# Articulation dynamique de structures temporelles pour l'informatique musicale

Dimitri Bouche et Jean Bresson

UMR 9912 STMS : IRCAM - CNRS - UPMC
Paris, France
{bouche,bresson}@ircam.fr

**Résumé**

Les systèmes informatiques musicaux se divisent principalement en deux catégories : les systèmes de composition en « temps différé » et les systèmes de performance en « temps réel ». De récents travaux et applications ont introduit des systèmes se situant à la frontière de ces deux paradigmes, permettant à des processus de calcul statiques de s'intégrer au sein de contextes dynamiques, et inversement. Cela nécessite l'élaboration de techniques mettant en oeuvre des modèles réactifs intégrant des structures temporelles se déroulant sur le long terme, ne permettant pas l'utilisation de stratégies classiques de vision à court terme. Nous introduisons un système établissant une représentation des données musicales de haut niveau (orientée objet), accompagnée d'un moteur d'exécution dynamique bas niveau. Les stratégies de planification purement réactives n'étant pas adaptées aux contraintes musicales de ce système, nous dressons une étude comparative des techniques existantes et proposons un modèle mélangeant mécanismes statiques et dynamiques.

## Table des matières

**1 Introduction** **2**

**2 Structures et exécution en informatique musicale** **3**

2.1 Représentation . . . 3

2.2 Planification . . . 3

2.3 Stratégies classiques d'ordonnancement . . . 3

**3 Vers un modèle réactif pour la composition** **6**

3.1 Structure de donnée . . . 6

3.2 Moteur d'exécution . . . 7

3.3 Révision automatique des plans . . . 10

**4 Conclusions et Perspectives** **12**

**5 Remerciements** **13**

# 1 Introduction

Les processus de création musicale peuvent se décomposer en deux étapes qui sont l'écriture (la composition) et la performance (le concert). La composition est effectuée en « *temps différé* » : elle peut de dérouler sur une durée indéterminée, sans aucune contrainte de temps, et son objet est d'élaborer un scénario temporel. Le concert, apparenté au « *temps réel* », possède une durée finie ou régie par le scénario déterminé lors de l'écriture, et son exécution doit donc répondre à des contraintes temporelles plus ou moins dures. Les logiciels d'informatique musicale se divisent ainsi en deux principales catégories, correspondant à ces deux phases dans lesquelles le temps est géré de manière distincte, voire opposée.

La première phase, nommée « composition assistée par ordinateur » (CAO), vise à fournir des outils permettant la création et la manipulation de structures musicales. Ces structures peuvent ensuite être lues comme des partitions et exécutées par des musiciens ou par des lecteurs et synthétiseurs audio. Dans ce cas de figure, les calculs permettant la génération des structures musicales sont intégralement décorrélés de leur exécution : elles peuvent être construites sans nécessiter d'optimisation particulière, et considérées comme statiques et exécutables a posteriori par un moteur dédié.

La seconde phase fait intervenir des environnements interactifs dits orientés *performance*, qui se concentrent sur l'exécution de processus en temps réels, permettant une grande intéractivité entre le résultat en cours d'exécution et les actions d'un utilisateur ou d'un musicien sur scène. Dans ces systèmes, le rendu sonore est le résultat de calculs périodiques déclenchés par des interruptions provenant du moteur audio ou de l'environnement extérieur, et ces calculs se doivent d'être exécutées en temps borné. On parle alors d'exécution entremêlée à la phase de calcul.

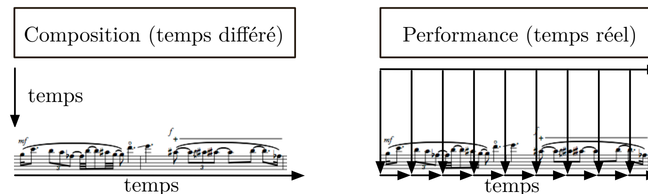


Figure 1 – Gestion du temps en composition et performance

Des travaux récents s'orientent vers l'établissement d'environnements à la frontière de ces deux paradigmes, permettant à des processus de composition de faire partie de la performance [1], et vice-versa [7]. Dans le cadre de cet article nous considérons un environnement de composition auquel nous intégrons des capacités réactives. Cela permet de répondre aux problématiques posées par les besoins actuels des compositeurs, notamment la possibilité d'intégrer des processus de composition interagissant avec l'environnement extérieur et générant des données musicales au sein d'un scénario temporel [10]. Pour cela, nous abordons des techniques d'ordonnancement incrémental et dynamique que nous intégrons dans une structure réactive. Notre système est implanté et testé au sein du logiciel de CAO OpenMusic [6].

La section 2 présente une étude des mécanismes d'écriture et d'exécution relatifs à l'informatique musicale en les situant par rapport aux travaux similaires dans d'autres domaines. La section 3 définit un modèle réactif satisfaisant les contraintes posées par un modèle hybride mêlant composition et performance, et détaille le fonctionnement du moteur de planification et d'ordonnancement de celui-ci. Finalement, nous présentons les applications possibles et en cours d'étude puis discutons ces travaux et les perspectives ouvertes.

