UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE NATURALI

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE DELLA COMUNICAZIONE MUSICALE

Plugin di esportazione da formato IEEE 1599 a MIDI

Relatore: prof. Luca A. Ludovico

Co-relatore: prof. Adriano Baratè

Tesi di:

Vittorio Soana

matr. 71002

A.A. 2008/2009

Indice

- Capitolo 3 -

- Introduzione -

Si è soliti immaginare l'esecuzione di una partitura da parte di un musicista.
Solo da quando sono stati creati gli elaboratori si sono sviluppate tecniche e anche composizioni indirizzate a questo strumento; la Computer Music si sviluppa inizialmente negli USA per merito di ricercatori, primo fra tutti Max Mathews, che nei laboratori della società telefonica Bell, a partire dal 1958, comincia ad applicare la sintesi dei suoni musicali mediante elaboratore e redige i primi programmi (music 4, music 5) destinati a divenire il fondamento di numerosi pezzi musicali, negli anni '60 e oltre. Uno dei primi obbiettivi che si propone è quello di sviluppare sistemi di composizione automatica, ma il problema di formalizzare un complesso modello logico matematico, capace di comporre musica, non è certo dei più semplici, quindi la composizione resta un aspetto secondario rispetto alla ricerca tecnologica, fino a quando i lavori della Bell Labs arrivano, alla fine degli anni '60, a Stanford , dove lavora John Chowning, musicista, che realizza un brano musicale sintetizzato con la tecnica, già utilizzata nel campo della comunicazione, della modulazione di frequenza: “Terenas”. Da questi primi input si sono sviluppate tecniche ed anche centri di ricerca sempre più all’avanguardia che come obbiettivo avevano la produzione di musica da parte dell'elaboratore, fino ad arrivare al più recente IRCAM di Parigi che ha creato il software MAX/MSP.[9]
La Computer Music, cercando di scoprire sempre più interazioni tra computer e musica, rappresenta uno strumento con il quale l'uomo può creare un linguaggio musicale nuovo per esprimere il proprio pensiero. La ricerca in informatica musicale è quindi motivata anche da ragioni artistiche, oltre che da quelle scientifiche e tecnologiche, come la creazione di nuove forme compositive e di nuove metodologie di interpretazione musicale. È con questa ricerca che si sono sviluppate la musica algoritmica, frattale o genetica solo per fare qualche cenno. Alcuni campi di studio della Computer Music sono: la composizione assistita o automatica, la sintesi del suono, l'analisi e l'elaborazione del suono, l'analisi musicale e l'esecuzione di partiture.

L’obiettivo della mia tesi è proprio quello di far eseguire una partitura dal computer.
L' esecuzione di una partitura può essere considerata sotto due diversi punti di vista. In primo luogo come il risultato finale di una composizione automatica ed in secondo luogo come elemento puramente autonomo, dove il compositore immette nel computer direttamente le note.
Il primo passo è quello di rendere la partitura tradizionale “comprensibile” al computer e quindi riscriverla in un linguaggio informatico veloce e di facile implementazione. Quest'analisi è stata sviluppata dal Laboratorio di Informatica Musicale (LIM) presso l’Università degli Studi di Milano, creando il formato IEEE 1599 basato su un linguaggio di marcatura (Mark-up Language), con il quale è possibile descrivere l’informazione musicale in tutte le sue forme compresa la partitura, l’audio/video…
Con questo presupposto ho sviluppato un plugin che potesse decifrare questo formato così da rendere possibile la generazione di suoni che siano i corrispettivi della partitura iniziale.
Già nei primi personal computer erano state implementate le specifiche MIDI che hanno permesso, con il collegamento di sintetizzatori, di gestire applicazioni per l’esecuzione, la registrazione e l’editing. Il formato MIDI è un protocollo di comunicazione che inviando messaggi dà indicazioni sui comandi da eseguire, come per esempio l’esecuzione di una nota da parte di un generatore di suoni.
Il plugin che ho creato genera delle sequenze di messaggi MIDI inerenti agli eventi descritti dal formato IEEE 1599. Questi messaggi sono relativi alle note da eseguire, al tempo in BPM, all’esecuzione degli strumenti che suonano le varie voci della partitura.
Le sequenze di messaggi MIDI generate dal plugin vengono esportate e salvate su una memoria di massa scelta dal utente, dopo di che il file può essere eseguito da qualsiasi player audio. Oppure c'è la possibilità di eseguire operazioni di editing utilizzando sequencer o programmi di editing.

Ho voluto suddividere la tesi in tre capitoli. I primi due analizzano e descrivono la ricerca teorica che ho dovuto affrontare, per poi arrivare, a sviluppare il plugin software vero e proprio. In questa prima parte, teorica, sono illustrate le specifiche tecniche dei due formati che ho utilizzato: La codifica MIDI e il formato IEEE 1599.
Ho descritto la codifica MIDI nel contesto storico nella quale si è sviluppata, per capire perché si è arrivati ad avere necessità di un protocollo di comunicazione inizialmente tra sintetizzatori a tastiera e successivamente tra qualsiasi applicazione musicale che sia software o hardware.

Per il formato IEEE 1599 ho dato importanza alla struttura logica con il quale i vari livelli sono connessi, facendo comunque un'introduzione al linguaggio su cui si basa, cioè l'XML.

Nella terzo capitolo invece ho descritto i passi che mi hanno portato a scrivere fisicamente il programma software; quindi lo studio del linguaggio C#, con il quale è stato sviluppato; l'analisi dell'informazione musicale. Infine ho illustrato le funzionalità tecniche del plugin: l'analisi del formato IEEE 1599, la creazione dei messaggi MIDI, la mappatura del documento IEEE 1599, l'esportazione del file finale.

La funzione di questo elaborato di tesi è quella di poter creare i file MIDI delle opere codificate in IEEE 1599, così da ottenere un inventario sintetizzato. Con questo plugin sarà possibile anche variare le sonorità originali e sperimentare su opere compiute. In tal modo si permetterà ai fruitori di interagire con un repertorio musicale vario in maniera semplice e veloce.

- Capitolo1 -

Il MIDI

1.1 - La storia del MIDI

MIDI è l'acronimo di Musical Instrument Digital Interface, cioè interfaccia digitale per strumenti musicali. La sua nascita si deve principalmente a due fattori: il primo era l'esigenza di collegare facilmente tra loro diversi strumenti musicali elettronici, il secondo la crescente disponibilità di tecnologie digitali a basso costo. Nel 1981 Dave Smith e Chet Wodd della Sequential Circuit propongono una prima ipotesi del protocollo MIDI , le cui specifiche vengono definite nel 1982 in collaborazione con Roland, Korg e Yamaha, e pubblicate sotto il nome di “The Complete SCI MIDI”. Dopo alcuni mesi di sperimentazione del nuovo protocollo, il documento SCI viene rivisto e ribattezzato “MIDI Specification 1.0” in cui sono definiti i messaggi e le caratteristiche fisiche e logiche della loro trasmissione tuttora in vigore. L'americana Sequential, ai primi del 1983, mise in commercio il sintetizzatore Prophet 600 (fig.1), primo al mondo ad essere dotato di porte midi.

fig.1: Prophet 600

All’inizio la maggior parte dei produttori di strumenti furono restii nell’accettare il MIDI; alcuni di loro infatti, possedevano già proprie reti iter-strumentali.
Tra il1983 e il 1984 i sintetizzatori digitali MIDI ebbero una prima diffusione di massa con la commercializzazione di prodotti come lo Yamaha DX7 e il Korg Poly 800.
Dopo numerosi incontri informali tra produttori e utenti MIDI, nel 1984 sorgono contemporaneamente tre organizzazioni: l’internazionale IMA (International MIDI Association) per la publicazioni di specifiche MIDI e come punto di libero scambio delle informazioni sulle possibili applicazioni del MIDI;l’americana MMA (MIDI Manufacturers Association) e la giapponese JMSC (Japanese MIDI Standards Commitee) per l’applicazione delle specifiche MIDI.[4]

Dal marzo 1984 tutte le aziende che dotano i propri prodotti di porte midi sono obbligate a rispettare alcune caratteristiche base, che garantiscono la compatibilità fra gli strumenti delle diverse marche. Negli anni che seguno questo storico accordo, il MIDI non subisce sostanziali modifiche, se non alcuni aggiornamenti e miglioramenti.

1.2 - Le applicazioni MIDI

Il MIDI è l’unione di un linguaggio di comunicazione e una normativa di specifiche hardware che permette a strumenti musicali elettronici, controller, computer e altre apparecchiature che ne rispettino la codifica, di comunicare tra loro consentendo l’organizzazione e l’automazione di un sistema musicale MIDI anche molto complesso.
Il sistema musicale può essere il collegamento di più apparecchiature fisiche collegate tra loro oppure una rete ibrida costituita da hardware e software.
Il linguaggio MIDI crea dei messaggi digitali utilizzati per controllare generatori di suoni e altri dispositivi che servono alla configurazione automatica del sistema musicale. Inoltre il MIDI permette di memorizzare questi messaggi, mediante un apparecchio chiamato Sequencer, per essere successivamente editati, modificati e rispediti alle apparecchiature presenti sulla rete.
I settori di applicazione del MIDI sono molteplici e vanno dal utilizzo non professionale, come ad esempio piccoli home studio, a quello professionale in studi di registrazione, applicazioni multimediali, controllo luci-effetti, post produzione e

ovviamente in situazioni live. Tutti questi settori si avvalgono del MIDI per la facilità di utilizzo, la flessibilità e rapidità di programmazione.

1.3 - Il linguaggio MIDI dalla sintassi ai canali

Arrivati a questo punto della trattazione si può dire che il MIDI è un protocollo di comunicazione dati. Quando due dispositivi si scambiano informazioni per mezzo del codice MIDI , ciò che in effetti si trasmette è una serie di bit, è necessario ora capire come una serie di numeri, possa comunicare messaggi di senso compiuto.

La prima regola del protocollo MIDI sancisce la differenziazione tra **Status Byte** (byte di stato) e **Data Byte** (byte di dati). Lo Status Byte fornisce la chiave di lettura che serve allo strumento ricevente per una corretta interpretazione dei byte di dati, cioè servono per definire in modo univoco un comando. Il bit più significativo dello status byte è uguale ad 1 e quindi uno status byte può assumere un valore compreso tra 128 e 255. Gli Status Byte trasmettono quindi il tipo di informazione (suona una nota, alza il volume, ecc).
I Data Byte invece servono per inviare gli eventuali parametri necessari per un corretto funzionamento dello Status Byte. Il bit più significativo del Data Byte è uguale a 0 e quindi può assumere un valore decimale compreso tra 0 e 127.

I **canali MIDI** sono una proprietà intrinseca dei messaggi MIDI. Infatti ogni messaggio può essere inviato e ricevuto su 16 diversi canali (logici). Ognuno di questi canali di comunicazione può trasmettere una determinata informazione che può essere ricevuta da un dispositivo sintonizzato su quel canale. Questi canali sono indipendenti tra di loro e possono inviare messaggi in parallelo.[1]

1.4 - I Tipi di messaggi MIDI [2]

I messaggi MIDI si suddividono in due categorie differenti:

- I Channel Message
- I System Message

1.4.1 - Channel Message

I messaggi di canale sono quelli che possono essere indirizzati ad uno qualsiasi dei sedici canali MIDI, essi si dividono in:

- *Channel voice message*
- *Channel mode message*

Lo Status Byte di un generico messaggio di canale è sempre nel formato:
< 1 xxx nnnn >.
Dal momento che il byte più significativo (MSB) dello Status Byte (il primo a sinistra), è sempre uguale a "1", restano solo tre bit (xxx) per definire i tipi di comandi da trasmettere.
Quando l' apparecchiatura ricevente riconosce il messaggio di canale, prende in esame i successivi quattro bit (nnnn), che sono utilizzati per specificare su quale dei sedici canali disponibili, il messaggio è stato trasmesso.

