CAGE: UNA LIBRERIA DI ALTO LIVELLO PER LA COMPOSIZIONE ASSISTITA DA COMPUTER IN TEMPO REALE

Andrea Agostini
HES-SO, Ginevra
Conservatorio di Cuneo
and.agos@gmail.com

Éric Daubresse
HES-SO, Ginevra
eric.daubresse@hesge.ch

Daniele Ghisi
HES-SO, Ginevra
danieleghisi@gmail.com

SOMMARIO

Questo articolo è un'introduzione a *cage*, una libreria per l'ambiente Max [1] costituita da moduli di alto livello per la composizione assistita da computer (CAC). La libreria, che al momento è ancora nella fase denominata *alpha*, è composta da un insieme di strumenti che facilitano la manipolazione dei dati musicali simbolici e che risolvono una serie di problemi tipici della CAC, come ad esempio generazione di note o di profili melodici, interpolazione armonica o ritmica, operazioni ispirate a processi tipici del *digital signal processing*, automi cellulari e L-sistemi, moduli per la *musical set theory*, strumenti per la generazione e la gestione di partiture. Questo progetto, supportato dalla *Haute École de Musique* di Ginevra, ha una connotazione spiccatamente pedagogica: tutti i moduli nella libreria sono astrazioni, che si prestano quindi ad essere facilmente analizzate e modificate.

1. INTRODUZIONE

Questo articolo descrive alcuni dei concetti principali e dei componenti della libreria *cage* [2] per Max, contenente diversi moduli di alto livello per la composizione assistita da computer (CAC). In questo articolo, versione italiana di [1], completiamo la visione d'insieme della libreria e forniamo una panoramica sui suoi obiettivi.

cage è interamente basata sulla libreria *bach: automated composer's helper*, che è sviluppata da due degli autori [2, 3]. *bach* è una libreria di circa 200 oggetti e astrazioni finalizzata a dotare i Max un insieme di 'primitive' per la manipolazione di dati simbolici musicali, insieme ad alcune interfacce grafiche per la loro rappresentazione e manipolazione. All'interno di *bach*, i dati sono rappresentati costantemente attraverso usi specializzati di una generica struttura chiamata *llll* ('Lisp-like linked list') che è essenzialmente, come l'acronimo suggerisce, una struttura ad albero nella forma di una lista annidata, ispirata direttamente dal linguaggio di programmazione Lisp.

[1] *http://cycling74.com*
[2] *www.bachproject.net/cage*

Di conseguenza, i moduli di *bach* sono per lo più strumenti per la manipolazione a basso livello di *llll* (per esempio per compiere rotazioni, sostituzioni, inversioni...) o per operazioni più complesse ma essenzialmente basilari, come la risoluzione di problemi a vincoli o la quantizzazione ritmica.

Diversamente da *bach*, i moduli di *cage* compiono in generale operazioni di più alto livello, con una connotazione più musicale che tecnica (ad esempio: generazione di materiale melodico, o calcolo di una modulazione di frequenza simbolica). In ogni caso, *cage* eredita da *bach* alcuni meccanismi e principi fondamentali, come ad esempio il fatto che la comunicazione tra moduli differenti avviene, la maggior parte delle volte, attraverso *llll*.

Due criteri principali hanno presieduto all'ideazione e allo sviluppo della libreria.

Il primo è l'idea che si trova all'origine di *cage*: la volontà di costruire una libreria di moduli pronti all'uso, che implementino un certo numero di processi di CAC largamente utilizzati. Di conseguenza, una parte della libreria è ispirata apertamente a librerie già esistenti per altri programmi (in particolare le librerie *Profile* [4] ed *Esquisse* [5, 6] per *Patchwork*, portate successivamente in *OpenMusic*); un'altra parte della libreria è invece esplicitamente dedicata a problemi tipicamente associati con interazioni in tempo reale (come ad esempio *cage.granulate*, il motore di granulazione simbolica).

Il secondo criterio è una forte connotazione pedagogica [3]: tutti i moduli della libreria sono astrazioni, che si prestano quindi ad essere facilmente analizzate e modificate. Non è difficile, per l'utilizzatore che voglia imparare a trattare i dati musicali, copiare, modificare o ritoccare le patch per le proprie necessità. In quest'ottica, tutti gli oggetti della libreria sono intrinsecamente *open source*: sebbene ciascun processo implementato sia concepito per un utilizzo sostanzialmente standard, l'utilizzatore avanzato potrà partire facilmente da queste astrazioni e modificarne il comportamento. Questa vocazione pedagogica è completata dal fatto che la libreria sarà meticolosamente documentata tramite file di aiuto, fogli di riferimento e una collezione di tutorials.

