envoi des messages est ici un effet de bord réalisé par la fonction `send_messages`. Il n'a donc aucune influence sur le flot booléen `o` qui représente l'action.

- Enfin l'utilisation d'un état `Stop` permet de ne passer qu'un seul pas de temps dans l'état `Send`. De cette manière le message n'est envoyé qu'une seule fois.

Mise en séquence

Lors de la phase de normalisation, on associe à chaque événement instrumental une liste d'actions atomiques. Pour obtenir le comportement voulu, il faut, lors de la détection d'un événement, exécuter en séquence le nœud `send` sur l'ensemble des éléments de cette liste. On utilise pour cela les trois nœuds suivants :

```
let node sequence al size = ended where
  rec i = 0 fby (if e then (i+1) else i)
  and reset e = run (al i) () every (true fby e)
  and ended = (i >= size)
```

```
let node toSeq l bpm i () =
  let n = List.length l in
  if (i < n) then send (List.nth l i) bpm ()
  else true

let node seq l bpm =
  sequence (toSeq l bpm) (List.length l)
```

- Le nœud `sequence` prend en argument une fonction qui à un entier associe un processus temporel représenté par un flot booléen. Cette fonction représente la séquence à exécuter. Le compteur `i` permet de mémoriser l'élément de la séquence qui est en cours d'exécution. Ainsi, lorsqu'un processus termine, si la valeur de `i` n'est pas égale à la taille de la séquence, on incrémente `i`, puis on lance le processus suivant. `ended` est le flot booléen qui représente la séquence. Tant que `i` est inférieur à la taille de la séquence `size`, c'est qu'il reste des processus à exécuter. `ended` prend donc la valeur *false*. Dans le cas contraire, la séquence est achevée, `ended` prend alors la valeur *true*.

- La fonction `toSeq` prend en argument une liste d'actions `l`, un flot de tempo `bpm` et un entier `i`. Cette fonction, permet de lancer le processus `send` sur la $i^{\text{ème}}$ action de la liste `l`. L'application partielle de cette fonction : `toSeq l bpm`, associe donc un processus temporel à un entier `i`. Pour exécuter une séquence d'actions représentée par une liste `l`. Il suffit donc de donner en argument au nœud `sequence` cette fonction et la taille de la liste. C'est précisément ce que fait le nœud `seq`.

- Ainsi, lorsqu'un événement déclencheur est détecté, il suffit d'appeler le nœud `seq` sur la liste qui lui est associée. À l'opposé, si un événement est manqué, on appelle le nœud `seq` sur une liste mise à jour en fonction des attributs d'erreur de chacune des actions (cf. section 4.2).

Parallèlisme et attente

La représentation des processus temporels en terme de flot booléen permet de définir très facilement un opérateur de mise en parallèle de deux processus.

```
let node (||) p1 p2 = ended where
  rec automaton
     Play ->
        do ended = false
        until (p1 & p2) then Stop
  | Stop ->
        do ended = true done
  end
```

Cet automate traduit tout simplement le fait que la mise en parallèle de deux processus temporels termine lorsque les deux processus sont terminés.

Par ailleurs, il est très facile de tester la présence de signaux. On peut donc assez aisément écrire les nœuds `await_detected_seq` et `await_missed_seq` qui attendent que l'événement `i` soit reçu sur l'un des canaux, `event` ou `missed_event` avant de lancer la séquence d'actions correspondantes.

```
let node await_detected_seq l bpm event i = o where
  rec automaton
    Await ->
    do o = false unless (detected event i) then Go
  | Go -> do o = seq l bpm done
  end

let node await_missed_seq l bpm event missed_event i = o where
  rec automaton
     Await ->
       do o = false
       unless (missed missed_event l i) then Go
  | Go ->
       let new_l = update l event in
       do o = seq new_l bpm done
  end

let node await_seq l i bpm event missed_event =
  (await_seq l bpm event i)
  ||
  (await_missed_seq l bpm missed_event i)
```

- La fonction `detected` renvoie un booléen qui indique si l'événement `i` a été reçu sur le canal `event`.
- Symétriquement, `missed` renvoie un booléen qui indique si l'événement `i` à été reçu sur le canal `missed_event`.
- `await_detected_seq` écoute le canal `event`. Si l'événement `i` est reçu, on lance `seq` sur la liste `l` associée à l'événement.
- `await_missed_seq` écoute le canal `missed_event`. Si l'événement `i` est manqué, on commence par mettre à jour la liste `l` selon l'événement détecté et les attributs d'erreur de chacune des actions. On lance ensuite `seq` sur cette liste actualisée.
- Enfin, `await_seq` permet d'écouter les deux canaux en parallèle.