# 2 Structures et exécution en informatique musicale

## 2.1 Représentation

Une note, pouvant être considérée comme l'objet minimal d'une partition, nécessite au moins deux actions à exécuter par un synthétiseur : une action *note-on*, et une action une *note-off*. La *note-on* doit être déclenchée au temps de la note, et la *note-off* au temps + durée de la note. Entre ces deux actions, des contrôles continus peuvent également être transmis pour faire varier des paramètres tels que le volume, la fréquence, ou d'autres effets offerts par le synthétiseur (vibrato, filtrage, réverbération etc.). Un objet musical possède donc une extension temporelle et correspond a un ensemble de processus qui se déroulent dans le temps.

Toutefois, ces processus doivent être organisés de manière à faciliter la manipulation musicale : le déplacement temporel d'une note dans une partition doit engendrer un déplacement de son ensemble d'actions. Suivant ce principe, des objets musicaux de plus haut niveau contiennent d'autres objets : un accord contient plusieurs notes simultanées, une suite d'accords contient plusieurs notes ou accords sous un référentiel de temps commun etc. Il est donc logique en informatique musicale de représenter les données sous forme hiérarchique [3], où la partition est considérée comme l'objet parent de plus haut niveau.

## 2.2 Planification

Contrairement à des systèmes d'intelligence artificielle ayant des objectifs à réaliser en un temps imparti ou avec un ensemble de contraintes à respecter, un environnement de composition ou de performance musicale n'a pas ou peu de degré de liberté quant à la planification des tâches. Cette phase de planification est régie par une organisation décrite au préalable par le compositeur, se retrouvant dans la représentation haut niveau des données, et gérée par un « planificateur » ou *planner*. Par exemple, les séquenceurs sont utilisés à cet effet par les compositeurs, leur permettant de disposer les structures musicales sur une *timeline* [5] à la manière de notes sur une partition, établissant ainsi un ordre de succession qui devra être respecté lors du rendu de ces même structures. Le planner traduit ensuite ces informations en un ou plusieurs *plans* à exécuter, contenant des actions datées et triées. Pour ce qui est des objets élémentaires, les actions sont connues à l'avance. Par exemple, une note se décompose en deux appels aux fonctions *note-on* et *note-off* (voir section 2.1), et une courbe de contrôle OSC [23] d'une durée $t$ et échantillonnée à une fréquence $f$ se décompose en une suite de $f * t$ appels à la fonction *osc-send*. Dans le cas de notations ou d'objets définis par un utilisateur, le planner doit être informé de la nature des actions correspondantes afin de les inclure dans le plan.

## 2.3 Stratégies classiques d'ordonnancement

L'exécution du ou des plans produits par le planner est gérée par un « ordonnanceur » ou *scheduler*. Celui ci déclenche les actions des plans en temps voulu [1] et est aussi en charge de réaliser les tâches de plus bas niveau non écrites par le compositeur et n'ayant en général pas d'impact direct sur le rendu attendu (calculs, mises à jour graphique etc.). Nous dressons ici un aperçu des stratégies de planification et d'exécution possibles en informatique musicale, couvrant les principaux cas d'utilisation.

1. Nous considérons dans cet article que l'exécution d'un plan et effectuée grâce à une « *boucle d'exécution* » (apparaissant dans la figure 8) comparant périodiquement la date de la prochaine action du plan et la date actuelle du scheduler.

### 2.3.1 Statique

Cette stratégie est celle adoptée dans les systèmes dédiés à la composition comme OpenMusic. La planification des actions est totalement réalisée au préalable de leur exécution, et ne peut être sujette à modifications. La partition, ou son équivalent informatique qu'est le séquenceur, est traduite et aplatie en un plan statique (la structure hiérarchique n'est pas propagée au niveau du système d'ordonnancement). La figure 2 schématise ce processus de planification pour une simple partition musicale.

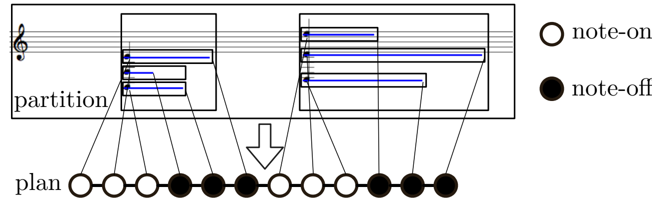


FIGURE 2 – Processus de planification statique d'une partition contenant deux accords

Exécuter un plan revient à traiter une file d'exécution à priorités définies par les temps [15] auxquels les actions sont disposées sur la partition. L'édition du plan produit par le planner n'étant pas autorisée durant l'exécution, tous les calculs ont été effectués à l'avance et l'unique rôle du scheduler est d'exécuter les actions en temps voulu. La planification est donc entièrement décorrelée de l'ordonnancement. Par conséquent, cette stratégie ne permet pas de prendre en compte des interactions avec la partition durant son exécution.

### 2.3.2 Réactive

Nous présentons ici une stratégie purement réactive, utilisée par certains logiciels dédiés à la performance musicale. A l'opposé de la précédente, celle-ci permet un maximum d'interactions pendant l'exécution d'une pièce. La planification réactive, largement étudiée et utilisée en robotique [13], peut se résumer à ne planifier qu'une action à la fois. Une fois cette action exécutée par le scheduler, celui-ci sollicite le planner pour connaître la prochaine. De cette manière, pendant le rendu d'une partition, n'importe quelle modification peut être opérée sur celle-ci sans impacter les performances (ou engendrer au maximum une requête de planification si la modification concerne une zone temporelle comprise entre le temps actuel du scheduler et le temps de la prochaine action prévue).