[3] La libreria *cage* è supportata dalla HES-SO.

2. UN APPROCCIO IN TEMPO REALE ALLA COMPOSIZIONE ASSISTITA DA COMPUTER

Il paradigma del tempo reale influenza profondamente la natura stessa del processo compositivo. Per esempio, compositori che lavorano nel dominio della musica elettroacustica spesso hanno bisogno che il computer si adatti immediatamente a ogni cambiamento di parametri. Similmente, compositori che lavorano con dati simbolici potrebbero volere che il computer si adatti nel più breve tempo possibile alla nuova configurazione di dati. Il paradigma che soggiace a *cage* è lo stesso che ha improntato la libreria *bach*: creare e modificare dati simbolici non è necessariamente un'attività fuori dal tempo, ma segue il flusso temporale compositivo, e si adatta ad esso (si veda anche [3, 7, 8]).

3. COMPOSIZIONE DELLA LIBRERIA

La libreria *cage* è composta da diverse famiglie di moduli, che descriveremo più dettagliatamente nei prossimi paragrafi, non prima di aver però sottolineato che il progetto non ha ambizioni di completezza. La composizione assistita da computer è un dominio molto vasto, e le pratiche sono solitamente personali: le specificità del singolo compositore spesso prevalgono sulla messa a punto di procedimenti standardizzati. In ogni caso, ci è sembrato di poter identificare, come linea guida per la formazione della libreria, alcune tipologie generale di approccio e alcune operazioni specifiche di uso ormai comune. Tentiamo di esemplificare almeno alcune di queste operazioni, sperando che il nostro lavoro sia utile anche ai compositori che volessero partire da esse per implementare i propri processi ed operazioni individuali.

3.1 Generazione di altezze

La prima famiglia di moduli che analizziamo è dedicata alla generazione di altezze, secondo diversi criteri: *cage.scale* e *cage.arpeggio* generano, rispettivamente, scale ed arpeggi all'interno di un'estensione data. Si può specificare il tipo di accordo o di scala sia tramite un nome simbolico (p.e. **F#m**, **ReM**), sia tramite un pattern di *midicents*[4] . I nomi di scale e accordi possono anche contenere alterazioni quartitonali. *cage.harmser* genera invece serie armoniche, a partire da una fondamentale data, con un fattore di distorsione opzionale.

Mentre per i moduli appena citati la generazione di altezze dà come risultato l'intera scala, arpeggio o serie, altri moduli si occupano invece di generare le altezze ad una ad una, in tempo reale: *cage.noterandom* genera aleatoriamente altezze, a partire da un insieme di altezze dato (prendendo anche in considerazione pesi di probabilità opzionali, che possono essere a loro volta generati ad esempio attraverso *cage.weightbuilder*); *cage.notewalk* genera un cammino aleatorio all'interno di un insieme di altezze, data una lista di passi permessi. In entrambi i casi il risultato dell'operazione è progettato per essere utilizzato in combinazione con *bach.transcribe*, che trascriverà in tempo reale il flusso di note in entrata. Inoltre l'elemento selezionato volta per volta può essere convalidato o scartato dall'utente tramite un *lambda loop*.[5]

3.2 Generazione e trattamento di profili melodici

Una famiglia di moduli è specificatamente dedicata alla generazione e al trattamento di profili melodici, in maniera esplicitamente ispirata dalla libreria *Profile* di OpenMusic e PatchWork [4]. Una funzione può essere convertita in una sequenza di altezze (un profilo melodico) grazie a *cage.profile.gen*. Questo profilo può quindi essere modificato in varie maniere: può essere compresso o espanso (con *cage.profile.stretch*), invertito (con *cage.profile.mirror*), approssimato a una griglia armonica o a una scala (con *cage.profile.snap*), costretto all'interno di una certa estensione (con *cage.profile.rectify*), perturbato aleatoriamente (con *cage.profile.perturb*) o filtrato (con *cage.profile.filter*). Il filtraggio del profilo avviene tramite l'applicazione di un filtro mediano, medio o personalizzato - quest'ultimo è definibile dall'utente tramite un *lambda loop* (si veda la Fig. 1).