Pour exécuter une partition complète, il suffit donc de lancer en parallèle un nœud `await_seq` par événement. Un partition *Antescofo* devient donc un programme *Lucid Synchrone*.

Simulation

Pour simuler le comportement de la machine d'écoute, nous avons ajouté un nœud qui permet de simuler une performance. Ce nœud a un comportement similaire au processus `run_performance` de l'interprète RML. Il prend en entrée une liste de triplets (ev, d, t) qui signifient :*Après d secondes, envoyer ev sur le canal* `ext_event` *et t sur le canal* `bpm`. Pour exécuter ce comportement, on utilise le nœud `seq` défini dans la section 4.3

```
let node send_event (ev,delay,tempo) = event,bpm where
  rec automaton
    Init ->
        do
        unless (delay = 0.0) then Send
        unless (delay <> 0.0) then Wait
  | Wait ->
        let nb_steps = int_of_float (delay /. clock_step) in
        let rec cpt = nb_steps -> pre cpt - 1 in
        do
        until (cpt = 1) then Send
  | Send ->
        do emit bpm = tempo
        and emit event = ev done
  end

let node toSeqPerf l i () =
  let n = List.length l in
  if (i < n) then send_event (List.nth l i) () else true

let node run_performance l =
  sequence (toSeqPerf l) (List.length l)
```

Les nœuds `send_event`, `toSeqPerf` et `run_performance` se comportent exactement comme les nœuds `send`, `toSeq` et `seq`, respectivement.

4.4 Discussion

Contrairement à l'implémentation en *Reactive ML*, l'ensemble du programme est cette fois statique. Il n'y a pas de création dynamique de processus. Ceci peut être vu comme une faiblesse ou comme un avantage. En effet, l'ensemble des processus doit être lancé dès le début du programme ce qui peut être très coûteux en terme de temps de calcul. Cependant, l'aspect statique permet d'apporter des garanties sur le temps nécessaire pour effectuer un pas d'exécution. Contrairement à *Reactive ML* cette durée peut être calculée avant l'exécution. On est donc assuré que le temps de calcul n'explosera jamais. Il est, de plus, bien plus facile de vérifier, voire prouver des propriétés sur un programme statique que sur un programme dynamique.

Pour ces diverses raisons, les systèmes critiques, avion, train etc. sont bien souvent codés à l'aide d'un formalisme synchrone statique [12]. À l'heure actuelle *Antescofo* ne sert que d'intermédiaire, il envoie des messages de contrôle à un logiciel, bien souvent Max MSP, qui lui, déclenche effectivement les sons électroniques. Le système ne nécessite donc pas autant de garanties temporelles qu'un pilote automatique d'avion, il peut se contenter d'être précis au centième de seconde (limite de perception de l'oreille humaine). Cependant, à terme, *Antescofo* sera peut être appelé à prendre en charge des procédures de traitement du signal qui, elles, nécessitent une grande précision. La vision statique reste donc un élément à creuser pour le futur.

Cela étant, bien que plus sûre, l'implémentation est beaucoup moins transparente qu'en RML. Il est par exemple difficile de donner une traduction précise à chacune des constructions du langage. Nous sommes donc passés par une phase de normalisation. Cependant, cette approche nous a poussé à laisser de côté une partie importante du langage original. Les comportements `global` et `local` qui correspondent pourtant à de vraies réalités musicales ont ainsi dû être écartés.

Cette implémentation ne permet donc pas de donner une sémantique aux diverses constructions du langage d'*Antescofo*. Elle permet en revanche de donner une sémantique à la partition dans sa globalité. On laisse ainsi de côté une complexité, certe nécessaire au moment de la composition, mais peut être inutile au moment de la performance.