***Channel Voice message**:*

I principali messaggi MIDI sono i Channel Voice Message, che comprendono tutti i messaggi che vengono generati da eventi di esecuzione.
Tra questi per esempio: Quale tasto è premuto, quale interruttore o pulsante è stato selezionato, quale controller si sta utilizzando (pitch band, pedale, breath ecc.); oppure, la dinamica della tastiera intesa come velocità di abbassamento dei tasti (velocity), o informazioni relative ai cambi di programma (program change).

Ora vorrei soffermarmi su quei messaggi di canale che poi saranno utilizzati nell'elaborato, che sono i messaggi di NOTE ON, NOTE OFF e i PROGRAM

CHANGE.

NOTE ON e NOTE OFF:

Quando da una tastiera si preme un tasto, essa manda via MIDI un messaggio detto Note on con le specifiche di che nota si tratta, con quale intensità si sta suonando (se la tastiera è sensibile alla dinamica) e in quale ottava.
Un esempio di messaggio di Note on è il seguente:

Status Byte: 1001cccc (Note on), cccc indica il canale di trasmissione. Data Byte 1: 0nnnnnnn (Key number) gli n indicano il numero della nota che deve essere suonata. Data Byte 2: 0vvvvvvv (Velocity number), le v indicano la velocità con cui si preme il tasto.

N.B. Le note sono rappresentate da numeri: 0 per la nota C, 1 per C#, 2 per D ecc.

Quando il tasto viene rilasciato la tastiera produce un messaggio di Note off che indica quale tasto è stato rilasciato. Il messaggio è simile a quello di Note on:

Status Byte: 1000cccc (Note off), cccc indica il canale di trasmissione. Data Byte 1: 0nnnnnnn (Key number) gli n indicano il numero della nota che deve essere rilasciata. Data Byte 2: 0vvvvvvv (Release velocity), le v indicano la velocità di rilascio del tasto.

N.B. In genere il messaggio Note off viene sostituito con un messaggio Note on con velocità uguale a 0.

PROGRAM CHANGE:
Questo messaggio è utilizzato per cambiare il programma in memoria, e può essere riferito ad una patch, un preset o una performance sul ricevente. Quando sul dispositivo mittente si seleziona un programma accadono due cose: primo, si prepara a suonare con il timbro prescelto, secondo, trasmette questa variazione su un canale MIDI.

I comandi di Program Change ,possono essere inviati tramite comandi manuali sullo strumento (tastiera) master, oppure memorizzati su sequenze al punto esatto in cui tali programmi dovranno essere cambiati. Per ogni "programma" , si intende una generica configurazione funzionale memorizzata e richiamabile da una qualsiasi apparecchiatura MIDI. Perciò può essere inteso come il preset di un sintetizzatore, il timbro di un campionatore, un particolare effetto di un DSP, i livelli di un mixer, ecc.
Non vengono trasmessi i parametri relativi a un certo programma, ma solo il numero che lo contraddistingue. I preset timbrici selezionabili via MIDI, possono assumere i valori da 0 a 127.

Status Byte: 1100cccc (Program change), cccc indica il canale di trasmissione.
Data Byte 1: 0nnnnnnn (Preset number) gli n indicano il numero della patch da selezionare.
Data Byte 2: non c è.

Gli altri Channel Voice Message sono:
- **POLYPHONIC KEY PRESSURE** (prevede che la pressione esercitata sui tasti venga inviata distintamente per ogni tasto premuto).
- **CONTROL CHANGE** (utilizzato per indicare i cambiamenti di posizione di un qualunque Controller).
- **CHANNEL PRESSURE** (trasmette il valore della pressione esercitata su di un tasto dopo che questo è stato abbassato).
- **PITCH BEND CHANGE** (è utilizzato per indicare, la posizione del Pitch Bend, che sulle tastiere elettroniche può essere variata muovendo una rotella.)

***Channel Mode Message**:*

I messaggi di modo non sono altro che una categoria all'interno dei messaggi di Control Change. Dal punto di vista della sintassi sono assolutamente identici, ma da quello del contenuto risultano differenti.

Infatti, questi riguardano principalmente il controllo generale di un apparecchiatura a non un singolo parametro. Esistono tre stati fondamentali:

- **OMNI**: sta a significare che lo strumento risponde a tutti i messaggi ricevuti contemporaneamente su tutti e sedici i canali. Lo stato può essere ON o OFF.
- **POLI**: significa che lo strumento viene suonato in maniera polifonica, ossia utilizza più di una voce.
- **MONO**: significa che lo strumento viene suonato in maniera monofonica (utilizza una sola voce).

I modi disponibili sono ottenuti combinando gli stati appena descritti :

modo 1: Omni On / Poly

modo 2: Omni Off / Poly

modo 3: Omni On / mono

modo 4: Omni Off / mono

1.4.2 - System Message

Questi messaggi, a differenza dei Channel Message non contengono informazioni di canale, bensì di sistema, quindi possono essere ricevuti da qualsiasi apparecchiatura MIDI. Sono utilizzati soprattutto per gestire la sincronizzazione tra gli strumenti.

Essi si dividono ulteriormente in:

- *System common message*
- *System Real time message*
- *System exclusive message*

Lo Status Byte di un messaggio di sistema si presenta nella forma:

< 1111 xxxx >;

i quattro bit più significativi sono uguali a uno, mentre gli altri quattro bit sono utilizzati per definire uno dei sedici possibili messaggi.

System common message:

E' un piccolo gruppo di messaggi che viene utilizzato per espletare particolari funzioni MIDI quando nel sistema sono presenti apparecchiature quali: Sequencer, Drum Machine, Master Keyboard.

I messaggi più comuni si presentano nella forma <Status Byte>, <Data Byte 1> e <Data Byte 2>, anche se il secondo Data Byte non è sempre necessario.

I messaggi sono:

-**MIDI TIME CODE QUARTER FRAME** (traduce in codice MIDI il protocollo di sincronizzazione audio/video universalmente riconosciuto, cioè l'SMTPE).

- **SONG POSITION** (permette di assegnare un indirizzo assoluto a ogni tempo della sequenza consentendo la conversione del codice SMTPE i codice equivalente al SongPositionPointer, premettendo di sincronizzare registratori multi traccia e sequenze senza dover ripartire dall'inizio del nastro, ma da una posizione qualsiasi).

- **SONG SELECT** (permette la selezione di una song in una Drum Machine o sequenze. Esso non contiene informazioni che riguardi l'esecuzione e può assumere valori da 0 a 127).

- **TUNE REQUEST** (impiegato per accordare i vecchi sintetizzatori MIDI analogici, non è più utilizzato nei moderni modelli digitali).

***System Real Time Message**:*

I messaggi in tempo reale sono costituiti da un solo byte e possono essere inviati in qualsiasi momento, anche inframmezzandosi ai byte di altri messaggi. In questo caso le operazioni di trasmissione di altri messaggi vengono temporaneamente sospese.

Questi messaggi sono:

- **TIMING CLOCK** (si tratta del segnale di sincronismo del MIDI scambiato tra sequenze).

- **START** (impone a tutte le apparecchiature collegate di portarsi alla prima battuta della song).

- **CONTINUE** (impone a tutti i dispositivi collegati di partire in play/rec dal primo tempo successivo all'ultimo STOP effettuato).

- **STOP** (impone a tutte le apparecchiature collegate di fermare la riproduzione).

- **ACTIVE SENSING** (inviato quando tra due apparecchiature collegate manca il flusso di dati MIDI per una durata di 300 millisecondi, è trasmesso per mantenere attivo il dialogo tra i dispositivi).
- **SYSTEM RESET** (consente di re-inizializzare tutti i dispositivi del sistema riportandoli alle condizioni di default).

System Exclusive Message:

Questi messaggi servono per accedere alle funzioni del dispositivo MIDI e per poterlo programmare in maniera altrimenti impossibile. Il protocollo MIDI infatti non prevede per esempio la modifica della curva di inviluppo di un timbro musicale, ma intervenendo con un messaggio di sistema esclusivo direttamente al dispositivo è possibile effettuare questa modifica. Ci sono molti parametri che si possono modificare con messaggi di sistema esclusivo e per sapere quali sono bisogna vedere il manuale dello strumento MIDI.
Poiché ci sono diverse apparecchiature di marche differenti si è pensato di introdurre all'interno di un messaggio di sistema esclusivo un ID, <Manufacture ID>. Cioè un codice identificativo che indica alle apparecchiature aventi lo stesso ID che il messaggio di sistema esclusivo è per loro e tutte le altre apparecchiature collegate, ma con ID differente, ignoreranno quel messaggio.
Questi messaggi si presentano nella seguente forma generale:
<Status byte> <Manufacture ID> <Data byte 1> … <EOX>
I messaggi esclusivi sono:
- **SYSTEM EXCLUSIVE** (indica che sta per avere inizio una trasmissione di sistema esclusivo).
- **END OF EXCLUSIVE** (è utilizzato per avvisare la fine della trasmissione di un messaggio esclusivo).

1.5 - La temporizzazione nel MIDI

La concezione di sincronismo in un esecuzione di un brano da parte di strumentisti è norma comune per la riuscita realizzazione, questa ideazione è possibile grazie a un riferimento comune che può essere il clic di un metronomo o il tempo dato dal direttore d'orchestra o ancora, in complessi pop, il tempo del batterista.
In questo senso anche la macchina che esegue un brano ha bisogno di un rapporto preciso. In funzione del tipo di applicazione, il segnale di riferimento può derivare dal rispetto di un tempo di esecuzione (tempo relativo) o da un vero e proprio orologio (tempo assoluto).
La distinzione tra questi due metodi di distinzione sta alla base dei concetti di sincronizzazione e di time code. Per esempio, per una durata di tempo stabilita (un minuto), il tempo può essere calcolato da un orologio e in funzione della durata della nota da un quarto: 120 note della durata da un quarto per un tempo di esecuzione di 120 bpm, oppure, 60 note della durata di un quarto per un tempo di 60 bpm; il tempo trascorso è sempre un minuto. In un sistema di temporizzazione relativo ci si riferisce a una unità variabile che dipende dal tempo di esecuzione, in uno assoluto, il riferimento è un unità definita misurata in ore, minuti, secondi, millisecondi, ecc.
Il time code altro non è che la conversione in digitale di un segnale di sincronismo analogico.[3]

Ci sono due standard di time code relativo: PPQN o impulsi per durata della nota di un quarto e MIDI Timing Clock.

1.6 - MIDI Sequencing

La caratteristica generale del MIDI è quella di poter eseguire i comandi impartiti in tempo reale oppure d'immagazzinare i dati per futuri utilizzi in tempo differito.
Per questo secondo utilizzo è possibile memorizzare liste di messaggi che poi verranno inviati ai circuiti di generazione del suono. I programmi che svolgono questa funzione

di memorizzazione sono i **Sequencer,** tali applicazioni sono in grado di registrare più esecuzioni musicali e distribuirle su più tracce.

I compiti generali di un Sequencer possono essere riassunti nei seguenti punti:
- Rispondere a comandi di Record , Play, Stop, ecc.;
- Ricevere i messaggi dalla porta fisica (MIDI IN), o da altre applicazioni software;
- Interpretare i messaggi che arrivano e memorizzarli su richiesta dell'utente;
- Temporizzare ogni evento registrato;
- Ritrasmettere quando è richiesto i messaggi ricevuti, della porta fisica (MIDI OUT) o ad altre applicazioni, nello stesso ordine in cui erano arrivati.

Il grande vantaggio di un sequenze MIDI sta nel fatto che non viene registrato un suono, ma l'istruzione che lo genera; questo permette perciò di riascoltare il brano registrato utilizzando timbri diversi, consentendo di sperimentare orchestrazioni differenti, per poi scegliere la più adatta. I cambi di programma possono essere memorizzati nella traccia sotto forma di dati MIDI, automatizzando un processo prima manuale.

1.7 - Le tracce MIDI

Solitamente i Sequencer MIDI sono strutturati come i registratori a nastro multi-traccia, con tracce parallele e controlli di registrazione, riproduzione e avanzamento. Le tracce non sono gestite dal computer in modo parallelo, ma sono distribuite in varie zone della memoria.