La hiérarchie des structures musicales est ici entièrement préservée : l'exécution d'un objet peut déclencher celle d'un objet « enfant », qui sera lui aussi sollicité par le planner, et ainsi de suite. L'exemple d'un tel processus de planification controlé par l'exécution, et donc par le scheduler, est schématisé dans la figure 3.

Dans ce cas de figure, contrairement à la stratégie statique présentée en 2.3.1, la planification est entremêlée avec l'ordonnancement. Le scheduler contrôle les appels successifs au planner en y intégrant à la volée ses résultats, qui sont des plans *singleton*. Ce type de stratégie favorise les interactions du fait du coût négligeable de l'édition de partition. En revanche, aucun avantage n'est tiré de la connaissance préalable et sur le long terme que constitue une partition. En particulier, il n'est pas possible de regrouper les calculs du planner à un instant donné pour optimiser l'exécution en temps réduit de nombreuses actions. Par exemple, considérant un accord contenant $n$ notes simultanées, une telle stratégie engendrera $n-1$ calculs de planification

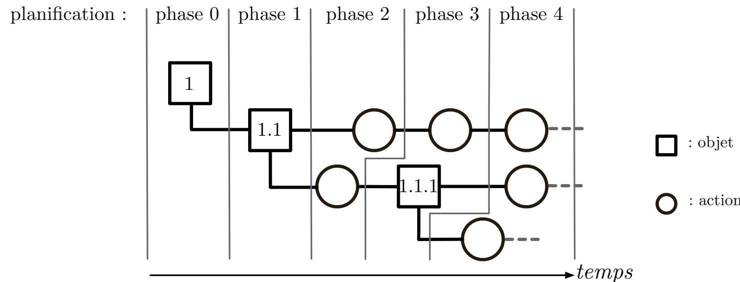


Figure 3 – Processus de planification réactif d'une partition sous forme hiérarchique

entre les $n$ appels à la fonction *note-on*, pouvant entrainer des retards perceptibles faisant entendre une suite de notes plutôt qu'un accord.

### 2.3.3 Stratégies intermédiaires

Notre objectif étant de proposer un environnement de composition réactif, nous nous intéressons également à des stratégies intermédiaires, permettant d'adapter une stratégie statique à des besoins dynamiques. Ces besoins sont catégorisés comme des stratégies de planification continues, ou *continual planning* [9].

La première approche est d'autoriser les modifications sur le plan d'exécution dans une stratégie statique [4]. Si l'on considère que la partition (donc le plan) peut être modifiée pendant son exécution, on autorise planner et scheduler a opérer de manière concurrente, et non plus uniquement séquentielle [22]. Le planner fournit un plan complet au préalable mais peut être appelé à tout moment par le scheduler pour effectuer une modification sur le plan [2]. Les opérations élémentaires autorisées sur le plan en cours d'exécution sont schématisées en figure 4.

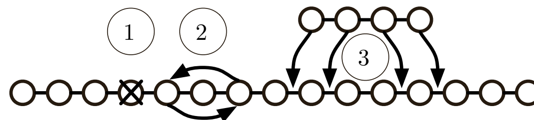


Figure 4 – Plan statique éditable : 1) Dé-planification 2) Re-planification 3) Planification

Cette stratégie, bien que fonctionnelle, s'avère inefficace dans le cas de partitions mobilisant dynamiquement de trop nombreux objets. En effet, l'insertion ou suppression d'un objet équivalent à $N$ actions sont réalisées en $\Theta(N)$ et le déplacement en $\mathcal{O}(2N)$. De fait, l'exécution du plan peut se trouver mise en attente pendant une durée non négligeable, et impacter fortement la cohérence temporelle en sortie.

Considérant la structure hiérarchique des partitions (voir section 2.1), une seconde approche est de conserver un ou plusieurs niveaux de hiérarchie dans la planification des actions à exécuter [3]. La figure 5 illustre une planification à un niveau de hiérarchie conservé. Le planner

2. A plus bas niveau, l'accès concurrent de ces deux entités sur le plan est sécurisé par un système de verrou.

3. En musique, ne conserver qu'un faible niveau de hiérarchie suffit à un gain important de performances étant donné la profondeur en générale limitée des objets inclus dans la partition.

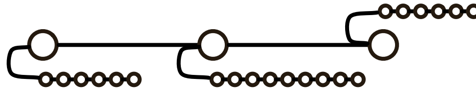

Figure 5 – Plan statique éditable à deux niveaux de hiérarchie

traduit alors la partition en un « plan d'objets », et chaque objet réfère à un plan qui lui est propre [4]. De cette manière, l'insertion ou édition d'un objet directement dans la partition peut être réalisée en $\mathcal{O}(M)$ où $M$ est le nombre d'objets dans la partition [5]. En revanche, le problème de complexité de calcul posé par la stratégie précédente est conservé à un niveau plus bas dans la hiérarchie, lors de l'édition du contenu des objets de la partition.

# 3 Vers un modèle réactif pour la composition

Nous définissons ici un modèle réactif adapté la composition (dans la gestion de processus considérés « temps différés »), composé d'une structure de donnée adéquate et d'une stratégie réactive de planification et d'ordonnancement.

## 3.1 Structure de donnée

Nous définissons ici, comme introduit en section 2.1, une représentation hiérarchique des données. Soit un objet $O : <t_O, id_O, C_O>$ où :

— $t_O$ est une date relative à l'objet parent de $O$,
— $id_O$ est un identifiant hiérarchique unique,
— $C_O$ est le contenu de $O$.