5 Évaluation des interprètes

5.1 Partitions et performances

Pour confronter nos deux interprètes à des partitions de taille réelle, nous avons écrit un module *Parser*. Ce module prend en entrée deux fichiers : `file.asco` et `file.perf`. Le premier est un fichier de partition *Antescofo* classique (cf. section 2). On en extrait les partitions électronique et instrumentale exprimées à l'aide de notre ensemble de types (cf. section 2.5). Ces partitions peuvent alors servir d'entrée à l'un des deux interprètes.

Le second fichier représente une performance associée à la partition. On en extrait une liste de triplets qui est ensuite donnée en entrée aux fonctions de simulations de performances `run_performance`. On peut de cette manière reproduire à l'envie l'exécution d'une partition selon une performance donnée. Ce comportement peut être très appréciable dans une situation de prototypage.

5.2 Couplage avec Max/MSP

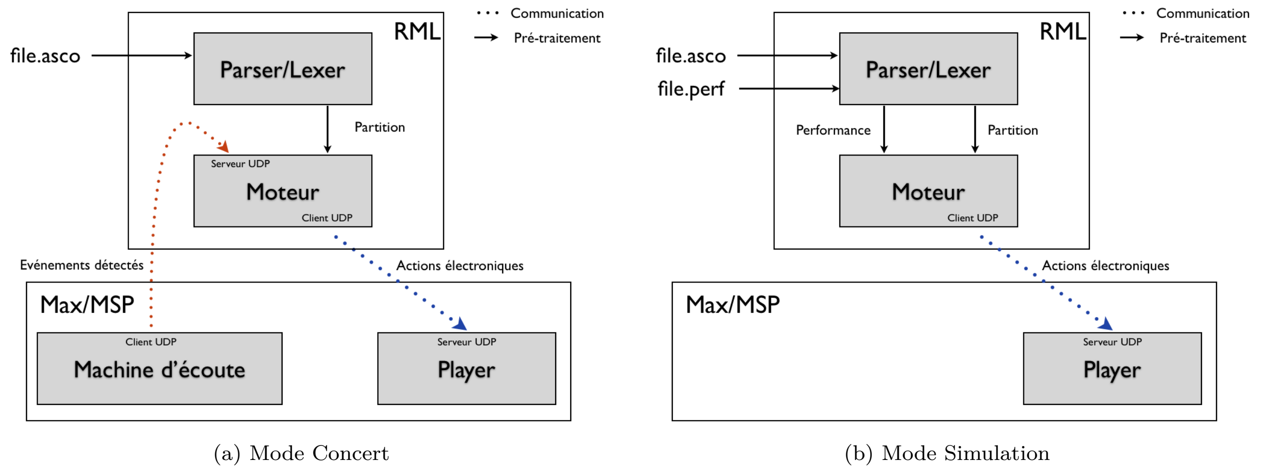
L'*Antescofo* original a été conçu pour fonctionner au sein du logiciel Max/MSP. Max est un logiciel de création d'applications multimédias interactives qui, grossièrement, fonctionne par passage de messages. Ainsi, *Antescofo* envoie des messages de contrôle qui peuvent être récupérés par une multitude d'objets au comportement divers : synthèse sonore, gestion de la lumière, accompagnement MIDI etc. La grande force d'*Antescofo* réside donc dans le fait qu'il peut facilement être intégré dans des applications musicales complexes.

Dans le but de rester le plus proche possible de ce comportement, nous avons lié nos interprètes à Max/MSP. La communication entre les deux applications se fait par le biais de sockets UDP en réseau local. Ainsi, l'interprète peut envoyer des messages de contrôle directement à Max. Il peut donc être utilisé de la même façon que la machine réactive originale.

Symétriquement, il est possible de coupler nos interprètes à la machine d'écoute d'*Antescofo*. Lorsqu'un événement est détecté, Max envoie à l'interprète l'événement et le tempo estimé. Celui-ci renvoie en contrepartie les messages de contrôle spécifiés par la partition. Notons que ce mode d'exécution correspond à une situation de concert. Il n'y a donc plus besoin de fournir un fichier de performance.

La figure 8 illustre les deux modes d'exécution possibles.

Par ailleurs, en interne, Max/MSP fonctionne un peu comme un programme synchrone. Les messages envoyés lors d'un pas de temps, sont reçus à l'instant suivant. En Max, l'épaisseur du pas de temps est un paramètre réglable par l'utilisateur. Typiquement cette épaisseur est de l'ordre de la milliseconde. C'est aussi l'épaisseur que nous avons adoptée pour nos horloges de base. D'autre part, la communication en réseau local par le biais de sockets UDP est quasiment instantanée (de l'ordre de la dizaine de nanoseconde). Il n'y a donc pas de différence de comportement audible entre nos interprètes et la machine réactive originale.