3.3 Processi ispirati da pratiche elettroacustiche

cage contiene un gruppo di moduli dedicati all'emulazione simbolica di processi appartenenti al dominio della sintesi audio o del *digital signal processing*.

cage.freqshift permette la trasposizione di materiale linearmente sull'asse delle frequenze, come in un modulatore ad anello a banda laterale singola. A causa della stretta relazione tra i due processi, *cage.pitchshift* viene considerato come appartenente alla stessa categoria, benché un'operazione di *pitch shifting* applicata alla notazione musicale corrisponda a una semplice trasposizione.

cage.rm e *cage.fm* calcolano rispettivamente modulazioni ad anello e modulazioni in frequenza. L'idea alla base di tali tecniche, largamente utilizzate da compositori legati allo spettralismo, è la seguente: partendo da due accordi (un accordo 'portante' e uno 'modulante'), le cui note sono assimilate a semplici onde sinusoidali, viene calcolato lo spettro ottenuto modulando reciprocamente questi due gruppi di sinusoidi. Ogni componente dello spettro risultante è quindi rappresentato come una nota nell'accordo risultante. Questa operazione richiede un insieme di approssimazioni e compromessi che possono rendere il risultato significativamente diverso dal comportamento corrispondente in un contesto di trattamento audio: in ogni caso, si tratta di un approccio efficace per generare ricche

[4] *cage*, come *bach*, adotta la convenzione di *Patchwork* e *OpenMusic* di esprimere le altezze e gli intervalli in midicents, ovvero centesimi di nota MIDI.

[5] Un *lambda loop* in *bach* e *cage* è una configurazione di feedback simbolico: i moduli che supportano tale comportamento hanno uno o più *'lambda'* outlet dedicati, che forniscono i dati che devono essere accettati o modificati; tali dati sono quindi processati in una specifica sezione della patch, e il risultato è reimmesso in un *'lambda'* inlet dedicato del modulo originale. Questa configurazione è utilizzata all'interno di *bach* per definire comportamenti personalizzati per operazioni specifiche come ad esempio un criterio di ordinamento, o un processo che debba essere applicato a ogni elemento di una *llll*. Il nome *'lambda'* fa riferimento al fatto che questa configurazione, in un certo senso, permette di passare una sezione di patch come pseudo-argomento di un modulo. Si tratta tuttavia di una mera allusione: nel processo non entrano in gioco né lambda calcolo né funzioni interpretate.

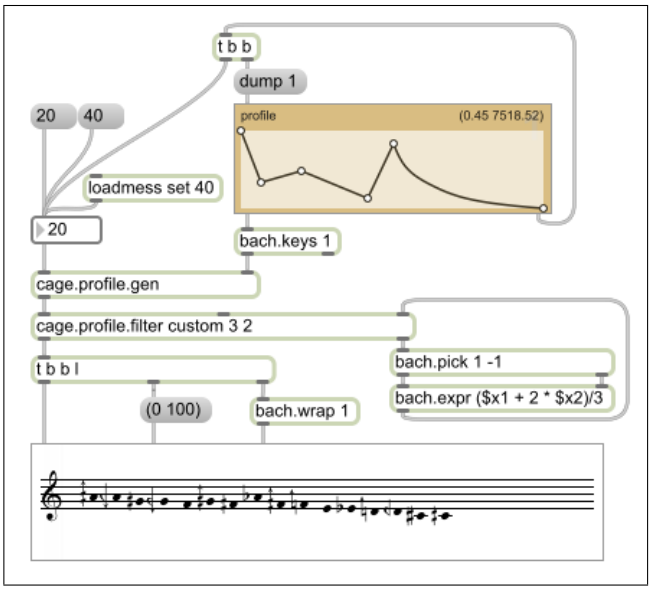

Figura 1. Un profilo melodico è costruito da una funzione definita in un oggetto *bach.slot* e campionata su 20 punti. Il profilo è quindi filtrato da un processo espresso tramite un *lambda loop*, che opera su finestre di tre note; ogni finestra è sostituita con un singolo valore, la media del primo ed ultimo elemento della finestra stessa, pesata con i pesi $(1, 2)$. Questo processo di filtraggio è ripetuto due volte. Si può notare che, a causa del finestraggio, il risultato contiene quattro note in meno del campionamento originale.