Les identifiants hiérarchiques sont construits en ajoutant un identifiant local unique $i$ à l'identifiant de l'objet parent $id$ :

$$O_X \in C_{O_Y} \rightarrow id_{O_X} = id_{O_Y}.i$$

Cette représentation présentée en figure 6 facilite la manipulation des données à haut niveau, et permet le développement de stratégies d'ordonnancement hiérarchiques et dynamiques [14].

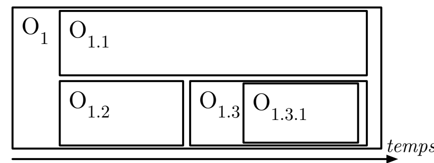


Figure 6 – Structure hiérarchique des données ($O_1$ représente la partition)

La planification d'exécution d'un objet produit un plan composé d'actions, que nous définissons comme des structures de la forme $a : <t^a, id^a, f^a>$ où :

— $t^a$ est une date,
— $id^a$ est un identifiant unique,

4. Exécuter ce plan nécessite donc une double boucle parcourant le « plan d'objets » et les plans des objets.
5. Dans un cas d'utilisation musicale réaliste, ce coût se révèle faible en général.

— $f^a$ est une fonction.

Le planner produit le plan $P_O$ correspondant à un objet $O$ en accédant à $C_O$. Dans ce processus, pour chaque $a_j = <t^{a_j}, id^{a_j}, f^{a_j}> \ \in P_{O_i}$ :

— Les **fonctions** $f^{a_j}$ dépendent du type du contenu de $C_{O_i}$ et sont indépendantes du modèle de planification et d'ordonnancement. Leur définition est principalement la tâche de l'utilisateur ou du dévelopeur.
— Les **dates** $t^{a_j}$ des actions doivent être exprimées comme des temps absolus (et non relatives à l'objet parent) dans $P_{O_i}$.
— Les **identifiants d'action** $id^{a_j}$ sont automatiquement déduits de $id_{O_i}$. Par exemple, à deux actions provenant d'un objet ayant pour identifiant $id_{O_i} = a.b$ seront assignés les identifiants $id^{a_1} = a.b.1$ et $id^{a_2} = a.b.2$. Cette correspondance permet de conserver un lien entre une action et son objet d'origine lors de son exécution.

## 3.2 Moteur d'exécution

L'objectif principal du moteur d'exécution d'un objet de structure décrite en section 3.1 est de proposer une stratégie de planification et d'ordonnancement permettant l'édition dynamique des objets à coût faible, tirer avantage de la connaissance sur le long terme qu'un environnement de composition apporte et assurer une interactivité avec l'environnement extérieur. Pour les besoins de ce moteur d'exécution, nous étendons la définition de la structure $O$ définie en section 3.1 en lui ajoutant les champs suivants :

— $state_O$ est l'état courant de l'objet, *play*, *pause* ou *stop* (voir section 3.2.1),
— $plan_O$ est le plan courant relatif à l'objet $O$ (voir section 3.2.2),
— $interval_O$ est la fenêtre temporelle du planner pour l'objet $O$ (voir section 3.2.2),
— $localtime_O$ est le temps local de l'objet (voir section 3.2.3),
— $count_O$ est le nombre de requêtes de planification consécutives effectuées pour l'objet (voir section 3.2.3).

### 3.2.1 Interface de contrôle

Pour qu'un objet soit lu, il doit être « inscrit » dans le registre du scheduler ($register_S$ pour un scheduler $S$). Une fois inscrit, le scheduler considère qu'il doit effectuer le rendu de cet objet si son état ($state_O$) est égal à *play*. Nous proposons une interface de contrôle composée de fonctions élémentaires permettant de prendre en compte la hiérarchie des objets dans le contrôle de leur rendu (voir procédures 1 à 3).

**Procedure 1:** $play_object_S(O)$

**if** $O \in register_S$ **then** $state_O \leftarrow play$;
**else** ajouter $O$ dans $register_S$;

**Procedure 2:** $pause_object_S(O)$

**for** $O_i$ ***in*** $register_S$ **do**
  **if** $id_O$ *est un préfixe de* $id_{O_i}$ **then** $O_i.state \leftarrow pause$;
**end**

**Procedure 3:** $stop_object_S(O)$

**for** $O_i$ ***in*** $register_S$ **do**
  **if** $id_O$ *est un préfixe de* $id_{O_i}$ **then** retirer $O_i$ de $register_S$;
**end**

La hiérarchie, repartition dans les identifiants des objets, permet de localiser les objets en cours de lecture issus d'un objet parent au sein du registre (voir section 3.2.2). Par exemple, si un objet $O$ doit être arrêté, il convient d'arrêter tous les objets dont l'identifiant a pour préfixe $id_O$.

Le scheduler est ensuite en charge de solliciter le planner pour associer un plan à tout objet de son registre, et les exécuter [6] en suivant les stratégies de planification et d'ordonnancement décrites en section 3.2.2 et 3.2.3.