Quando si registra una nuova traccia, il programma riserva una particolare zona di memoria identificata da una specifica etichetta che contiene il numero a cui la traccia è associata. In fase di riproduzione, il Sequencer identifica le etichette di tutte le tracce che sono state registrate e inizia l'esecuzione operando una lettura dei dati con le varie tracce in modo sincrono. Ogni traccia può essere assegnata a una canale MIDI specifico (da 1 a 16) per permettere l'esecuzione con strumenti diversi.

- Capitolo 2 -

Il formato IEEE 1599

2.1 - Il linguaggio XML[8]

XML è l'acronimo di eXtensible Markup Language, e costituisce uno standard approvato dal W3C.
L' XML è un linguaggio di marcatura (mark-up language), cioè una codifica che fa uso di un insieme di marcatori (mark-up) convenzionali, ossia aderenti a regole definite e standardizzate.
Un linguaggio di marcatura deve specificare:
- *qual è il significato di ogni mark-up*
- *come si distinguono i mark-up dal testo*
- *quali mark-up sono consentiti*
- *quali mark-up sono richiesti*

Ad esempio, SGML (Standard Generalized Markup Language) fornisce gli strumenti per rappresentare i primi tre punti.
SGML é uno standard internazionale per la descrizione di testi elettronici di tipo mark-up. Precisamente, SGML non è un semplice linguaggio bensì un meta-linguaggio, ovvero un linguaggio per la descrizione formale di linguaggi.
Dal punto di vista di SGML, un documento è una struttura gerarchica di elementi annidati; tale linguaggio non ha i mezzi per specificare aspetti di presentazione di tali elementi, né è in grado di fornire informazioni sul ruolo di alcun elemento. Queste informazioni sono sottintese dal creatore dell'applicazione SGML e sono solitamente auto-esplicative, o fornite a commento, oppure presenti nella documentazione che accompagna la specifica formale. In SGML, l'unica informazione che può sulla natura di un elemento è in quali contesti e livelli di gerarchia esso possa o debba apparire.
Tutti i documenti che presentano la medesima gerarchia di elementi sono considerati

dello stesso tipo. La struttura di un particolare tipo di documenti avviene tramite la cosiddetta definizione del tipo di documento (DTD, Document Type Definition).

XML è un sottoinsieme di SGML, e ne rappresenta una semplificazione in quanto:

- trascura molte opzioni sintattiche e varianti
- trascura alcune caratteristiche del DTD

I suoi principali vantaggi sono:

- *Interscambiabilità in rete*: possibilità di comunicare e condividere agevolmente l'informazione codificata, in ambienti distribuiti e su Internet in particolare.
- *Indipendenza dalla piattaforma*: è una caratteristica direttamente collegata al punto precedente.
- *Struttura gerarchica*: particolare disposizione a rappresentare informazione strutturata secondo schemi gerarchici.
- *Intelligibilità*: facilità di lettura dell'informazione codificata.
- *Estensibilità*: predisposizione ad aggiunte, integrazioni e modifiche, da operare qualora il linguaggio si mostrasse carente o inadeguato agli scopi.
- Disponibilità *di tool per l'implementazione del formato e per la validazione dei file prodotti*.

Ciascuno di questi punti trova applicazione nel campo della musica, in cui l'informazione è fortemente strutturata e gerarchizzata.

Gli **elementi** rappresentano i mattoni costitutivi di XML. Essi consentono di attribuire un preciso significato ad una parte di documento.
Ciascun tipo di elemento è rappresentato da un contrassegno (mark-up), detto anche etichetta (tag).
La sintassi dei tag è <element_name>…</element_name>, ove element_name rappresenta una stringa alfanumerica arbitraria. Dunque il singolo elemento viene delimitato in realtà da una coppia di etichette, dette rispettivamente etichetta di apertura (start tag) e di chiusura (end tag).Il carattere "/" deve essere anteposto all'identificativo nel tag di chiusura. All'interno di tale coppia si trova il contenuto del tag
Tra i diversi elementi, uno ricopre una particolare importanza. Si tratta dell'elemento ***document***, detto anche **etichetta radice** (*root tag*): è l'elemento più esterno, di

conseguenza è quello che racchiude tutti gli altri elementi del documento. Esso deve esistere sempre.

Gli attributi vengono utilizzati per aggiungere informazione ad un elemento.
Sono sempre associati allo start tag:

```
<element_name attribute_1="value_1" … attribute_N="value_N" >
…
</element_name>
```

Un elemento può avere un numero qualsiasi di attributi distinti.
Ogni attributo presenta un nome (che precede il segno "=") e un valore (scritto tra virgolette dopo il segno "="), ed è separato dal nome dell'elemento e dai successivi e/o precedenti attributi da uno spazio. Gli attributi sono posti all'interno delle parentesi angolari dello *start tag*.

2.1.1 - Il DTD

L'idea base consiste nell'associare un tipo al documento. Volendo cercare un'analogia con la programmazione, pensiamo a classi e oggetti. Un tipo di documento rappresenta una classe di documenti con una struttura ed una semantica simile.
Il DTD fornisce un significato standard per descrivere in modo dichiarativo la struttura di un tipo di documento. Ciò significa descrivere: quali (sub-)elementi può contenere un elemento, se quest'ultimo possa contenere del testo o meno, quali attributi appartengano all'elemento, tipizzazione e defaultizzazione degli attributi.
Un DTD é logicamente composto da 2 parti: Element Type Definition, Attribute List Declaration.

L'Element Type Definition specifica la struttura del documento, i contenuti consentiti (content model) e gli attributi consentiti (dal significato delle dichiarazioni delle liste di attributi).
Esempi di definizioni nei DTD possono essere:

```
<!ELEMENT A (B*, C, D?)>
",” = A può contenere gli elementi B, C e D, in questo ordine e con la corrispondente cardinalità; significa sequence of
```

cardinalità di default (non specificata) = 1 e solo 1

cardinalità “?” = 0 o 1

cardinalità “*” = 0 o più

cardinalità “+” = 1 o più

<!ELEMENT A (B | C+)>

“|” = A contiene o l’elemento B o l’elemento C; significa choice of

<!ELEMENT A (#PCDATA)>

elemento di testo

<!ELEMENT A EMPTY>

elemento vuoto

<!ELEMENT A (#PCDATA| B | C)*>

elemento misto

L’Attribute List Declaration è la lista degli attributi permessi per ogni elemento. Ogni attributo è specificato da: name, type, e altre informazioni.

<!ATTLIST ELEM attrib1 CDATA #IMPLIED>

attrib1 è di tipo testuale e non è richiesto

<!ATTLIST ELEM attrib1 CDATA #IMPLIED attrib2 CDATA #REQUIRED>

attrib1 e attrib2 sono di tipo testuale; il primo non è richiesto, il secondo sì

<!ATTLIST ELEM attrib1 CDATA #IMPLIED “aaa”>

attrib1 è di tipo testuale, non è richiesto e ha valore di default “aaa”

<!ATTLIST ELEM attrib1 CDATA #REQUIRED “aaa”>

attrib1 è di tipo testuale, è richiesto e ha valore di default “aaa”

<!ATTLIST ELEM attrib1 CDATA #FIXED “aaa”>

attrib1 è di tipo testuale e ha valore costante “aaa”

<!ATTLIST ELEM attrib1 (aaa|bbb) #IMPLIED “aaa”>

attrib1 è di tipo enumerato (valori elencati tra parentesi) e ha valore di default “aaa”

<!ATTLIST ELEM id ID #REQUIRED>

attrib1 è di tipo ID (assegnazione di un valore univoco)

<!ATTLIST ELEM ref IDREF #IMPLIED>

attrib1 deve essere un riferimento a un ID

2.2 - Introduzione al formato IEEE 1599

IEEE 1599 è un nuovo formato basato sul XML per descrivere le informazioni musicali eterogenee nella loro interezza. In un singolo file i contenuti relativi a simboli musicali, partiture, tracce audio, computer performance, cataloghi di metadati, testi e grafici, di una opera musicale sono collegate e reciprocamente sincronizzate nello stesso contesto (framework). I contenuti sono organizzati in una struttura a più livelli che supportano differenti codifiche.

La musica può essere descritta secondo molti punti di vista. Ad esempio, un esecutore di solito è interessato a una partitura, un musicologo a dei documenti autografi, un dee-jay alle registrazioni audio, e un ascoltatore alle registrazioni di spettacoli. L'informazione musicale è intrinsecamente eterogenea, poiché si tratta di una serie di diversi mezzi di comunicazione, nonché una serie di descrizioni diverse. Tutti i documenti danno un contributo alla descrizione generale del brano musicale, ed è di notevole interesse, da un punto di vista culturale, scientifico e commerciale,

Per fornire l'acceso a tali informazioni, come dimostra la crescente quantità di progetti di digitalizzazione e di data base multimediali.

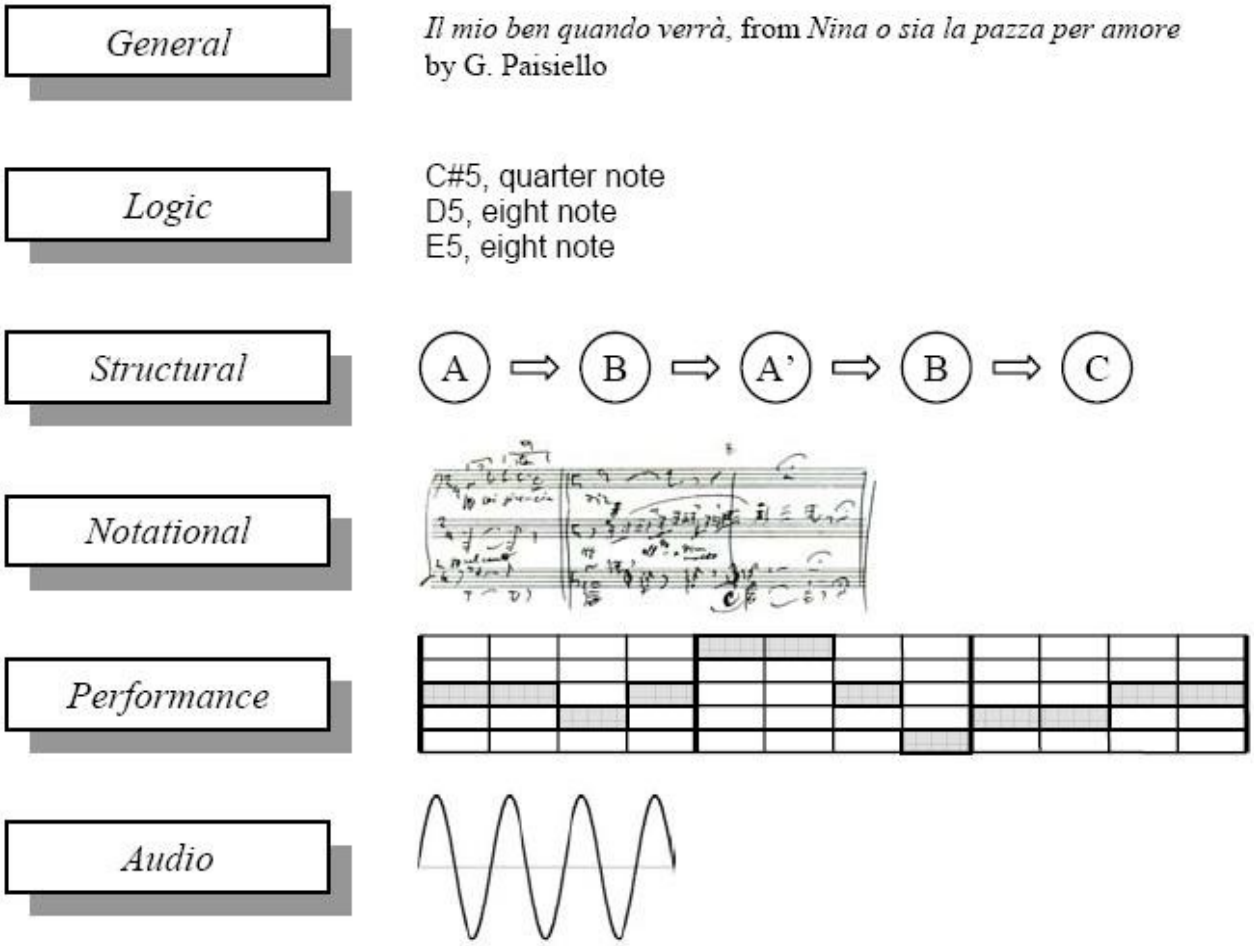
2.3 - La struttura a livelli del formato IEEE 1599

L’informazione musicale è senza dubbio ricca e complessa. Gli autori del formato IEEE 1599 hanno individuato sei diversi strati (layers) per descriverla nella sua interezza:

1. *general* (Catalogo delle informazioni dell'opera musicale, metadati del brano)
2. *logic* (Descrizione logica dei simboli della partitura)
3. *structural* (Identificazione degli oggetti musicali e delle loro relazioni)
4. *notational* (Rappresentazione grafica delle partiture)
5. *performance* (Esecuzioni da parte del computer)
6. *audio* (Registrazioni digitali del brano del opera musicale)

Ogni livello rappresenta un grado di astrazione diverso per l’informazione musicale.