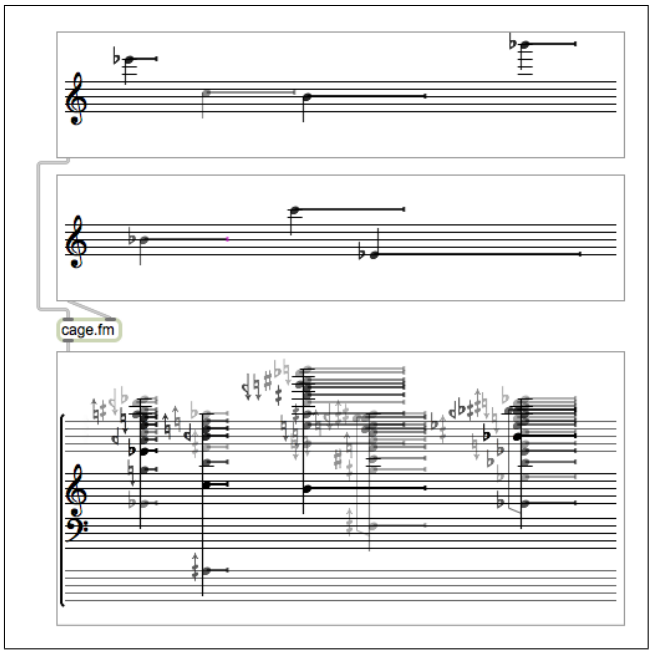

Figura 2. Un esempio di modulazione di frequenza di due partiture, ottenuta attraverso l'astrazione *cage.fm*. La partitura 'portante' e 'modulante' sono in alto, il risultato è in basso. La velocità MIDI delle note (trattata come ampiezza delle corrispettive componenti sinusoidali) è rappresentata in scala di grigi.

famiglie armoniche a partire da materiali semplici, e da ciò discende il suo interesse compositivo.

Benché *cage.rm* e *cage.fm* si ispirino esplicitamente alla libreria *Esquisse* [5, 6] per *OpenMusic*, il loro paradigma operativo e alcuni dettagli computazionali differiscono. In particolare, essendo concepiti per lavorare tenendo conto del parametro temporale, questi due moduli possono accettare non solo semplici accordi, ma anche partiture che rappresentano variazioni di accordi 'portanti' e 'modulanti' nel tempo. In questo caso, l'esito sarà una nuova partitura contenente il risultato di tali variazioni nel tempo (si veda la Fig. 2). Per quanto concerne l'effettivo calcolo interno, i due moduli compiono una stima delle opposizioni di fase generate dalle modulazioni, e cancellano le componenti interessate da tale opposizione, diversamente da quanto accade nella libreria *Esquisse*. Per questa ragione, i risultati dello stesso processo applicato nei due ambienti possono essere significativamente diversi.

cage.virtfun restituisce una o più stime della fondamentale virtuale di un accordo, che si può percepire, ad esempio, dopo l'applicazione di un processo di waveshaping. L'implementazione è estremamente semplice: si percorre la serie di subarmoniche della nota più grave dell'accordo, finché non viene trovata una frequenza le cui armoniche approssimano tutte le note dell'accordo, entro una certa tolleranza. *cage.virtfun* può essere anche applicato a sequenze di accordi nel tempo; in questo caso, il risultato sarà la sequenza delle fondamentali virtuali. D'altro canto, l'operazione numerica compiuta da *cage.virtfun* ha un ambito di applicazione più ampio: può essere considerata come il calcolo di un massimo comun denominatore approssimato di un insieme di numeri. Come tale, è richiamato ad esempio da *cage.accrall*, per stabilire un'unità minimale ritmica 'ragionevole' all'interno di una partitura in notazione proporzionale.

cage.delay e *cage.looper* estendono il concetto di linea di ritardo con feedback nel dominio simbolico. Il loro obiettivo è creare strutture ripetitive in cui il materiale possa essere alterato a ogni passaggio attraverso un *lambda loop*. La differenza tra i due moduli risiede nell'unità musicale che viene passata attraverso il *lambda loop*: un singolo accordo nel caso di *cage.delay*, un'intera sezione di partitura nel caso di *cage.looper*. In entrambi i casi, il tempo di ritardo può essere cambiato a ogni ripetizione. In linea di principio non c'è limitazione alla ricchezza dei processi che possono essere applicati al materiale nel *lambda loop*: il risultato musicale può essere quindi ben più complesso di una semplice iterazione.

cage.cascade∼ e *cage.pitchfilter* estendono il principio di filtraggio al dominio simbolico. Il primo applica a una partitura una catena di filtri a due poli e due zeri, analogamente agli oggetti Max *biquad∼* e *cascade∼*, emulando la risposta in frequenza di un filtro digitale con risposta all'impulso infinita. Il secondo opera direttamente sulle altezze, anziché sulle frequenze, applicando a una partitura un filtro definito da una funzione, ad esempio tramite un oggetto *function* o *bach.slot*. In entrambi i casi, la velocità MIDI di ogni nota è modificata a seconda della risposta del filtro, e le note con velocità al di sotto di una certa soglia sono rimosse. È anche possibile interpolare tra differenti configurazioni di filtri nel tempo (si veda la Fig. 3).