(a) Mode Concert

(b) Mode Simulation

FIGURE 8 – Illustration des deux modes d'exécution.

5.3 Évaluation

Pour évaluer l'interprète il nous fallait des partitions de taille *réelle*. Nous avons donc écrit en Java une petite application qui transforme un fichier MIDI en partition *Antescofo*. Une des voix est considérée comme la partie instrumentale, les autres sont transformées en groupe `tight`. Ceci permet de tester une des structures de contrôle, a priori la plus coûteuse, sur des partitions de taille conséquente.

Nous avons donc gagné, à peu de frais, une gigantesque base de donnée d'évaluation. Le corpus, volontairement éclectique, que nous avons retenu pour notre évaluation est présenté dans le tableau 1.

Nom	Nombre d'événements	Nombre d'actions
Au clair de la lune, Trad. Français	22	47
Greensleeves, Trad. Anglais	869	920
Hymne russe, Trad. Russe	196	1041
Yesterday, The Beatles	174	1289
Concerto pour deux violons, 3ème mvt, Bach	1317	3591
Concerto pour violon, 2ème et 3ème mvt, Tchaikovsky	3705	11062

TABLE 1 – Corpus d'évaluation

Dans cette première approche, la procédure d'évaluation est assez simpliste. Il s'agit de mettre en concurence nos interprètes avec la machine réactive d'*Antescofo*. Nous avons ainsi exécuté chacune des pièces du corpus en utilisant tour à tour chacune des machines réactives, et ce pour des performances diverses. Aucune différence n'était perceptible à l'oreille. On peut donc considérer que les performances de nos interprètes sont comparables, au moins en première approche, aux performances de la machine réactive d'*Antescofo*.

Bien entendu, cette procédure est loin d'être suffisante. Elle donne néanmoins une première validation de notre travail.

Il faudrait dans le futur, poursuivre cette évaluation de manière plus poussée et rigoureuse. On pourrait par exemple comparer des fichiers contenant la trace de l'exécution d'une performance par chacun des systèmes. Il faudrait alors se pencher sur les différentes métriques qui permettraient de comparer proprement ces deux exécutions. On pourrait par exemple s'inspirer des métriques utilisées dans le cadre des évaluations de MIREX [22]. Il serait, de plus, nécessaire d'étendre l'évaluation à l'ensemble des structures du langage d'*Antescofo*.

6 Conclusion

Le langage d'*Antescofo* offre la possibilité de composer une œuvre mixte en traitant les parties électroniques de la même manière que les parties instrumentales. Dans le cadre de cette étude, nous avons étudié ce langage d'*Antescofo* sous de mutliples aspects.

Synchronisation et gestion des erreurs

Nous avons commencé par une phase d'analyse des structures du langage. Celle-ci nous à amené à redéfinir les notions de synchronisation et de gestion des erreurs de manière complètement indépendante. En conséquence, nous avons enrichi le langage de deux nouveaux attributs de gestion des erreurs `causal` et `partial`, qui correspondent à de véritables réalités musicales.

Le langage est ainsi devenu plus modulable. La notion de point de contrôle, en particulier, permet d'imaginer toutes sortes de stratégies de synchronisation. On pourrait par exemple développer un véritable langage de contraintes qui permettrait au compositeur de définir des stratégies de synchronisation en fonction de ses besoins. Le langage deviendrait ainsi beaucoup plus souple.

Pour ce faire on pourrait, par exemple, imaginer un module indépendant qui calculerait, au moment du lancement d'un groupe, les points de contrôle qui lui sont associés. Il n'y aurait ainsi plus besoin de modifier le cœur de la machine réactive pour ajouter une nouvelle stratégie.

Sémantique

L'un des enjeux majeurs, lorsqu'on développe un nouveau langage, est de définir clairement la sémantique des différentes structures de contrôle. Le langage d'*Antescofo* offre des structures de données particulièrement complexes, il est donc assez difficile d'en préciser la sémantique. Les travaux précédents font, par exemple, appel à des notions telles que les automates temporisés [15].