### 3.2.2 Planification par intervalle

Le planification par intervalle [17] produit des plans de forme classique, à savoir des listes d'actions triées et datées. La hiérarchie est ici préservée : le plan d'un objet $O$ contient, pour les objets de structure $O$ dans $C_O$, des appels à la fonction $play_object_S$. La production du plan d'un objet ($plan_O$) est cette fois limitée à une fenêtre temporelle ($interval_O$). Ce fenêtrage temporel de la planification se situe à l'intermédiaire d'une planification statique (cf. section 2.3.1) et réactive (cf. section 2.3.2), et produit des plans « partiels » [7]. En comparaison à une planification purement réactive, cela permet d'éviter une surcharge inutile de calculs lorsque de nombreuses actions doivent être exécutées dans un intervalle de temps restreint [8], auquel cas les requêtes de planification successives entraineraient des retards potentiels quant aux exécutions désirées. Le nombre de requêtes de planification dépend ici du temps et non pas du nombre d'actions à exécuter par un objet.

Les opérations d'édition d'objets peuvent être effectuées à faible coût, bien que dans une moindre mesure que dans le cas de la planification réactive. En effet, comme les plans ne consistent pas en des *singleton* d'actions mais en des listes d'actions valables dans un certain intervalle de temps, la probabilité qu'ils soient affectés par une modification des données est augmentée. Cependant, la flexibilité du fenêtrage permet aussi bien de tendre vers une planification réactive que statique, et offre donc un contrôle sur l'optimisation des modifications et interactions (voir section 3.3).

Les intervalles soumis au planner sont dépendants des objets et peuvent être modifiés. Cela permet de prévoir des plans entiers d'objets, tout comme des plans à court-terme, et donc de planifier de manière optimale un objet considéré comme statique ou de limiter les calculs préalables pour les objets considérés comme dynamiques. Par conséquent, les objets statiques d'une partition se verront assigner un intervalle infini, alors que les objets pouvant être édités pendant leur exécution possèderont un intervalle court [9].

A chaque objet $O$ est assigné un « temps local » ($localtime_O$, géré par le scheduler, voir section 3.2.3) à partir duquel une phase de planification démarre. Si un objet $O$ a un temps local $t1$, une requête au planner produira un plan valable dans l'intervalle $[t1; t1 + interval_O[$.

6. Lors de l'exécution de son registre, la *boucle d'exécution* parcourt donc une liste d'objets (verticalement) et leurs plans (horizontalement).

7. Seule une partie de l'objet est planifiée.

8. Une planification préalable d'une liste d'actions aux dates proches ou égales est optimale car élimine toute phase de calcul entre les différentes exécutions.

9. Cette étape de décision, non détaillée dans cet article, est effectuée par une analyse préalable de la partition.

La fonction ***make_ plan*** produisant un tel plan est définie par l'équation suivante :

$\boldsymbol{make_plan}(id_O) = \{(a_1, ..., a_n) \mid [t^{a_1}; t^{a_n}] \subset [t1; t1 + interval_O[\}$

La figure 7 illustre cette méthode de planification pour les objets statiques et dynamiques.

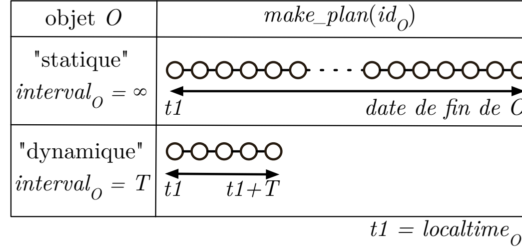


FIGURE 7 – Illustration de la planification par intervalle

Cette stratégie nécessite une communication active entre scheduler et planner : les plans à exécuter étant potentiellement de durées non homogènes, les requêtes de panification doivent être gérées pour chaque objet et peuvent être déclenchées par le temps (*time-triggered*, fin du plan en cours) ou par réaction à un événement (*event-triggered*, modification d'un objet au temps de l'exécution).

### 3.2.3 Ordonnancement incrémental de plans partiels

Une fois que le plan « partiel » d'un objet a été entièrement exécuté, le scheduler sollicite le planner pour le prochain plan. Si aucun résultat n'est produit pour l'intervalle de temps spécifié, le scheduler renouvelle une requête après avoir incrémenté le temps local de l'objet jusqu'à ce qu'un plan soit produit. Le temps local de chaque objet n'est donc pas nécessairement le temps actuel du scheduler.

**Procedure 4:** $next_plan(id_O)$

```
plan_O ← make_plan(id_O)
count_O ← count_O + 1
localtime_O ← localtime_O + interval_O
while not(plan_O) do
    if count_O < l then
        next_plan(id_O)
    else
        plan_O ← <localtime_O, id_O, next_plan(id_O)>
        count_O ← 0
    end
end
```

Durant ce procédé, le nombre de requêtes consécutives au planner doit être limité pour éviter une surcharge de calcul [10]. Nous notons $l \in \mathbb{N}^*$ cette limite. Si aucun plan n'a été trouvé et

10. Cet horizon temporel est choisi empiriquement de manière à maintenir la durée d'une planification faible par rapport à la précision temporelle souhaitée.

que la limite est atteinte, le planner produit automatiquement un plan singleton au temps local courant (et donc futur) pour renouveler une requête de planification. Cette stratégie peut être formalisée de manière récursive et est appliquée grâce à la fonction décrite dans la procédure 4.