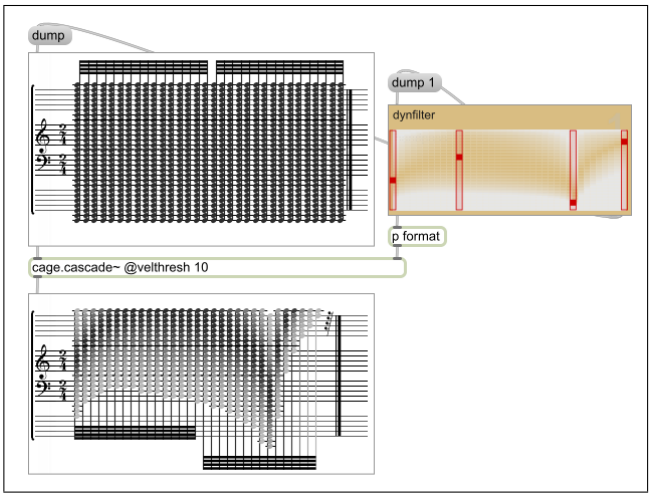

Figura 3. Un esempio di filtraggio dinamico di una partitura ottenuto attraverso *cage.cascade*$\sim$ pilotato da uno slot di tipo *dynfilter* in un oggetto *bach.slot*. Ogniqualvolta i parametri del filtro sono modificati tramite l'interfaccia, il risultato è aggiornato in tempo reale.

cage.granulate è un motore di granulazione simbolica. I parametri della granulazione sono gli stessi del corrispondente processo elettroacustico: l'intervallo di tempo tra due grani, la durata di ogni grano, la regione di partitura da cui i grani devono essere estratti. L'altezza e la durata dei singoli grani può essere modificata. Basandosi su questi parametri, *cage.granulate* riempie in tempo reale un oggetto *bach.roll* collegato al suo outlet.

3.4 Interpolazione armonica e ritmica, formalizzazione dell'agogica

L'astrazione *cage.chordinterp* opera un'interpolazione lineare all'interno di un insieme di accordi, attraverso l'assegnazione di differenti pesi per ciascuno di essi. Analogamente, un'interpolazione ritmica può essere ottenuta tramite il modulo *cage.rhythminterp*.

cage.timewarp, d'altro canto, opera una distorsione temporale di una partitura, ottenuta tramite una funzione (definita tramite *lambda loop*) da applicarsi a ogni onset di ogni evento discreto nella partitura. Tra le altre cose, questo meccanismo è abbastanza flessibile da permettere di ottenere qualsiasi forma di rallentando o accelerando, una volta definita la funzione appropriata - un compito che è facilitato dall'astrazione *cage.accrall*, che permette di esprimere l'agogica attraverso un insieme di parametri di alto livello, come la durata totale del processo o la velocità iniziale o finale.

3.5 Automi cellulari, L-sistemi, ecc.

L'astrazione *cage.chain* implementa automi cellulari monodimensionali e L-sistemi. Tale modulo opera riscritture di una data lista, a seconda di un insieme di regole definite dall'utente attraverso un messaggio o un *lambda loop*. Le sostituzioni possono avvenire sui singoli elementi (ad esempio: una certa lettera o nota viene sostituita da una lista di lettere o note), o su sequenze di elementi aventi

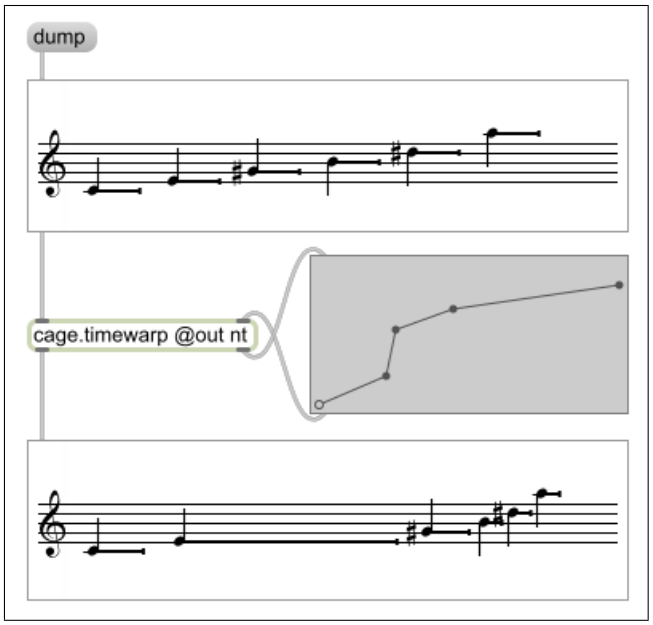

Figura 4. Un esempio di distorsione temporale ottenuta grazie a *cage.timewarp*. La funzione nel *lambda loop* associa il tempo nella partitura originale (in alto), rappresentato sull'asse x, al tempo nella partitura risultante (in basso), rappresentato sull'asse y.

lunghezza prefissata (ad esempio: ogni coppia di elementi è rimpiazzata da una certa altra sequenza); in quest'ultimo caso, *cage.chain* gestirà il comportamento ai bordi a seconda dei valori di alcuni specifici attributi (*pad*, *align*). In sintesi, questo modulo facilita la generazione di automi cellulari a dimensione 1, o di frattali ottenuti per sostituzione.

cage.life si occupa invece di automi cellulari a due dimensioni (il più famoso dei quali è il 'gioco della vita' di John Conway). Le regole per questi automi sono definite tramite *lambda loop*. La grandezza delle sottomatrici su cui dev'essere operata la sostituzione può essere definita dall'utente.