Reactive ML permet des constructions très dynamiques. Il est donc bien adapté à une situation qui nécessite beaucoup de contrôle. En conséquence, il était relativement naturel de traduire les diverses structures du langage d'*Antescofo* en terme de processus d'exécution. *Reactive ML* a, de plus, une sémantique claire est bien définie [13] donc, par traduction, nous avons pourvu ces structures d'une sémantique relativement simple. Leurs spécifications s'expriment, en effet, de manière quasiment transparentes en RML. Ce paradigme était donc particulièrement bien adapté à notre problème, la concision de l'implémention de l'interprète RML (moins de 400 lignes de code comparé aux milliers de lignes du séquenceur écrit en *C++*) en est la preuve.

Lucid Synchrone, à l'inverse ne permet pas de gérer facilement la création dynamique de processus. Il n'était donc pas question de traduire, une par une, les fonctions d'exécution de chacune des constructions du langage d'*Antescofo*. Nous avons donc défini une forme normale pour les partitions. La partition est cette fois considérée dans sa globalité. On peut, de cette manière, donner un sens à l'interprétation d'une partition en termes de flots de données. Cette interprétation, plus statique, offre plus de garanties notamment en terme de temps de calcul nécessaire.

On pourrait pour compléter cette étude, s'intéresser à d'autres paradigmes de programmation synchrone. Le langage Ptolemy [23], par exemple, repose sur une conception du temps légèrement différente [24]. Il peut donc être intéressant d'essayer de traduire le langage d'*Antescofo* dans ce formalisme.

Évaluation

Nous avons donc développé deux machines réactives permettant d'interpréter des partitions *Antescofo*. Ces deux systèmes ont, à l'instar de l'*Antescofo* original, la possiblité de communiquer avec Max/MSP. Par ailleurs, les deux interprètes disposent d'une fonction de simulation qui permet de rejouer à l'identique une performance donnée.

Tout est donc en place pour confronter ces deux interprètes avec la machine réactive originale. Malheureusement, nous n'avons pas eu le temps de développer une véritable procédure d'évaluation. On pourrait par exemple faire jouer les différents systèmes de manière concurrente sur une même exécution et comparer des fichiers de trace.

Prototypage

L'interpréte RML est très modulable. Il est, en effet, très facile d'ajouter de nouvelles constructions. En outre, il permet de rejouer une performance à l'identique. C'est donc un outil idéal de prototypage. Nous l'avons ainsi utilisé avec succès pour prototyper les attributs de gestion d'erreur `causal` et `partial`. Cet interprète pourrait donc être utilisé, dans le futur, comme oracle pour tester de nouvelles constructions en situation réelle.

Perspectives

Dans le cadre de cette étude, nous nous sommes cantonnés à l'étude de la partie haute d'*Antescofo* : le séquenceur. Cela étant, la machine d'écoute est, elle aussi, un système réactif. Celui-ci est cependant bien plus complexe car il fait intervenir des notions pointues de traitement du signal et d'apprentissage automatique [25]. Il pourrait donc être intéressant d'analyser cette machine d'écoute à la lumière des paradigmes de programmation synchrones. Cela pose en particulier des problèmes d'efficacité redoutables.

À l'heure actuelle, *Antescofo* ne peut qu'envoyer des messages de contrôle qui sont ensuite traités par Max/MSP. Pour en faire un instrumentiste à part entière, il faudrait que le système soit capable de synthétiser seul des signaux sonores. *Faust* [26], par exemple est un langage synchrone flot de donnée dédié à la synthèse de signaux musicaux. Pour l'intégrer au sein d'*Antescofo*, il faut être capable de manipuler deux échelles de temps [27]. L'une très rapide pour permettre la synthèse, et l'autre bien plus lente qui pour gérer les messages de la machine d'écoute. On retrouve une architecture semblable au sein de Max/MSP. En effet, les messages de contrôle de Max sont cadencés sur une horloge assez lente. La synthèse est prise en charge par la partie MSP qui est, elle, bien plus rapide. L'utilisation des langages synchrones permettrait de reproduire cette architecture en apportant plus de garanties sur le comportement du système.