Ici, étant donné que la hiérarchie est conservée, si un objet contient des objets enfants, sa lecture déclenche aussi leur lecture. Ceux-ci sont alors aussi « inscris » dans le scheduler, et doivent être exécutés. Le scheduler doit donc effectuer à la fois une traversée horizontale (sur les plans) et verticale (sur les objets, voir section 3.2.1). Ce fonctionnement est comparable à une structure multi-agents à multiples horizons temporels [18]. Cela peut impliquer des délais dans l'exécution des actions étant donné que le scheduler doit effectuer des tests sur tous les plans d'objets à chaque tour de boucle, même si ce n'est pas nécessaire. Pour palier ce problème, le registre d'objets peut être trié dynamiquement [8]. Si ce registre est trié par ordre croissant de date de prochaine action a exécuter, le scheduler doit seulement tester le premier objet du registre à chaque tour de boucle. Ce tri est réalisé par le scheduler lorsqu'il est au repos. Considérant que chaque plan d'objet $O_i$ est noté $P_{O_i} = (a_1, a_2, ..., a_n)$, le temps de repos du scheduler au temps $t$ peut être calculé selon la formule suivante :

$$idle_t = min(t^{a_1}, a_1 \in P_{O_i} | O_i \in Register) - t$$

A chaque instant, si une opération de tri a besoin d'être réalisée, un test vérifie sa faisabilité durant le temps $idle_t$ [11].

## 3.3 Révision automatique des plans

Le scheduler doit prendre en compte les modifications dynamiques sur la partition durant son exécution. Ces modifications peuvent aussi bien provenir de l'environnement extérieur, de l'interface utilisateur ou des actions des plans d'objets qui eux mêmes peuvent être programmés pour agir sur la partition (voir section 3.3.2). Pour ce faire, nous définissons dans cette section l'architecture réactive pour la composition décrite en figure 8. Celle ci est composée d'une représentation « haut niveau » des données musicales (structure de donnée et interface utilisateur), d'un planner et scheduler décris en section 3.2, et d'une « *thread-pool* » (voir section 3.3.2).

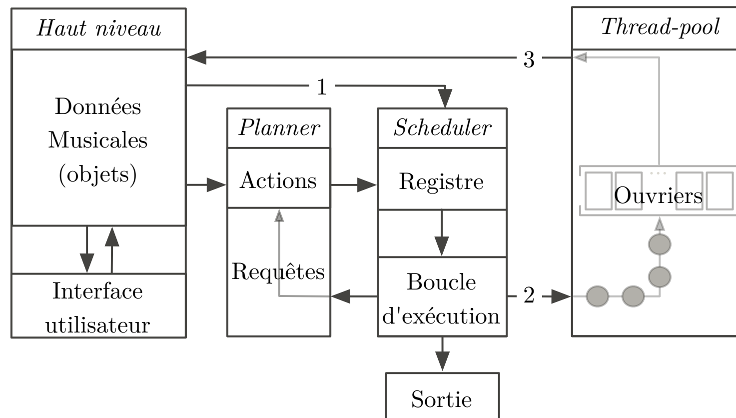


FIGURE 8 – Architecture globale

11. La faisabilité d'un tel tri est décidée en comparant le temps disponible $idle_t$ à la pire durée théorique du tri (obtenue à partir de la complexité de l'algorithme et de la fréquence du processeur).

### 3.3.1 Edition des données et re-planification

Lors de l'application d'une modification sur la partition, le moteur d'exécution doit réagir de manière adéquate pour la prendre en compte dans le rendu. En effet, certaines modifications peuvent engendrer une re-planification, et d'autres non.

Nous présentons dans la procédure 5 une fonction permettant de supprimer le plan partiel d'un objet et de l'actualiser à partir d'un temps $t$. En considérant la planification par intervalle

**Procedure 5:** $reset_plan(id_O, t)$

$localtime_O \leftarrow t$
$next_plan(id_O)$

décrite en section 3.2.2, il est convenable dans notre cas d'étude musical de ne pas actualiser le plan partiel d'un objet si celui-ci correspond à une fenêtre temporelle de l'ordre de quelques millisecondes [12]. Nous présentons une fonction ***edit_ content*** permettant d'associer une réaction appropriée du moteur d'exécution à toute modification d'un objet $O$. Cette fonction est appelée par l'objet lui même à la suite de toute modification de son contenu si $O \in registers_S$ (dans le cas contraire, aucune révision de la planification n'est nécessaire). Le résultat de la fonction ***edit_ content*** est un 2-tuple de la forme $<id_O, w>$ (nommé « requête ») où :

— $id_O$ est l'identifiant de l'objet ayant subit une modification,
— $w$ est l'intervalle temporelle, interne à l'objet, concernée par la modification.

Dans la définition suivante, $t1$ et $t2$ correspondent aux dates minimales des éléments du contenu modifié d'un objet $O$ si elles peuvent être déterminées par le système [13]. Nous définissons le comportement de la fonction ***edit_ content*** par les équations suivantes :

$$\boldsymbol{edit_\ content}(id_O, t1, t2) = \ <id_O, [t1; t2]>$$
$$\boldsymbol{edit_\ content}(id_O, null, null) = \ <id_O, [0; \infty[>$$

Toute requête de la forme $<id^i, w>$ générée est automatiquement exécutée par la fonction $\boldsymbol{process}_S$ définie comme suit :

$$\boldsymbol{process}_S(<id_O, w>) = \begin{cases} \boldsymbol{reset_\ plan}(id_O, t_{min}) & si\ w \cap [t_{min}; t_{max}[\ \neq \emptyset \\ \emptyset & sinon \end{cases}$$

où $t_{min}$ est le temps actuel du scheduler et $t_{max} = max(t^{a_i} | a_i \in plan_O)$.

Grâce à ce système de génération de requêtes, le scheduler réagit automatiquement à une modification de tout objet $O$ (voir 1 en figure 8), en ne déclenchant des calculs supplémentaires seulement si nécessaire. Que ces modifications proviennent de l'environnement extérieur ou du système lui-même ne change pas ce comportement.