Un'astrazione strettamente connessa alle due precedenti è *cage.lombricus*, che implementa un meccanismo per costruire sistemi generativi basati su regole. Questo modulo accetta una serie di elementi iniziali, raggruppati in famiglie, con un peso associato ad ogni famiglia. Il compito dell'astrazione è creare una sequenza di un numero arbitrario di elementi, facendo in modo che il numero relativo di occorrenze di elementi di ogni famiglia rispetti la distribuzione dei pesi associati alle famiglie. Il *lambda loop* dell'astrazione fornisce le proposte di elementi che andrebbero incatenati alla sequenza esistente, insieme all'intera sequenza costruita fino a quel punto; ogni proposta può essere rifiutata o accettata con un certo punteggio assegnato in base a regole personalizzabili: tra gli elementi accettati, il 'vincitore' sarà scelto in base al punteggio e al peso della famiglia cui appartiene. Se a un certo punto non viene trovato alcun elemento idoneo, l'astrazione è in grado di tornare sui propri passi e sostituire un elemento scelto precedentemente con uno diverso, avente punteggio più basso ma potenzialmente in grado di permettere la

costruzione di una catena più lunga. È necessario sottolineare che gli elementi non devono necessariamente essere copiati letteralmente nella sequenza risultante: ad esempio, l'utente può voler dare al sistema un insieme di intervalli come elementi iniziali, e ottenere una sequenza melodica come risultato del processo: la sostituzione può avvenire all'interno del *lambda loop* descritto in precedenza. In sintesi, il meccanismo soggiacente a *cage.lombricus* condivide alcune caratteristiche con automi cellulari e L-sistemi da un lato (in particolare un comportamento basato su regole di sostituzione costruttive) e con la programmazione a vincoli dall'altro (la capacità di prendere decisioni basate su pesi, e la possibilità per l'algoritmo di tornare sui suoi passi), senza appartenere strettamente ad alcuna delle due categorie. Sebbene questo meccanismo possa apparire complicato, un'analisi accurata delle pratiche compositive nostre e di altri compositori (primo fra tutti Michaël Jarrell) suggerisce che tale procedimento possa essere ben adatto a modellizzare una grande quantità di tecniche di formalizzazione musicali.

3.6 Strumenti per la *set theory*

Un gruppo di moduli di *cage* si occupa di rappresentazioni tipiche della *set theory* musicale, ossia della teoria matematica, elaborata in origine negli Stati Uniti a partire dagli anni Sessanta, per categorizzare gli oggetti musicali (agglomerati armonici in primis) e descriverne le relazioni. Due moduli di *cage*, *cage.chroma.topcset* e *cage.chroma.frompcset*, operano conversioni tra insiemi di *pitch class* e vettori di *chroma* (si veda [9]); altri due moduli, *cage.chroma.tocentroid* e *cage.chroma.fromcentroid*, operano conversioni tra i vettori di *croma* e centroidi tonali, ottenuti tramite la trasformata descritta da Harte and Sandler in [10]. Il calcolo del centroide tonale a partire dal vettore di *chroma* implica una perdita d'informazione; la conversione inversa quindi non è univoca: l'algoritmo restituisce un singolo vettore di *chroma*, tra tutti quelli aventi il vettore in input come centroide.

3.7 Partiture

cage contiene un gruppo di moduli per il trattamento globale di partiture: *cage.rollinterp* interpola tra il contenuto di due oggetti *bach.roll*, a seconda di una curva di interpolazione o di un singolo valore (nel caso di interpolazione statica). *cage.envelopes* aiuta a sincronizzare una famiglia di funzioni alla durata totale della partitura, rendendo più agevole la modifica in tempo reale della partitura a seconda dei valori delle funzioni a ogni istante. *cage.scissors* divide la partitura contenuta in un oggetto *bach.roll* in base a punti di divisione verticali (tempo) e orizzontali (voci); l'astrazione restituisce quindi una matrice contenente in ogni cella il corrispettivo estratto di partitura. *cage.glue* compie l'operazione inversa: riempie un singolo *bach.roll* riassemblando una matrice contenente partiture più piccole, in base alla disposizione verticale (tempo) e orizzontale (voci) implicita nella matrice stessa o definita esplicitamente. *cage.ezptrack* parte da una sequenza di accordi e tenta di ricostruire delle voci musicali, in maniera simile a quanto gli algoritmi di *partial tracking* fanno con l'analisi dei dati armonici.