Il grafico seguente mostra gli scopi dei vari livelli:



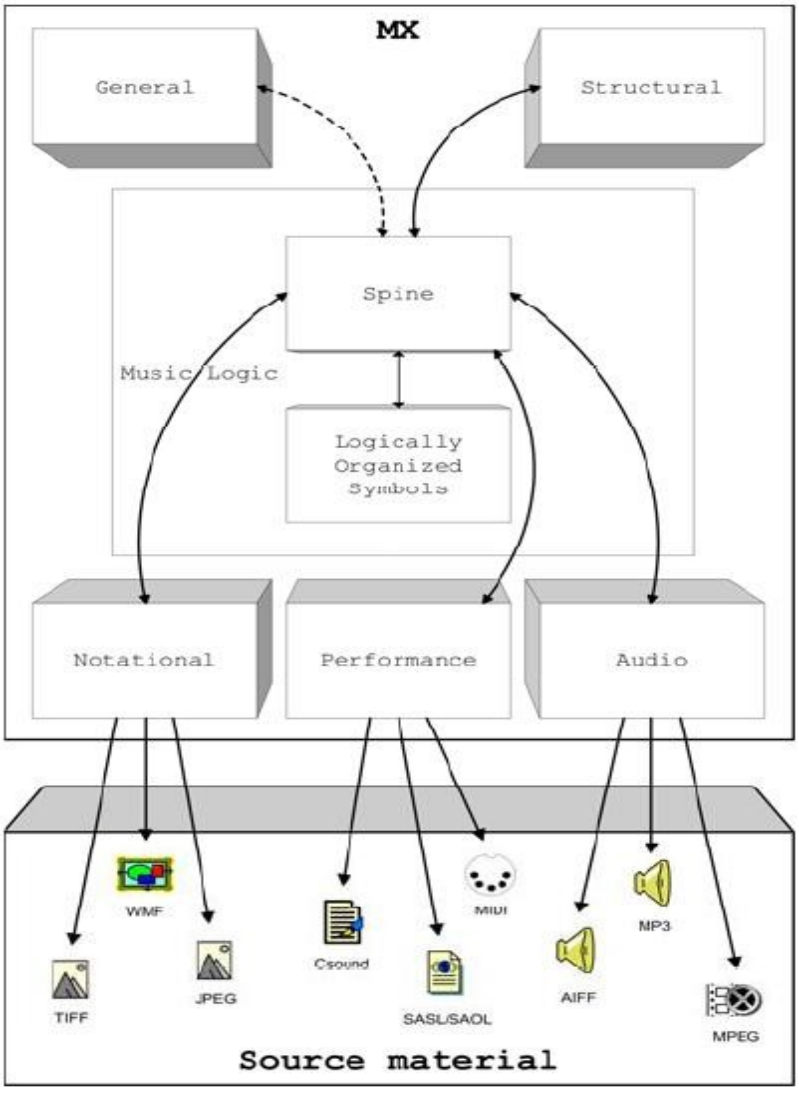
Un generico documento IEEE 1599 ha una struttura XML simile alla figura che segue:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ieee1599 SYSTEM
       "http://www.mx.dico.unimi.it/ieee1599.dtd">
<ieee1599>
       <general>...</general>
       <logic>...</logic>
       <structural>...</structural>
       <notational>...</notational>
       <performance>...</performance>
       <audio>...</audio>
</ieee1599>
```

2.4 - Relazione spazio-temporale nel formato IEEE 1599

Esistono numerosissimi formati per la rappresentazione dei diversi aspetti musicali (ad esempio MP3 e WAV per l'*audio*, MIDI e MPEG per la performance, TIFF e JPG per la notation e così via); però manca una struttura che rappresenti la relazione spazio-temporale implicita nella musica.

IEEE 1599 individua tale struttura nello *spine*. Lo *spine* può essere visto come una funzione di mappatura bidimensionale tra i domini dello spazio e del tempo, e rappresenta in pratica un collante tra i diversi strati. Esso si compone di eventi, ciascuno dei quali presenta un riferimento nel dominio dello spazio e del tempo. Grazie all'uso dello *spine*, i differenti formati di file possono essere messi in relazione per ottenere una descrizione completa dell'informazione musicale.



In questo approccio, descrizioni eterogenee dello stesso brano musicale non sono semplicemente collegati tra loro, ma un ulteriore livello di dettaglio è prevista: ove possibile, le informazioni musicali sono connesse ad eventi musicali singoli. Il concetto di evento musicale è lasciato intenzionalmente vago, dato che il formato è flessibile e adatto a molti scopi. Un evento musicale può essere definito come una astrazione della partitura, o di qualcosa che è considerato importante dall'autore della codifica. Ad esempio note e pause della partitura possono essere considerati eventi musicali.

2.5 - Il Layer General

Il layer *general* prende in considerazione il brano musicale nella sua interezza, dandone un inquadramento generale e raggruppando le informazioni sulle relative istanze.

```
<!ELEMENT general (description, casting?, related_files?, analog_media?, notes?, rights?)>
```

La parte più importante ai fini dell'identificazione di un brano è quella relativa all'elemento *description.*

```
<!ELEMENT description (work_title?, work_number?, movement_title, movement_number?, genre?, author+)>
```

2.6 - Il Layer Logic

Il layer *logic* presenta tre sottolivelli (elementi figli)

1. *spine* contiene la funzione di mappatura spazio-temporale
2. *los* descrive gli oggetti in partitura

3. *layout* si occupa di rappresentare il layout del brano

Chiaramente, *spine* e *los* sono fondamentali per un documento IEEE 1599 la loro presenza è dunque richiesta.

In particolare, *los* contiene le informazioni ricavabili dalla partitura intesa come insieme di oggetti musicali (note, pause, segni di articolazione,...), mentre ignora gli aspetti di *layout* (margini, impaginazione,...) cui è dedicato un elemento apposito.

Lo *spine* è fondamentale data la natura multi-livello dell'IEEE 1599: senza lo *spine*, che funge da collante tra i vari livelli di rappresentazione, il formato IEEE 1599 non avrebbe significato.

2.7 - Lo spine

L'elemento *spine* rappresenta la struttura logica che implementa l'integrazione dei layer *notational*, *performance* e *audio* con il music *logic*. Il suo obiettivo è costruire una struttura astratta cui fanno riferimento tutti gli strati che descrivono le proprietà del materiale originario.

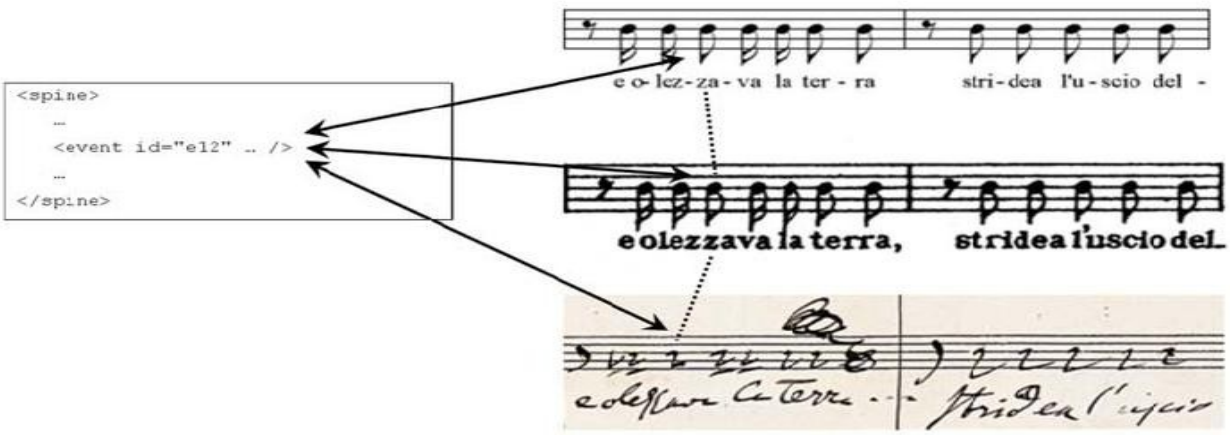
Lo *spine* praticamente è una lista di eventi, dove la definizione e la granularità degli eventi può essere scelta dall'autore della codifica. Dal momento che si limita a elencare gli eventi e di fornire loro una identificazione univoca, la mera presenza di un evento nello *spine* non ha alcun significato semantico. Di conseguenza, ciò che è nello *spine* deve avere una controparte in alcuni livelli, altrimenti l'evento non sarà definito e la sua presenza nella lista (e nel file XML) sarebbe assolutamente inutile.

La figura che segue illustra un breve esempio di *spine*.