### 3.3.2 Modification autonome de la partition

Une modification interne est définie comme une modification provenant des objets eux mêmes. Par exemple, un objet peut être programmé pour que, lors de sa lecture par le moteur d'exécution, il déclenche un calcul construisant un nouvel objet et l'intégrant à la partition.

12. Auquel cas la prochaine planification est prévue suffisamment tôt pour considérer le plan partiel courant comme définitif sans impacter la cohérence musicale (du point de vue de la perception temporelle).
13. Certaines modifications ne contiennent pas de datation explicite. Dans ce cas, $t1$ et $t2$ sont *null*.

Dans ce cas, une action du plan partiel de cet objet consiste en un calcul s'étalant sur une durée non négligeable par notre système. Son exécution directe dans la boucle d'exécution entrainerait une interruption de la lecture de tout objet durant la progression du calcul.

Pour cela nous introduisons une « *thread-pool* » [16], structure contenant des threads « *ouvriers* » pouvant exécuter des « *tâches* » en logique best-effort et non-préemptibles [14]. Une tâche est une sous-structure d'action, possédant une fonction de rappel permettant de rediriger le résultat du calcul une fois produit. La création d'une tâche est décidée dans la boucle d'exécution du scheduler : dans le cadre de l'informatique musicale, nous disposons d'un ensemble de fonctions connues pouvant être exécutées en un temps négligeable (messages MIDI, OSC etc.). Si une action $a_i$ possède une fonction $f^{a_i}$ non connue comme instantanée, une tâche est créée et soumise à la thread-pool (voir 2 en figure 8).

Une fois une tâche entièrement exécutée par un ouvrier, sa fonction de rappel est appelée avec le résultat produit. Celui-ci est alors intégré aux données musicales, et donc à des objets en cours de lecture (voir 3 en figure 8). Une requête est alors potentiellement générée comme décrit en section 3.3.1.

# 4 Conclusions et Perspectives

Nous avons présenté un modèle réactif intégrant un système de planification et d'ordonnancement dynamique pour la composition, implanté au sein du logiciel de CAO OpenMusic. Cet environnement dédié à la composition est un langage de programmation visuel permettant aux compositeurs de concevoir des programmes générant des données musicales (partitions, partitions, sons etc.). Cette nouvelle implémentation permet de sortir du cadre de création « temps différé » et de créer des partitions pouvant être programmées de manière à évoluer automatiquement durant leur lecture et en interaction avec l'environnement extérieur. La structure de donnée proposée pour composer le matériel musical permet de faciliter les stratégies bas niveau tout en facilitant la manipulation à haut niveau. Le moteur d'exécution et notamment la planification par intervalle présentés permettent au système de prendre en compte les modifications de la partition sans toujours nécessiter de calculs.

La gestion des calculs de long terme traitée en section 3.3.2 permet de programmer des partitions complexes sans entrainer de retard dans son exécution. Cependant, nous ne traitons pas la question de l'ordonnancement de ces calculs et ne fournissons donc aucune garantie quant à la validité d'une partition : les résultats des tâches sont intégrés quand ils sont disponibles et donc potentiellement à un temps absolu supérieur à celui prévu par le compositeur. En l'état, il convient donc au compositeur de prévoir des marges temporelles entre les dates de déclenchement de calculs et les dates d'intégration de leurs résultats. De plus, nous considérons des environnements déployés dans des systèmes d'exploitation non temps-réel, et notre architecture est soumise à un ordonnancement à préemption plus bas niveau. Par conséquent, l'étude des WCET [2] pour proposer un ordonnancement optimal des tâches n'est envisageable que pour une optimisation empirique. Cependant, nos travaux futurs concernent l'évaluation de la validité temporelle des partitions en utilisant des approximations réalisées par des simulations « hors-temps », ou par des analyses automatiques de complexité de fonctions. Dans ce cadre, il conviendra également d'étudier une utilisation dynamique et adaptive du fenêtrage temporel des plans. Nous nous intéressons aussi à proposer des outils et interfaces nouvelles permettant d'offrir un maximum de contrôle à l'utilisateur sur les mécanismes sous-jacents.

14. Une tâche est considérée comme indépendante de la temporalité du scheduler. Les synchronisations inter-tâches et intégrations de résultats dans l'environnement doivent être décidées et gérées en dehors de leurs exécutions [12].

Notre premier cas d'application est réalisé avec le système d'improvisation ImproteK [19]. Ce moteur de « *génération guidée* » permet de créer du matériel musical à partir d'une base de donnée et d'un scénario. Les fragments musicaux produits s'étalent sur un plus ou moins long terme et mobilisent des calculs aux durées non négligeables. Notre modèle réactif permet à ImproteK de produire des anticipations qui peuvent être réécrites suite à des modifications de scénario ou de paramètres, tout en exécutant de manière continue et synchronisée les résultats. ImproteK était précédemment contrôlé grâce à une inter-communication entre OpenMusic, Antescofo et le logiciel temps-réel Max/MSP [20]. Son développement autonome dans OpenMusic est en adéquation avec son fonctionnement : ImproteK écrit des anticipations, qui peuvent être réécrites, ce qui s'apparente à un travail de composition. Dans l'approche de communication avec les logiciels cités précédemment, le scheduler mobilisé étant « temps-réel », seule une vision à court-terme était considérée dans l'exécution.