3.8 Supporto per file SDIF

Una famiglia di moduli di *cage* è progettata per facilitare la lettura e la scrittura di file di analisi in formato SDIF [11, 12]. Questa famiglia contiene sottofamiglie per alcuni dei più comuni tipi di analisi e descrittori, in particolare frequenza fondamentale, picchi, *partial tracking*, *markers*.

A partire dalla versione 0.7.4, *bach* supporta la lettura e la scrittura di file SDIF attraverso gli oggetti *bach.readsdif*, uno strumento di basso livello che legge tutta l'informazione contenuta in un file SDIF e la converte in una *llll*, e *bach.writesdif*, che permette di scrivere file in formato SDIF a partire dalla loro rappresentazione come *llll*. Questa rappresentazione è completa: se colleghiamo l'output di *bach.readsdif* all'input di *bach.writesdif* otteniamo un file SDIF equivalente, quando non identico, all'originale. D'altro canto, questa completezza rende la manipolazione della rappresentazione stessa più farraginosa.

Per questa ragione *cage* annovera un insieme di moduli che implementano operazioni basilari sul contenuto dei file SDIF. Alcuni moduli convertono i dati SDIF in una sintassi direttamente comprensibile da *bach.roll*, per esempio *cage.sdif.ptrack.toroll* (si veda la Fig. 5). Altre astrazioni ristrutturano i dati di analisi in una forma più accessibile. A titolo di esempio, *cage.sdif.fzero.unpack* filtra i *frame* di tipo 1FQ0 (stime di frequenze fondamentali) e restituisce gli onset, le frequenze, i valori di confidenza, i punteggi e le ampiezze dai suoi diversi outlet sotto forma di *llll* strutturate con una sottolista per ogni *stream* SDIF. Due astrazioni gestiscono le rappresentazioni di *partial tracking* (*cage.sdif.ptrack.resolve* e *cage.sdif.ptrack.assemble*), permettendo la conversione tra una rappresentazione strutturata per istanti di tempo a una rappresentazione strutturata per indici delle voci.

In generale, non abbiamo considerato come scenario di utilizzo frequente la scrittura di un file SDIF a partire da una partitura, con una sola importante eccezione: i *marker*. Per questa ragione, la sola astrazione che fornisce una traduzione diretta da un oggetto di notazione a una *llll* contenente l'informazione SDIF è *cage.sdif.markers.fromroll*, che esporta tutti i *marker* di un oggetto *bach.roll*, ciascuno con la propria posizione nel tempo e il proprio nome.

3.9 *Rendering* audio

In aggiunta ai moduli di composizione assistita precedentemente elencati, *cage* contiene un insieme di utilità che mirano a rendere più semplice la prototipazione e la verifica delle soluzioni musicali. In particolare, due moduli di *cage* si occupano di *rendering* audio di partiture *bach*: *cage.ezaddsynth~* (un motore di sintesi additiva elementare) e *cage.ezseq~* (un campionatore elementare). Come *bach.ezmidiplay*, entrambi sono progettati per essere connessi direttamente all'uscita 'playout' di *bach.roll* o *bach.score*.

Il motore di sintesi additiva risponde all'esigenza di un *rendering* audio essenziale, che possa ovviare alle limitazioni degli strumenti MIDI: ciò può essere utile ad esem-

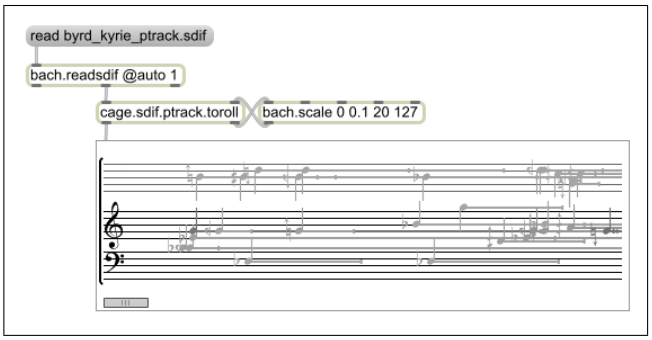

Figura 5. Un'analisi di *partial tracking* contenuta in un file SDIF è importata in un *bach.roll* attraverso *cage.sdif.ptrack.toroll*. Il *lambda loop* è usato per definire una mappatura personalizzata sulle velocità MIDI (se un *lambda loop* non è definito, l'astrazione utilizza una mappatura di default).