```
<ieee1599>
  <logic>
    <spine>
      <event id="e1" timing="0" hpos="0"/>
      <event id="e2" timing="2" hpos="2"/>
      <event id="e3" timing="2" hpos="2"/>
      <event id="e4" timing="1" hpos="1"/>
      <event id="e5" timing="1" hpos="1"/>
      ...
    </spine>
    <los>...</los>
  </logic>
</ieee1599>
```

L'attributo *timing* costituisce una temporizzazione virtuale in VTU (virtual timing units). Esistono delle norme per scegliere l'unità di misura, che non è fissata: i VTU non sono millisecondi, decimi o quarti d'ora, ma l'utente sceglie che significato temporale attribuire ai VTU di volta in volta. Nella scelta del VTU di riferimento è consigliabile utilizzare valori grandi, divisibili per molti divisori e potenze di due per i valori più lunghi. Ad esempio, in un corale scritto sostanzialmente a quarti si può assegnare il valore 1024 al quarto. Le potenze di due consentono di rappresentare molti valori con numeri interi (1024 = semiminima, 512 = croma, 256 = semicroma,…); un valore base divisibile per 3, per 5, per 7 consente di rappresentare con numeri interi le note che costituiscono terzine, quintine, settimine. In ogni caso, si tratta di una scelta del tutto arbitraria. La valutazione si deve basare anche su considerazioni di granularità degli eventi da descrivere. Se abbiamo a che fare con un corale di Bach in cui si usano solo metà, quarti e ottavi, si può effettuare l'assegnamento:
croma = 1, semiminima = 2, minima = 4.
La granularità risultante è molto grezza: nessun valore intero potrebbe rappresentare ad esempio un ottavo con il punto, in quanto la sua durata sarebbe di 1,5. Però in certi casi semplici potrebbe essere adeguata anche una scelta così rinunciataria.
Un discorso analogo a quello fatto per il *timing* vale per l'*hpos*, che però tiene conto della spazializzazione orizzontale (sempre virtuale) anziché della temporizzazione. La partitura va concepita come nella rappresentazione Score view di Finale, ossia tutta di seguito e senza ritorni a capo.
Si noti che il valore di *timing* e *hpos* non dipende dalla figura musicale stessa, ma da quella precedente. Si consideri la sequenza A-B-C-D, con A, C e D ottavi e B quarto, attribuendo agli ottavi 200 VTU e ai quarti 400 VTU: $\text{timing}_A = 0$, anche se la nota A è un ottavo; $\text{timing}_B = 200$, anche se la nota B è un quarto (infatti, la nota A che precede B è di un ottavo); $\text{timing}_C = 400$, dato che B è un quarto; $\text{timing}_D = 200$, dato che C è un ottavo. Stesso discorso vale per gli hpos.

Gli eventi elencati nello *spine* possono corrispondere a uno o a molti casi in altri livelli. Per esempio la figura che segue illustra un evento singolo che collega tre versioni differenti di partitura, corrispondenti a tre file diversi.



2.8 - Logical Organized Symbols

In questo sub-layer si trovano le informazioni simboliche di partitura. In altre parole, il contenuto informativo simbolico del brano (note, pause, segni di articolazione, dinamiche…) è interamente racchiuso nell'elemento *los*, che sta appunto per Logical Organized Symbols.

Questo Sub-layer è importantissimo ai fini della mia tesi perchè è il luogo in cui gli eventi della partitura sono codificati nel formato IEEE 1599 ed e quindi il livello nel quale il mio plug-in deve ricercare gli eventi da inserire nei messaggi MIDI, sarà quindi tema di trattazione, approfondito, nel paragrafo 1.2.14.

2.9 - Layout

L'informazione musicale codificata nel modo finora descritto consente grande flessibilità anche nel rendering visuale. Infatti, è facile definire elementi e proprietà che, sulla base dell'informazione contenuta nello *spine* e nel *logic*, organizzino i simboli

musicali su diversi media, a partire dalla carta stampata nei vari formati alle differenti risoluzioni degli schermi dei PC. Questo perché lo *spine* funge da ponte consentendo riferimenti indiretti dal *layout* (o meglio, dalle diverse istanze in *layout*) ai simboli in partitura. Simboli e segni non convenzionali possono essere modellati facendo uso dell'SVG (*Support Vector Graphics*).

2.10 - Il Layer Structural

Il layer *structural* si occupa di descrivere le relazioni interne al brano in esame: temi musicali, soggetti, sequenze o segmenti che si ripetono, o che presentano un particolare interesse.
Le informazioni qui racchiuse sono il frutto arbitrario di analisi di carattere musicologico, svolte manualmente o automaticamente.
Nel layer *structural* troviamo in pratica il risultato di una segmentazione, o la rappresentazione dell'informazione musicale basata su Reti di Petri. In futuro il formato potrebbe essere ampliato per supportare altri formalismi.

2.11 - Il layer Notational

Il layer *notational* si riferisce alle istanze "visive" di un pezzo musicale.
Sono state individuate due modalità di scrittura e lettura della musica: quella notazionale (*notational*) e quella grafica (graphical). Un'istanza notazionale è spesso in formato binario, come in NIFF o in ENGIMA, e rappresenta informazione simbolica. Altre istanze di questo tipo in formati XML alternativi sono: MusicXML di M. Good, Music Encoding Initiative (MEI) di P. Roland e Music Markup Language di J. Steyn. Un'istanza grafica, invece, contiene immagini che rappresentano la partitura. Anch'essa solitamente si presenta in formato binario (ad esempio, file JPEG, TIFF o PDF), ma si

può anche trattare di immagini vettoriali codificate in SVG (e quindi in formato testuale senza riferimenti espliciti alla musica). L'informazione di questo layer si lega alla parte spaziale dello *spine*, il che ne consente la corretta localizzazione.

2.12 - Il Layer Performance

Il layer *performance* si colloca tra gli strati *notational* e *audio*. I formati di file che sono supportati da questo layer codificano i parametri delle note da eseguire (così come avviene nel precedente layer *notational*) e i parametri dei suoni da creare (così come avviene nel successivo layer *audio*), ma sempre al fine di una produzione musicale sintetica, ossia da parte dell'elaboratore.
performance abbraccia i formati MIDI, CSound, SASL/SAOL (MPEG4) e potrebbe essere ulteriormente esteso.
I descrittori propri di questo layer non consentono il riferimento ai singoli eventi, ma solo a loro blocchi. In altre parole, un'istanza può essere relativa all'intero pezzo, a un suo intero segmento o a un solo strumento, ma non può avere granularità più fine.
Vediamo ora come viene definito il layer *performance* nel formato IEEE 1599:

```
<!ELEMENT performance (performance_instance+)>
```

Il layer in questione dunque si compone di una o più rappresentazioni in forma subsimbolica, dette performance_instance, che ora andiamo ad analizzare.

```
<!ELEMENT performance_instance (desc?, (MIDI | CSOUND | MPEG4), rights)>
```

La trattazione del MIDI merita un discorso a parte, dato che l'elemento è più ricco di informazioni rispetto al CSound e all'MPEG.

```
<!ELEMENT MIDI (MIDI_part_ref+)>
<!ATTLIST MIDI
        format (0 | 1 | 2) #REQUIRED
>
<!ELEMENT MIDI_part_ref EMPTY>
```

```
<!-- Gli attributi di tracce e canali servono per l'identificazione delle parti all'interno del file MIDI -->
<!ATTLIST MIDI_part_ref
        part_id IDREF #REQUIRED
        voice IDREF #IMPLIED
        spine_start_ref IDREF #IMPLIED
        spine_end_ref IDREF #IMPLIED
        track CDATA #REQUIRED
        channel CDATA #REQUIRED
>
<!ELEMENT CSOUND (part_ref+)>
<!ELEMENT MPEG4 (part_ref+)>
```

2.13 - Il Layer Audio

Il layer *audio* descrive le proprietà del materiale musicale audio. Si tratta del livello più basso nella nostra visione multi-strato.

I formati per la rappresentazione dell'informazione audio si possono suddividere in due categorie: compressi e non compressi. Tra questi ultimi troviamo PCM/WAV, AIFF e MuLaw. A loro volta, i formati compressi possono presentare perdita informativa (lossy) o meno (lossless). Tra i formati compressi con perdita ricordiamo MPEG e DOLBY, tra quelli senza perdita ADPCM e SHN.

Per poter collegare un generico file audio allo *spine*, è necessario estrarre le caratteristiche degli eventi musicali relative alla loro localizzazione temporale. Si tratta di informazioni che non dipendono dal particolare formato di codifica o dall'eventuale compressione.

2.14 - La codifica delle partiture nel formato IEEE 1599

Il formato IEEE 1599 per la strutturazione delle informazioni nella partitura stabilisce delle gerarchie. Il livello più alto, astrattamente, è quello costituito dall'intera partitura, mentre quello più basso è quello rappresentato dai simboli quali note e pause. La struttura può essere identificata in questi passi:

1. una partitura è costituita da più accollature
2. un'accollatura è formata da più pentagrammi
3. un pentagramma contiene (o meglio, può contenere) più parti
4. una parte può essere suddivisa in più voci
5. una voce intera viene considerata battuta per battuta
6. una battuta contiene accordi e pause
7. un accordo contiene una o più note.

Nel documento IEEE 1599 questa gerarchia genera il seguente scheletro:

```
<los>
        <!-- descrizione dell'accollatura -->
        <staff_list>
                <!-- descrizione delle proprietà dei pentagrammi -->
                <staff id="staff_1">   … </staff>
                <staff id="staff_2">   … </staff>
        </staff_list>
        <!-- descrizione delle singole parti -->
        <part id="flauto">
                <!--elenco delle voci della parte -->
                <voice_list>
                        <voice_item id="voce_flauto" staff_ref="staff_1"/>
                </voice_list>
                <measure number="1"> … </measure>
                <measure number="2">        … </measure>
                …
        </part>
        <part id="pianoforte">
                <voice_list>
                        <voice_item id="voce1" staff_ref="staff_2"/>
                        <voice_item id="voce2" staff_ref="staff_2"/>
                        …
                </voice_list>
                <measure number="1"> … </measure>
                <measure number="2">        … </measure>
```

```
            …
        </part>
        <!-- descrizione delle parti restanti… -->
</los>
```

Dove *staff_list* è l'accollatura (ossia l'elenco degli staff), staff è il pentagramma, *part* è la parte, *voice_list* è l'elenco delle voci in cui si divide la parte e *voice_item* è la singola voce.

Degna di nota è la costante presenza di identificatori (attributi id) per ciascun nuovo elemento gerarchico. Ciò consente di poter fare precisi riferimenti nei livelli inferiori. Ad esempio, questo permette di associare le part agli staff tramite l'attributo staff_ref di voice_item; o i voice in una measure ai voice_item della voice_list; o ancora i notehead che compongono gli accordi agli staff tramite l'attributo staff_ref.

Ora addentrandoci in livelli gerarchici più bassi troviamo le battute(measure), gli attributi di measure sono:

1. number, ossia il numero di battuta. Tale attributo logicamente è richiesto.
2. id, ossia un eventuale identificatore che potrebbe tornare utile in seguito per i riferimenti a questa particolare battuta da parte di un altro elemento. Si tratta di un attributo facoltativo.

Per identificazione di segnatura di tempo, di chiave e l'armatura per un ipotetica battuta corrente, sono descritte negli elementi figli di staff. Ad esempio, la chiave della battuta corrente si ricava scendendo in una voce (voice), considerandone gli eventi (rest o chord > notehead), risalendo dall'attributo staff_ref allo staff corrispondente e leggendo infine l'attributo, o gli attributi, key_signature figli di staff.[10]

- Capitolo 3 -

Progetto

3.1 - Breve cenno al linguaggio C#

C# è il primo linguaggio orientato ai componenti nella famiglia C e C++. E' un linguaggio di programmazione semplice, moderno, orientato agli oggetti e type-safe.
É il linguaggio con il quale è stato sviluppato il plugin di questo elaborato di tesi, per le sue caratteristiche implementative, e perché è il linguaggio su cui si basa tutto il framework del formato IEEE 1599, nel quale è inserito il plugin.

C# come già anticipato supporta la programmazione ad oggetti, questo significa che l'approccio alla programmazione è differente da linguaggi come C o Pascal, la metodica è di ragionare sui problemi che devono essere risolti dal programma di un computer e di affrontare il problema in maniera più intuitiva e dunque più produttiva. Infatti in un linguaggio object-oriented tutte le entità del dominio di un problema sono espresse attraverso il concetto di *oggetto.* Gli oggetti sono l'idea centrale nella programmazione ad oggetti, l'approccio richiede che un oggetto esegua dei lavori nel nostro interesse, per questo esso contiene dei dati ed i metodi che lavorano su tali dati. In questo modo l'oggetto è un insieme di funzionalità totalmente incapsulate.
C#, come la maggior parte dei linguaggi object-oriented, ha due tipi distinti: i tipi che sono intrinsechi al linguaggio (tipi primitivi) ed i tipi che possono essere creati dagli utenti (classi).
Le classi sono uno strumento per costruire strutture che contengano non solo dati ma anche il codice per gestirli. Ogni classe contiene degli oggetti che hanno le stesse

caratteristiche. Si avvale poi di relazioni di ereditarietà secondo le quali nuove classi di oggetti sono derivate da classi esistenti, ereditando le loro caratteristiche e estendendole con caratteristiche proprie.
Come tutti i costrutti che permettono di definire le strutture dati, una classe definisce un nuovo tipo di dato.

I membri di una classe sono:

- attributi (dati)
- metodi (procedure che operano su un oggetto)

La differenza sostanziale tra classi ed oggetti è che una classe è il progetto di un oggetto, cioè che un oggetto è un istanza di un tipo o classe.
Come già detto i metodi forniscono alle classi le loro caratteristiche comportamentali, quindi daremo loro un nome in funzione delle azioni che le classi devono esporre. I metodi vengono definiti e chiamati all'interno della classe a cui appartengono.[5],[7]

3.2 - Le classi del plugin IEEE1599toMIDI

Nel plugin da me creato ho implementato una sola classe di nome: ***IEEE1599toMIDI***, nella quale sono definiti i metodi di conversione ed esportazione dei due formati, ed inoltre è costruita l'interfaccia grafica, con la quale è possibile scegliere i criteri di creazione del file MIDI da esportare.
La classe viene scritta nel linguaggio C# nel seguente modo:

```
namespace IEEE1599toMIDI
{
   public partial class IEEE1599toMIDI : IEEE1599CommonInterface1
   {
      ….
    }
```

Questo codice dichiara le caratteristiche della classe. Questa interagisce con il framework IEEE 1599 cosi da richiamare i metodi e la classi implementate precedentemente che svolgono le azioni di apertura dei file XML del formato IEEE 1599 con il quale è possibile operare, nel mio caso esportandolo in un file MIDI.
All'interno delle graffe sono definiti gli attributi della classe e i metodi che sono descritti nel capitolo seguente.

3.3 - I metodi di inizializzazione del plugin IEEE1599toMIDI

I metodi che vengono eseguiti all'apertura del plugin IEEE 1599toMIDI sono due: uno di inizializzazione dell'interfaccia grafica e dei costrutti per la creazione dei file MIDI, l'altro di esecuzione di una query di ricerca nel file XML, aperto dall'utente.
I codici sono i seguenti:

```
public override bool DllInit(TabPage tab, ToolStripMenuItem menuItem)
        {
          this.tabPage = tab;
          this.menuItem = menuItem;

          // To use when class inherits from IEEE 1599CommonInterface1

          this.tabPage.Controls.Add(this.panelSpine);
          this.tabPage.Controls.Add(this.panelMidi);
          this.panelSpine.Visible = true;
          this.panelMidi.Visible = true;

            InitInstArray();
          builder = new ChannelMessageBuilder();
          sq = new Sequence();
          sec = new Sequencer();

          return true;
        }
```

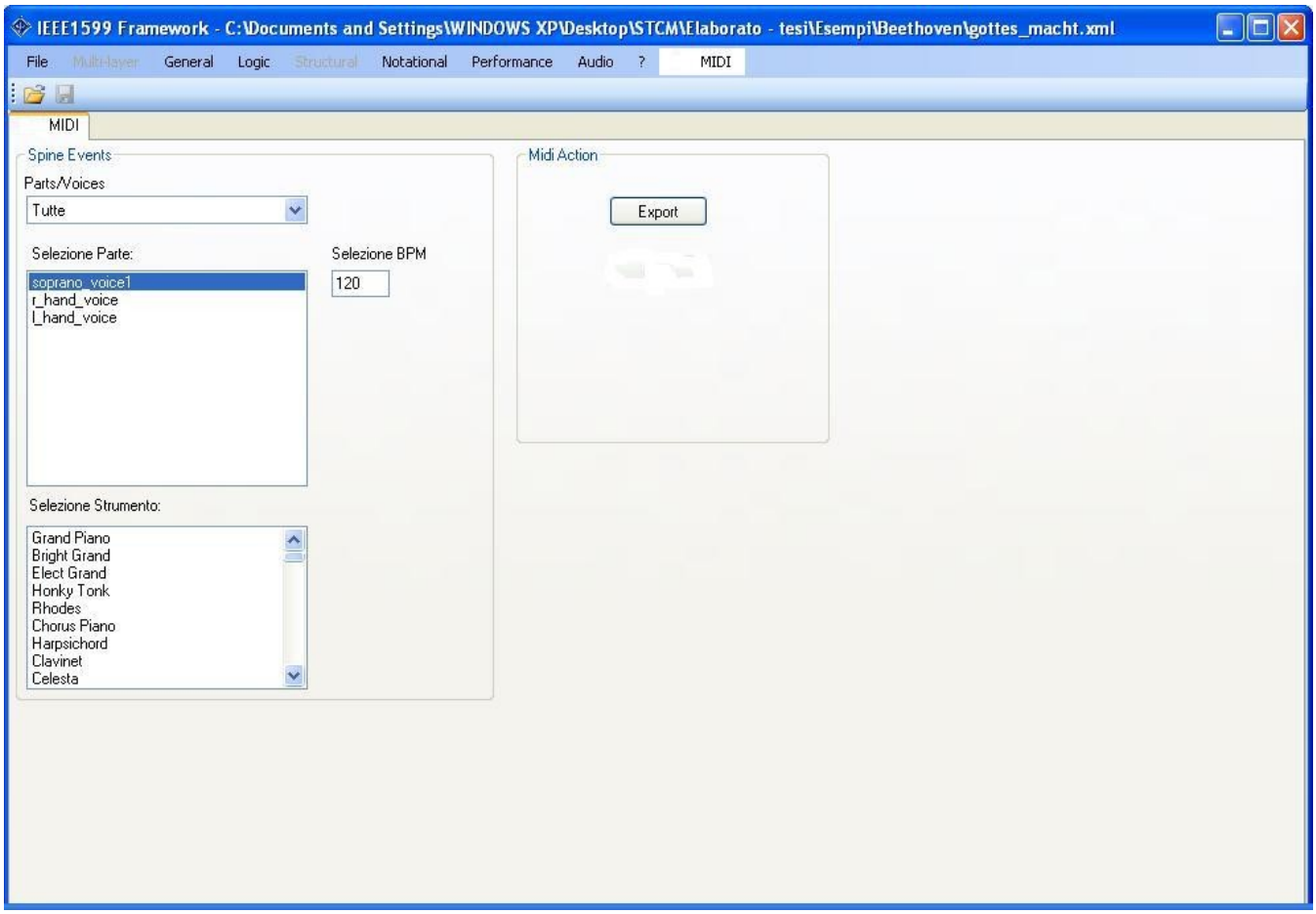
Questo è il primo codice relativo al metodo di inizializzazione, in esso vengono formattate le impostazioni dell'interfaccia all'interno di una finestra preimpostata (tabPage) che è relativa al framework IEEE15599. Inoltre vengono deifiniti i costruttori dei tipi di dato Sequence e Sequencer che sono le liste di dati nelle quali inserire le tracce MIDI prima create per poi poterle esportare in un formato standard ed eseguirle.

```
public override bool DllExec(IEEE 1599Document document)
      {

          XmlNodeList xList;
          this.xmlDocument = document;
          if (xmlDocument != null)
          {

              XmlNode xN2 =       document.Xml.SelectSingleNode("/IEEE
1599/logic/los/staff_list/staff/time_signature/time_indication");
              if (xN2 != null)
              {

                  // combobox filters
                  comboBoxFilter.Items.Clear();
```

```
            comboBoxFilter.Items.Add("Tutte");
            xList = document.Xml.SelectNodes("/IEEE 1599/logic/los/part/voice_list/voice_item");
            foreach (XmlNode xN in xList)
               comboBoxFilter.Items.Add(xN.Attributes["id"].Value);
            comboBoxFilter.SelectedIndexChanged += new EventHandler(comboBoxFilter_SelectedIndexChanged);
            if (comboBoxFilter.Items.Count > 0)
               comboBoxFilter.SelectedIndex = 0;
            xN2 = document.Xml.SelectSingleNode("/IEEE 1599/general/description/main_title");
            fileName = xN2.InnerXml;
            listBoxInstruments.Visible = false;
         }
         else
         {
            MessageBox.Show("Errore: Assenza valori Vtu");
         }
      }

      return true;
   }
```

Questo secondo metodo carica il file XML del formato IEEE 1599 in una variabile di tipo IEEE 1599Document di nome: XmlDocument. Nel quale vengono ricercate con una query le parti delle diverse voci dell'opera codificata. Queste parti hanno all'interno le definizioni degli eventi delle note eseguite, delle temporizzazioni di esse in relazione alle misure. Il risultato della query è inserita in una comboBox visualizzabile nell'interfaccia grafica con la quale è possibile selezionare la voce singola da esportare o tutte le voci insieme. Inoltre viene eseguita un operazione alternativa che ricerca nel documento IEEE 1599 i valori Vtu, se questi sono assenti perché non codificati ritorna un messaggio di errore e il plugin non può essere eseguito.

3.4 - L'interfaccia grafica del plugin IEEE1599toMIDI



Come si può notare l'interfaccia grafica è molto semplice ed intuitiva. Questa è la finestra che collega il codice all'utente. È possibile da essa selezionare le voci da esportare, (singola voce o tutte), e per ogni voce è possibile assegnare uno strumento che esegua la parte. Inoltre è necessario scrivere il valore di BPM con il quale il file MIDI sarà eseguito. Queste azioni sono inserite nel pannello SpineEvents perché sono relative ancora al primo formato IEEE 1599, infatti dopo queste selezioni il programma non crea ancora il file ma si prepara, memorizzandole in degli array o variabili temporanee. Nel secondo pannello, quello chiamo MidiEvents, è inserito un bottone che al momento della selezione richiama in principio il metodo costruttore della sequenza MIDI, successivamente descritto, dopo di che viene aperta un'ulteriore finestra per il salvataggio del file su una memoria di massa scelta dall'utente.

3.5 - Le librerie del plugin IEEE1599toMIDI

Le librerie di classi sono assolutamente importanti per fornire l'interoperabilità tra i linguaggi, in questo caso sono state importate tre diverse librerie che definiscono strutture e tipi di dato oltre che classi di oggetti.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using IEEE 1599StandardLibrary;
using IEEE 1599Framework;
using System.Xml;
using Sanford.Multimedia.Midi;
```

La libreria *System* è necessaria per tutte quelle operazioni di base della programmazione ad oggetti, contiene classi fondamentali e classi base che dichiarano i tipi di dati, di valore e di riferimento utilizzati comunemente, nonché eventi, gestori di eventi, interfacce, attributi ed eccezioni di elaborazione. In altre classi sono forniti servizi che supportano la conversione di tipi di dati, l'elaborazione di parametri di metodo, le operazioni matematiche, le chiamate di programmi locali e remoti, la gestione dell'ambiente dell'applicazione e la supervisione delle applicazioni gestite e non gestite.[6]

Le librerie *IEEE 1599StandardLibrary* e *IEEE 1599Framework* sono librerie create dagli sviluppatori del framework IEEE 1599 le quali contengono classi che definiscono gli eventi e le strutture dati dei file XML nei quali sono codificate l'informazione musicale delle opere. Con queste classi è stato possibile richiamare i file XML cosi da eseguire delle query e memorizzare i risultati. Inoltre sono dichiarate le classi di compilazione del framework così che il plugin IEEE1599toMIDI possa essere inserito in esso e compilato con tutto l'apparato implementativo.

La terza libreria *Sanford Multimedia Midi* è necessaria per tutte le operazioni

riguardanti la codifica MIDI: essa definisce le classi di tutti i tipi di messaggi MIDI e le strutture come sequenze e tracce, oltre che le modalità di temporizzazione insite nel protocollo MIDI. Data la sua importanza sarà trattata approditamente nel capitolo successivo.

3.6 - La libreria Sanford Multimedia MIDI

Questa libreria descrive le caratteristiche del protocollo MIDI, le strutture, i messaggi, le tipologie di temporizzazione, ecc. così da renderle implementabili in qualsiasi programma scritto nel linguaggio C#. Le specifiche della libreria e il toolkit (esempio di sviluppo di un sequencer), sono consultabili al sito: **http://www.lesliesanford.com/Programming/MIDIToolkit.shtml**.
Di questa libreria sono state utilizzate solo tre classi, che sono: **MIDI messages**, **Track class** e **Sequence class**. Esse hanno permesso di generare il file MIDI in maniera semplice e corretta.
Le tipologie di **MIDI message** definite sono:

- IMidiMessage
- ShortMessage
- ChannelMessage
- SysCommonMessage
- SysRealtimeMessage
- SysExMessage
- MetaMessage

Come si può notare si hanno delle corrispondenze con la codifica MIDI, sono stati implementati i tipi ChannelMessage e MetaMessage. I primi per generare i messaggi di Note On, Note Off e i Program Change, i secondi per la temporizzazione della sequenza.

La **Track class** rappresenta una collezione di MidiEvents. E' responsabile per il

mantenimento di un insieme di MidiEvents in modo corretto. I MidiEvents non sono direttamente aggiunti a una traccia. Invece, si aggiunge un MidiMessage, precisando la sua posizione assoluta nella traccia. La **Track class** crea quindi un MidiEvent per rappresentare il messaggio e lo inserisce nella sua collezione di MidiEvents.