Nous initions également une étude sur l'habillage sonore automatique de jeux-vidéos. Actuellement, la majorité des jeux-vidéos utilisent des « mappings » [11] entre des données du jeu et une base de donnée d'échantillons à lire [21]. Nous pensons un modèle permettant à des paramètres haut niveau d'un jeu d'être liés à des paramètres musicaux pour contrôler des processus compositionnels complexes dont les résultats sont exécutés et modifiés en continu, permettant de composer automatiquement la musique plutôt que de la déclencher.

## 5 Remerciements

Ce travail est financé par l'Agence Nationale de la Recherche, sous le projet ANR-13-JS02-0004. Les auteurs remercient Jean-Louis Giavitto, José Echeveste, Jérôme Nika et Romain Michon pour leurs relectures et suggestions.

## Références

[1] Andrea Agostini and Daniele Ghisi. Real-time computer-aided composition with bach. *Contemporary Music Review*, 32(1) :41–48, 2013.

[2] Sebastian Altmeyer and Gernot Gebhard. WCET Analysis for Preemptive Scheduling. In *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008.

[3] K. Barbar, M. Desainte-Catherine, and A. Miniussi. The Semantics of Musical Hierarchies. *Computer Music Journal*, 17(4), 1993.

[4] Dimitri Bouche and Jean Bresson. Planning and Scheduling Actions in a Computer-Aided Music Composition System. In *Proceedings of the Scheduling and Planning Applications woRKshop (SPARK)*, Jerusalem, Israel, 2015.

[5] Jean Bresson and Carlos Agon. Temporal control over sound synthesis processes. In *Sound and Music Computing*, Marseille, France, Mai 2006.

[6] Jean Bresson, Carlos Agon, and Gérard Assayag. OpenMusic. Visual Programming Environment for Music Composition, Analysis and Research. In *ACM MultiMedia 2011 (OpenSource Software Competition)*, Scottsdale, AZ, USA, 2011.

[7] Jean Bresson and Jean-Louis Giavitto. A Reactive Extension of the OpenMusic Visual Programming Language. *Journal of Visual Languages and Computing*, 4(25) :363–375, 2014.

[8] Roger Dannenberg. Real-Time Scheduling and Computer Accompaniment. In MIT Press, editor, *Current Directions in Computer Music Research*, volume 225-261, 1989.

[9] Marie E. desJardins, Edmund H. Durfee, Jr. Charles L. Ortiz, and Michael J. Wolverton. A Survey of Research in Distributed, Continual Planning. *AI Magazine*, 20(4), 1999.

[10] José Echeveste, Jean-Louis Giavitto, and Arshia Cont. A Dynamic Timed-Language for Computer-Human Musical Interaction. Research Report RR-8422, Inria, December 2013.

[11] Rob Hamilton. *Perceptually coherent mapping schemata for virtual space and musical method*. PhD thesis, Stanford University, Dept. of Music, 2014.

[12] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto : A time-triggered language for embedded programming. In *Proceedings of the IEEE*, volume 91, pages 84–99, 2003.

[13] J. Hernandez and J. Tores. Electromechanical design : Reactive planning and control with a mobile robot. In *Proceedings of the 10th Electrical Engineering, Computing Science and Automatic Control (CCE) Conference*, pages 13–18, October 2013.

[14] L.P. Kaelbling and T. Lozano-Perez. Hierarchical task and motion planning in the now. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1470–1477, Shanghai, China, May 2011.

[15] Mark Kahrs. Dream chip 1 : A timed priority queue. *IEEE Micro*, 13(4) :49–51, July 1993.

[16] Ronald Kriemann. *Implementation and Usage of a Thread Pool based on POSIX Threads*. Max-Planck-Institue for Mathematics in the Sciences, Inselstr. 22-26, D-04103 Leipzig, Germany, October 2004.

[17] Fernando López, Dietmar Jannach, JoséM. Martínez, Christian Timmerer, Narciso García, and Hermann Hellwagner. Bounded non-deterministic planning for multimedia adaptation. *Applied Intelligence*, 36(1) :29–60, 2012.

[18] Conor McGann, Frédéric Py, Kanna Rajan, and Angel Garcia olaya. Integrated planning, execution and estimation for robotic exploration. In *Proceedings of the HYCAS 2009 workshop*, 2009.

[19] Jérôme Nika and Marc Chemillier. Improvisation musicale homme-machine guidée par un scénario temporel. *Technique et Science Informatique, Numéro Spécial Informatique musicale*, 7(33) :651–684, 2015.

[20] Jérôme Nika, José Echeveste, Marc Chemillier, and Jean-Louis Giavitto. Planning Human-Computer Improvisation. In *International Computer Music Conference*, pages 1290–1297, Athens, Greece, 2014.

[21] Durval Pires, Valter Alves, and Licinio Roque. A software architecture for dynamic enhancement of soundscapes in games. In *Proceedings of the 9th Audio Mostly : A Conference on Interaction With Sound*, AM '14, pages 27 :1–27 :8, New York, NY, USA, 2014. ACM.

[22] Eric Cesar Jr. Vidal and Alexander Nareyek. A real-time concurrent planning and execution framework for automated story planning for games. In *AAAI Technical Report WS-11-18*, 2011.

[23] Matthew Wright, Adrian Freed, and Ali Momeni. Open sound control : State of the art 2003. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 153–159, Montreal, 2003. CNMAT. OpenSound Control.