Enfin, certains langages synchrones, dits hybrides [28, 29], permettent de gérer du temps continu. Cette notion pourrait être utilisée pour modéliser le temps musical de manière bien plus fine. En effet, dans l'approche que nous avons adoptée, nous considérons toujours des pas de temps d'épaisseur fixe. On pourrait ainsi se rapprocher d'une conception du temps plus proche du *temps lisse* de Boulez [30] qui considère un écoulement continu dans le temps d'une masse sonore en évolution.

Toutes ces problèmatiques sont liées à des projets de recherche très actifs sur les langages synchrones. Il serait donc intéressant de voir ce que peut apporter une approche atypique car motivée par des problèmes musicaux.

Références

[1] V. Tiffon : *Recherches sur les musiques mixtes*. Thèse de doctorat, université de Provence, 1994.

[2] A. Cont : Synchronisme musical et musiques mixtes : du temps écrit au temps produit. *Circuit : musiques contemporaines*, 22(1), 2012.

[3] A. Cont et M.R. Team : On the creative use of score following and its impact on research. *Sound and Music Computing, Padova, Italy*, 2011.

[4] A. Cont : Antescofo : Anticipatory synchronization and control of interactive parameters in computer music. *In Proceedings of the International Computer Music Conference*, 2008.

[5] E.W. Large et M.R. Jones : The dynamics of attending : How people track time-varying events. *Psychological review*, 106(1):119, 1999.

[6] C. André : Comparaison des styles de programmation de langages synchrones. *Projet AOSTE-Juin*, 5, 2005.

[7] J. Echeveste, A. Cont, J.L Giavitto et F. Jacquemard : Formalisation des relations temporelles dans un contexte d'accompagnement musical automatique. *In 8e Colloque sur la Modélisation des Systèmes Réactifs (MSR'11)*, volume 45, pages 109–124, 2011.

[8] N. Halbwachs : *Synchronous programming of reactive systems*. Springer, 1993.

[9] G. Berry et G. Gonthier : The esterel synchronous programming language : design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[10] F. Boussinot et R. De Simone : The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[11] P. LeGuernic, T. Gautier, M. Le Borgne et C. Le Maire : Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[12] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic et R. De Simone : The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[13] L. Mandel et M. Pouzet : Reactive ml : a reactive extension to ml. *In Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[14] P. Caspi, G. Hamon et M. Pouzet : Synchronous functional programming : The lucid synchrone experiment, 2008.

[15] J. Echeveste, A. Cont, J.L Giavitto et F. Jacquemard : Antescofo : a domain specific language for real time musician-computer interaction, 2012. Article soumis.

[16] L. Mandel et F. Plateau : Interactive programming of reactive systems. *Electronic Notes in Theoretical Computer Science*, 238(1):21–36, 2009.

[17] L. Mandel et F. Benbadis : Simulation of mobile ad hoc network protocols in ReactiveML. *In Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, avril 2005.

[18] L. Mandel : *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. Thèse de doctorat, Université Paris 6, 2006.

[19] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud : The synchronous data flow programming language lustre. *In Proceedings of the IEEE*, pages 1305–1320. IEEE, 1991.

[20] G. Kahn : The semantics of a simple language for parallel programming. *In Proceedings of IFIP Congress74*, pages 471–475, 1974.

[21] M. Pouzet : *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006.

[22] A. Cont, D. Schwarz, N.Schnell et C. Raphael : Evaluation of real-time audio-to-score alignment. *In MIREX*. ISMIR International Symposium on Music Information Retrieval, 2007.

[23] E.A. Lee et H. Zheng : Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. *In Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123. ACM, 2007.

[24] E.A. Lee : Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[25] A. Cont : A coupled duration-focused architecture for real-time music-to-score alignment. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(6):974–987, 2010.

[26] Y. Orlarey, D. Fober et S. Letz : Syntactical and semantical aspects of faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):623–632, 2004.

[27] C Pasteur : Clock domains in a reactive functional language. SYNCHRON, 2011.

[28] A. Benveniste, B. Caillaud et M. Pouzet : The fundamentals of hybrid systems modelers. *In Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 4180–4185. IEEE, 2010.

[29] A. Benveniste, T. Bourke, B. Caillaud et M. Pouzet : Divide and recycle : types and compilation for a hybrid synchronous language. *In Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 61–70. ACM, 2011.

[30] P. Boulez : *Penser la musique aujourd'hui*, volume 13. Denoël, 1963.