La **Sequence class** rappresenta una collezione di Track. Esso fornisce inoltre le funzionalità per il caricamento e il salvataggio di file MIDI, in modo che la sequenza si in grado di caricare e salvare se stessa.

3.7 - L'analisi degli eventi del formato IEEE 1599

Per creare il file MIDI relativo al documento IEEE 1599 vanno analizzati e quindi ricercati nei vari layer gli eventi relativi a note, pause temporizzazioni di esse. Questi eventi sono estrapolati dai sottolivelli *los*, per quanto riguarda le note, le pause e per il valori temporali delle singole voci (VTU e metriche rigo musicale), nello *spine* invece sono presenti i valori temporali delle singole note o pause, attributo timing. Come già riferito nel capitolo 1.2, ogni nota ha un indicativo univoco che serve al collegamento dei livelli, in questo caso al legame tra *los* e *spine*.
La parte di codice che serve a quest'analisi è il seguente:

```
XmlNodeList xList = xmlDocument.Xml.SelectNodes("/IEEE 1599/logic/los/part/measure/voice[@voice_item_ref='" + listBoxParts.Items[i] + "']/*");
            foreach (XmlNode xN in xList)
              {
              // inserisci note in traccia t
              String currentId = xN.Attributes["event_ref"].Value;
              object o = xmlDocument.CreateLosObject(currentId);
               XmlNode xN3 = xmlDocument.Xml.SelectSingleNode("/IEEE 1599/logic/los/part/voice_list/voice_item[@id='" + listBoxParts.Items[i] + "']");
                          XmlNode xN4 = xmlDocument.Xml.SelectSingleNode("/IEEE 1599/logic/los/staff_list/staff[@id='" + xN3.Attributes["staff_ref"].Value + "']/time_signature/time_indication");
```

```
int currentVtu = Convert.ToInt16(xN4.Attributes["vtu_amount"].Value);
int currentDenBat = Convert.ToInt16(xN4.Attributes["den"].Value);
int currentNumBat = Convert.ToInt16(xN4.Attributes["num"].Value);
....
```

Le voci differenti come già visto nel capitolo 2.3 vengono caricate nella comboboxFilter nel metodo di inizializzazione *DllExec*. Se viene selezionata l'opzione tutte gli indicativi delle parti vengono caricati nella listBoxParts nella quale è possibile definire gli strumenti che eseguiranno la parte. Per la ricerca viene instaurato un for che prende in consegna tutte le parti singolarmente e con la prima riga del codice, prima evidenziato, viene implementata una lista di eventi (xList), nella quale vengono aggiunti i valori degli attributi dei singoli eventi. Per fare un esempio vengono aggiunti alla lista i seguenti valori del documento IEEE 1599 :

```
<voice voice_item_ref="soprano_voice1">
        <chord event_ref="v1_e2">
          <duration num="1" den="2" />
          <notehead staff_ref="staff_1">
            <pitch step="B" octave="5" />
          </notehead>
        </chord>
      ....
```

Questa parte di codice del documento IEEE 1599 è relativo alle caratteristiche di una singola nota che il mio Plugin rileva da esso e inserisce nella lista di eventi tutti i valori degli attributi. Si può facilmente notare le indicazioni di altezza, pitch, di durata, duration, l'identificativo univoco che lo lega agli altri livelli del formato IEEE 1599, event_ref, e l'indicazione della staff a cui fa riferimento, Staff_ref.

Creata la lista di eventi, xList, che contiene tutti i valori sopra citati di ogni evento, della singola parte. Questa viene passata in rassegna con il Foreach e vengono caricate in delle variabili temporanee le indicazioni di metrica e vtu_amount a cui fanno riferimento utilizzando per la ricerca il collegamento dato dal valore di Staff_ref. Queste variabili serviranno per calcolare il corrispettivo valore di tick della nota nel messaggio MIDI.

Le operazioni citate vengono eseguite per ogni evento inserito nel documento IEEE

1599 e per ogni parte, in maniera automatica e contemporaneamente vengono creati ed inseriti nelle track i messaggi MIDI.

3.8 - La generazione dei messaggi MIDI e delle sequenze

Il codice di generazione dei messaggi MIDI e l'analisi del documento IEEE 1599 trattata nel capitolo precedente sono inseriti nel metodo denominato MidiFileCostructor(), in questo metodo viene inoltre mappato il documento IEEE 1599 creando o modificando il livello *performance*; Inserendo gli attributi dei tag figli impostati nello schema gerarchico logico del formato IEEE 1599.
Questo metodo si può dire che sia il cuore del plugin da me creato, in esso sono implementate le operazioni più importanti al fine di raggiungere l'obbiettivo da me proposto.
In questo capitolo verranno trattate solo delle operazioni riguardanti la generazione dei messaggi MIDI, relativi a note e pause lasciando la valutazione dei messaggi Program Change e di temporizzazione del file in capitoli a parte, rispettivamente 2.9 e 2.10.

Le prime operazioni fatte da questo metodo, per lo scopo di generare i messaggi, sono quelle relative all'inizializzazione delle strutture dati nelle quali saranno inseriti i messaggi. Il codice è il seguente:

```
trList.Clear();
trList.Capacity = listBoxParts.Items.Count;
sq.Clear();

for (int i = 0; i < trList.Capacity; i++)
            {
             Track tr = new Track();
               tr.Clear();
               trList.Add(tr);
        }
```

In questo stralcio di codice sono identificabili tre strutture dati: trList, sq, tr.
La prima è una lista di track che viene creata nel metodo di inizializzazione (dllInit) e

qui viene depurata di eventuali tracce inserite in esecuzioni precedenti del plugin, e poi si da un indicazione di quante tracce dovra contenere, questo valore è dato dal numero di voci codificate nel documento IEEE 1599 e visualizzate nella già descritta listBoxParts.

La seconda struttura dati è una sequenza MIDI chiamata sq, alla quale viene svolta un operazione di "pulitura" e cioè di cancellazione di eventuali tracce precedenti. In essa verranno inserite le tracce contenute nella lista di track e sarà il prodotto ultimo che verra poi esportato.

L'ultima struttura dati viene dichiarata e creata nel for ed è la struttura nella quale verranno inseriti i messaggi generati, essa è una track. Il numero di track generate varia a seconda del numero di voci presenti nel documento IEEE 1599. Sempre nel for esse sono inserite nella lista di track.

Al momento di uscita dal for ci si trova nella situazione di avere le strutture dati create vuote, o per l'esattezza avranno al loro interno dei valori NULL.

Il codice che segue è relativo alla generazione dei messaggi di note on e note off . É stato volontariamente depurato delle operazioni dei analisi e di mappatura per maggior chiarezza.

```
for (int i = 0; i < listBoxParts.Items.Count; i++)
                ….//Analisi documento IEEE 1599
                if (o is Chord) //struttura condizionale che rileva se l'evento è un accordo o nota singola
                {
                  Chord chord = o as Chord; // definizione di un tipo dato accordo nel quale sono inseriti i valori dell'evento
                  DurationType dur = chord.Duration; // definizione di un tipo dato durata nel quale sono inseriti i valori metrici
                  //creazione del messaggio di NoteOn con generatore di messaggi channelBuldier definito dalla libreria SanfordM
                  foreach (Notehead nh in chord.ChordNotes) // Per ogni nota dell'accordo viene generato il messaggio
                  {
                    builder.Command = ChannelCommand.NoteOn; // Definizione tipo messaggio
                    builder.MidiChannel = i; // definizione numero del canale, il valore è incrementale per ogni traccia
                    builder.Data1 = nh.MidiPitch; // definizione valore del pitch
                    builder.Data2 = 127; // definizione valore key velocity fisso a 127, valore massimo
                    builder.Build(); // metodo di creazione
                    // inserimento del messaggio nella track, più valore del tick assoluto dove inserire il messaggio
                    trList[i].Insert(tick, builder.Result);
                  }
                  //comando noteoff
                  foreach (Notehead nh in chord.ChordNotes)// Per ogni nota dell'accordo viene generato il messaggio
                  {
                    builder.Command = ChannelCommand.NoteOff;
                    builder.MidiChannel = i;
                    builder.Data1 = nh.MidiPitch; // riferimento al messaggio Note on
                    builder.Data2 = 0;
                    builder.Build();

                    // calcolo durata temporale dell'evento, conversione in MIDI ticks
                    int currentTick = tick + chord.CalculateMidiTicksDuration((currentVtu * currentNumBat) / currentDenBat);
```

```
                trList[i].Insert(currentTick, builder.Result);
            }
             // aggiornamento incremento temporale o tick assoluto, necessario per posizionamento messaggio successivo
            tick = tick + chord.CalculateMidiTicksDuration((currentVtu * currentNumBat) / currentDenBat);

        else if (o is Rest) // alternativa evento pausa
        {
            Rest rest = o as Rest; // dicharazione variabile temporale rest con i valori dell'evento
            DurationType dur = rest.Duration;
             // calcolo durata temporale dell'evento, conversione in MIDI ticks
            int currentRelTick = rest.CalculateMidiTicksDuration((currentVtu * currentNumBat) / currentDenBat);
            // aggiornamento incremento temporale o tick assoluto, necessario per posizionamento messaggio successivo
            tick = tick + currentRelTick;
        }
    }
    sq.Add(trList[i]); // inserimento tracce nella struttura dati sequence
}
```

Logicamente questo stralcio di codice prende per ogni voce del documento IEEE 1599 l'analisi del singolo evento nota o pausa già descritta, e crea: se questo è una nota o un accordo, i relativi messaggi di *note on* e *note off* insieme al valore temporale dell'evento convertito in tick del protocollo MIDI, i messaggi ogni volta generati vengono inseriti nella track selezionata appositamente per la voce di cui fa parte. Se l'evento è una pausa invece non verrà creato nessun messaggio, perché, ricordo, per il protocollo MIDI le pause sono definite da un assenza di messaggi e quindi di esecuzione, ma viene solo calcolato il valore temporale del evento e cosi incrementato il valore di tick assoluto cioè l'incremento temporale dove sarà inserito il messaggio successivo. Dopo di che le track sono inserite nella sequenza che sarà la struttura dati ultime per creare il file MIDI Per avere una visione più dettagliata suggerisco di vedere i commenti del codice.

3.9 - La mappatura del documento IEEE 1599

Eseguendo il plugin è possibile aggiornare o generare il livello *performance* relativo al file MIDI che si sta creando. Questa operazione avviene in automatico mentre si costruiscono le tracce MIDI e solo alla chiusura del plugin è possibile dare indicazioni di salvataggio del documento IEEE 1599 o no.

Come già detto nel capitolo riservato al Layer *performance* esso definisce i tratti essenziali di un esecuzione sintetica del pezzo. In questo Layer vengono indicate le informazioni con una struttura gerarchica degli elementi che sono:

- Performance
  - Midi_instance
    - Midi_mapping
      - Midi_event_sequence
        - Midi_event

In ognuno di questi elementi sono contenuti degli attributi inerenti alle informazioni di ogni elemento di cui fanno parte.

Lo scopo della mappatura è quella di inserire nel documento IEEE 1599 questi elementi, posizionati in maniera consona alla struttura gerarchica ed i loro attributi.

Per fare questo ho utilizzato del codice che andando ad agire direttamente sul documento scriva le stringhe in formato IEEE 1599 da me implementate.

```
XmlNode xn = xmlDocument.Xml.SelectSingleNode("/IEEE 1599/performance/midi_instance[@file_name='" + fileName
+"']");
        if (xn != null)
            xn.RemoveAll();
        //crea il Layer Performance e l'elemento figlio MIDI_istance
        else
        {
            XmlNode tempNode = xmlDocument.Xml.CreateDocumentFragment();
            tempNode.InnerXml = "<performance><midi_instance file_name=\"" + fileName + "\"
format=\"1\"></midi_instance></performance>";
            xmlDocument.Xml.SelectSingleNode("/IEEE 1599").AppendChild(tempNode);
        }
```

Questo è il primo passo nel quale è verificato se esistano mappature precedenti, e nel caso siano presenti vengono eliminati gli elementi figli, poi aggiornato con le nuove

informazioni. Nel caso in cui non siano mai state fatte operazioni di mappatura del Layer *performance* questo elemento e l'elemento figlio midi_instance vengono generati con il metodo AppendChild.