pio quando si lavora con griglie microtonali non standard, o quando gli inviluppi di ampiezza e panning o i glissandi non possono essere trascurati. Gli inviluppi devono essere definiti all'interno di *slot*. [6]

Il campionatore risponde all'esigenza di utilizzare *bach.roll* e *bach.score* come '*sequencer* aumentati': *cage.ezseq∼* considera il nome del campione da utilizzare, gli inviluppi di ampiezza, il panning, la velocità di riproduzione, eventuali filtri audio e il punto di inizio all'interno del campione (tutti definiti all'interno di *slot*). *cage.ezseq∼* è inoltre capace di pre-caricare in memoria i file audio, se una data cartella è assegnata. Se richiesto, *cage.ezseq∼* può trasportare ogni campione senza alterazione temporale (attraverso l'oggetto Max *gizmo∼*), in dipendenza dall'altezza della nota associata.

4. CONCLUSIONI

Nel momento in cui questo articolo viene scritto, una versione 'alpha' della libreria è disponibile [7] : non tutte le funzionalità descritte sono implementate, e la documentazione non è ancora completa. Ciononostante, la maggior parte dei moduli è già perfettamente utilizzabile. La prima versione completa della libreria sarà resa pubblica in ottobre 2014, all'occasione di una presentazione che avrà luogo a Ginevra, e sarà distribuita gratuitamente. A partire dall'anno accademico 2014-2015, *cage* sarà insegnata all'interno dei corsi di composizione e musica elettronica della *Haute École de Musique* di Ginevra, e in altre istituzioni partner.

4.1 Ringraziamenti

cage è un progetto di ricerca che nasce all'interno del centro di musica elettroacustica della *Haute École de Musique* di Ginevra, supportato dalla divisione musica e arte della *Haute École Specialisée* della Svizzera occidentale. Il nome *cage*, che nel contesto della denominazione anglosassone delle note rappresenta la famosa espansione di *bach*, è anche un acronimo che riconosce questo supporto: *composition assistée Genève*.

[6] Gli *slot* sono contenitori per metadati di diversa natura, associabili individualmente a ogni singola nota (si veda [2]).

[7] Si può ottenere la libreria dal sito www.bachproject.net/cage

5. BIBLIOGRAFIA

[1] A. Agostini, E. Daubresse, and D. Ghisi, "cage: a high-level library for real-time computer-aided composition," in *Proceedings of the International Computer Music Conference (ICMC 2014)*, (Athens, Greece), 2014, to appear.

[2] A. Agostini and D. Ghisi, "*bach*: an environment for computer-aided composition in Max," in *Proceedings of the International Computer Music Conference (ICMC 2012)*, (Ljubljana, Slovenia), pp. 373–378, 2012.

[3] A. Agostini and D. Ghisi, "Real-time computer-aided composition with *bach*," *Contemporary Music Review*, no. 32 (1), pp. 41–48, 2013.

[4] M. Malt and J. B. Schilingi, "Profile - libreria per il controllo del profilo melodico per Patchwork," in *Proceedings of the XI Colloquio di Informatica Musicale (CIM)*, (Bologna, Italia), pp. 237–238, 1995.

[5] J. Fineberg, "Esquisse - library-reference manual (code de Tristan Murail, J. Duthen and C. Rueda)," 1993.

[6] R. Hirs and B. G. editors, *Contemporary compositional techniques and OpenMusic*. Delatour/Ircam, 2009.

[7] A. Cont, *Modeling Musical Anticipation*. PhD thesis, University of Paris 6 and University of California in San Diego, 2008.

[8] M. Puckette, "A divide between 'compositional' and 'performative' aspects of Pd," in *Proceedings of the First Internation Pd Convention*, (Graz, Austria), 2004.

[9] M. Müller, *Information Retrieval for Music and Motion*. Springer Verlag, 2007.

[10] C. Harte, M. S, and M. Gasser, "Detecting harmonic change in musical audio," in *In Proceedings of Audio and Music Computing for Multimedia Workshop*, 2006.

[11] M. Wright, R. Dudas, S. Khoury, R. Wang, and D. Zicarelli, "Supporting the sound description interchange format in the max/msp environment," in *Proceedings of the International Computer Music Conference*, 1999.

[12] M. Wright, A. Chaudhary, A. Freed, S. Khoury, and D. Wessel, "Audio applications of the sound description interchange format standard," in *Proceedings of the Audio Engineering Society 107th Convention*, 1999.