Dopo queste prime mappature sono inseriti gli elementi figli prima citati:

```
//Mappatura singola parte
            xn = xmlDocument.Xml.SelectSingleNode("/IEEE 1599/performance/midi_instance");
            XmlNode tempNode2 = xmlDocument.Xml.CreateDocumentFragment();
             tempNode2.InnerXml = "<midi_mapping part_ref=\"" + listBoxParts.Items[i] + "\" track=\"" + i + "\" channel=\""
+ i + "\"><midi_event_sequence division_type=\"???\" ></midi_event_sequence></midi_mapping>";

//Mappatura singolo evento
                XmlNode tempNode3 = tempNode2.SelectSingleNode("/midi_mapping/midi_event_sequence");
                XmlNode tempNode4 = xmlDocument.Xml.CreateDocumentFragment();
                    tempNode4.InnerXml = "<midi_event timing=\"" + chord.CalculateMidiTicksDuration((currentVtu *
currentNumBat) / currentDenBat) + "\" spine_ref=\"" + currentId + "\"></midi_event>";
                tempNode3.AppendChild(tempNode4);
```

In questo modo sono descritte tutte le informazioni degli eventi dell'esecuzione del file MIDI generato.

3.10 - La temporizzazione

Questo capitolo tratta di un importante punto del programma, cioè la temporizzazione del file MIDI da generare. Infatti un'esecuzione anche nella classica accezione, cioè fatta da strumentisti, può essere svolta in varie velocità e l'importante è che tutte le voci o strumenti che ne prendono parte siano temporizzate a doc.

Nell'interfaccia del mio plugin è possibile decidere a quale velocità, misurata in valore BPM, verrà poi eseguito il file MIDI.

Importante è ricordare che i file MIDI non sono temporizzati con l'unità di misura classica, i BPM, ma ci sono varie modalità di definirla.

Nel mio caso ho svolto un'operazione di conversione, da BPM a messaggio MetaMessage di tipo *Temp*.

Di seguito mostro il metodo che contiene questa conversione:

```
private void textBoxBPM_MouseClick(object sender, MouseEventArgs e)
      {
         int x0 = 60000000 / Convert.ToInt16(textBoxBPM.Text);
         a = Convert.ToByte(x0 / 65536);
         int x1 = x0 - (65536 * Convert.ToInt16(a));
         b = Convert.ToByte(x1 / 256);
         c = Convert.ToByte(x1 - (256 * Convert.ToInt16(b)));
      }
```

Questa operazione è eseguita quando viene inserito dall'utente il valore numerico dei BPM. Nel messaggio MIDI per la temporizzazione c'è bisogno di una codifica in byte del valore in millesimi di secondi del quarto, suddiviso in blocchi di tre numeri, che corrispondono ai valori di unita-decine (c), centinaia-migliaia (d), milionesimi-miliardesimi (a) in esadecimale del valore temporale, poi convertito in byte dall'operazione Convert.
Questo byte è utilizzato poi nel messaggio MetaMassage che viene inserito nella prima posizione della sequenza. Il codice è il seguente:

```
MetaMessage mm = new MetaMessage(MetaType.Tempo, new byte[] { a, b, c });
            trList[i].Insert(0, mm);
            tick = 1;
```

Questa parte di codice è inserita nel metodo MidiFileCostructor() già descritto in precedenza.

3.11 - I messaggi Program Change

Nel plugin IEEE 1599toMIDI c'è la possibilità di definire quali strumenti eseguiranno le voci definite in partitura. Gli strumenti possibili sono dati da una libreria di suoni già presente su ogni computer e dichiarate da uno standard denominato General MIDI.
Il General MIDI, abbreviato per comodità in GM, è un insieme di specifiche definite nel 1991 dalla MMA (MIDI Manufacturers Association) e dal JMSC (Japan MIDI Standards Committee), con l'intenzione di fornire una certa compatibilità tra diversi strumenti MIDI. La caratteristica principale del General MIDI è quella di fissare la

corrispondenza tra program change e strumento assegnato.

Per fornire questa associazione ho creato un Array di stringhe nel quale sono inseriti i nomi dei 127 strumenti a disposizione con un'indice che ne definisce il valore Prog.Change. Gli strumenti sono i seguenti:

Piano

1 - Piano 1
2 - Piano 2
3 - Piano 3
4 - Honky-Tonk Piano
5 - Electric Piano 1
6 - Electric Piano 2
7 - Harpsichord
8 - Clav.

Bass

33 - Acoustic Bass
34 - Fingered Bass
35 - Picked Bass
36 - Fretless Bass
37 - Slap Bass 1
38 - Slap Bass 2
39 - Synth Bass 1
40 - Synth Bass 2

Lead

65 - Soprano Sax
66 - Alto Sax
67 - Tenor Sax
68 - Baritone Sax
69 - Oboe
70 - English Horn
71 - Bassoon
72 - Clarinet

Synth FX

97 - Ice Rain
98 - Soundtrack
99 - Crystal
100 - Atmosphere
101 - Brightness
102 - Goblin
103 - Echo Drops
104 - Star Theme

Chromatic percussion

9 - Celesta
10 - Glockenspiel
11 - Music Box
12 - Vibraphone
13 - Marimba
14 - Xylophone
15 - Tubolar Bell
16 - Santur

Strings/Orchestra

41 - Violin
42 - Viola
43 - Cello
44 - Contrabass
45 - Tremolo Strings
46 - Pizzicato
47 - Harp
48 - Timpani

Pipe

73 - Piccolo
74 - Flute
75 - Recorder
76 - Pan Flute
77 - Bottle Blow
78 - Shakuhachi
79 - Whistle
80 - Ocarina

Ethnic

105 - Sitar
106 - Banjo
107 - Shamisen
108 - Koto
109 - Kalimba
110 - Bagpipe
111 - Fiddle
112 - Shanai

Organ

17 - Organ 1
18 - Organ 2
19 - Organ 3
20 - Church Organ
21 - Reed Organ
22 - Accordion
23 - Harmonica
24 - Bandoneon

Ensemble

49 - Strings
50 - Slow Strings
51 - Synth Strings 1
52 - Synth Strings 2
53 - Choir Aahs
54 - Voice Oohs
55 - Synth Voice
56 - Orchestra Hit

Synth Lead

81 - Square Wave
82 - Saw Wave
83 - Synth Calliope
84 - Chiffer Lead
85 - Charang
86 - Solo Vox
87 - 5th Saw Wave
88 - Bass & Lead

Percussive

113 - Tinkle Bell
114 - Agogo
115 - Steel Drums
116 - Woodblock
117 - Taiko
118 - Melo Tom
119 - Synth Drum
120 - Reverse Cymbal

Guitar

25 - Nylon-str. Guitar
26 - Steel-str. Guitar
27 - Jazz Guitar
28 - Clean Guitar
29 - Muted Guitar
30 - Overdrive Guitar
31 - Distortion Guitar
32 - Guitar Harmonics

Brass

57 - Trumpet
58 - Trombone
59 - Tuba
60 - Muted trumpet
61 - French Horns
62 - Brass 1
63 - Synth Brass 1
64 - Synth Brass 2

Synth Pad

89 - Fantasia
90 - Warm Pad
91 - Polysynth
92 - Space Voice
93 - Bowed Glass
94 - Metal Pad
95 - Halo Pad
96 - Sweep Pad

Sound FX

121 - Guitar FretNoise
122 - BreathNoise
123 - Seashore
124 - Bird
125 - Telephone
126 - Helicopter
127 - Applause
128 - GunShot

L'utente ha la possibilità di scegliere lo strumento, il valore Prog. Change viene memorizzato in una lista di variabili temporali. Il codice di generazione del messaggio Program Change è il seguente:

```
//inserimento valore program change
            builder.Command = ChannelCommand.ProgramChange;
            builder.MidiChannel = i;
            builder.Data1 = progChange[i];
            builder.Build();
            trList[i].Insert(tick, builder.Result);
```

Questo stralcio di codice è inserito nel metodo MidiFileCostructor() e genera il messaggio per ogni voce dell'opera, il numero della voce è definita dalla variabile indice i.

3.12 - L'esportazione del file

In questo capitolo viene esposto come in generale il plugin da me creato esegue i metodi di generazione dei messaggi e salva il file finale MIDI su supporto di massa.
Le operazioni di esportazione vengono chiamate dal pulsante dell'interfaccia grafica *Export.* Dal click di esso viene richiamato un metodo che oltre ad eseguire il metodo fondamentale di generazione del file MidiFileCostructor(), già descritto, apre una finestra classica di salvataggio implementata dal sistema operativo Windows. Il codice del metodo è il seguente:

```
private void Export_Click(object sender, EventArgs e)
      {
         MidiFileCostructor();
         progChange.Clear();
         //Esportazione del file midi inserito nella sequenza
         SaveFileDialog sfd = new SaveFileDialog();
         sfd.DefaultExt = ".mid";


         if (sfd.ShowDialog() == DialogResult.OK)
         {
            sq.SaveAsync(sfd.FileName);
         }

      }
```

All'uscita del metodo MidiFileCostructor() è stata generata e riempita la struttura dati sq (sequenza MIDI) che ha come metodo di salvataggio *SaveAsync,* implementato dalla

libreria Sanford Multimedia Midi, con il quale è possibile attribuire al file un estensione, che di default ho assegnato il formato MIDI (.mid), e cosi designargli un'altra posizione nella memoria di massa scelta dall'utente.
Ogni volta che il pulsante Export viene premuto i valori della lista program Change (progChange) vengono eliminati, cosi da poter fare altre prove assegnando strumenti differenti alle varie voci.

- Conclusioni e sviluppi futuri -

Con il presente elaborato si è studiata una nuova tipologia di esecuzione delle opere musicali, sia quelle già codificate nel formato IEEE 1599, che quelle che saranno codificate, così da creare un archivio sintetizzato e alternativo ai file audio.
Si è data la possibilità di esplorare le partiture e di sperimentare su di esse sviluppando varianti delle ensemble orchestrali per cui erano pensate o variando il tempo di esecuzione, così da avere delle nuove prospettive su delle opere che sono nel bagaglio culturale dei fruitori.
Si è modificato inoltre il formato IEEE 1599 cosi da poter aggiungere o aggiornare il livello *performance* riferito alle esecuzioni sintetiche del file MIDI.
Le versioni future del plugin IEEE1599toMIDI potranno prendere in considerazione anche le informazioni di dinamica dell'opera andando ad agire sui valori di velocity del MIDI.
Si potranno sviluppare altre interazioni e sperimentazioni da parte dell'utente; come creare un sequencer interno, con la possibilità di modificare in tempo reale i valori preimpostati.

Bibliografia

- [1] David Miles Huber, *The MIDI Manual*, Focal Press
- [2] Jeffrey Rona, *The MIDI Companion*, Hal Leonard Cor
- [3] Giovanni Perotti, *MIDI Computer immagine e suono*, Jackson libri
- [4] Enrico Paita, *Computer e musica*, Jackson libri
- [5] AAVV, *Professional C#,* Wronx
- [6] Barbara Doyle, *C# Programming,* Thomson Course Technology
- [7] Tom Archer, *Inside C#*, Mondadori Informatica
- [8] Wyke-Rehman-Leupen, *Programmare XML*, Mondadori Informatica
- [9] Galante-Sani, *Musica Espansa Percorsi elettroacustici di fine millennio*, LIM (le sfere)
- [10] Otto Karolyi, *La grammatica della musica*, Einaudi

Ringraziamenti

Un sentito ringraziamento al professor Ludovico ed al dottor Baratè che mi hanno aiutato nei momenti di stallo per far funzionare il plugin.
Un ringraziamento speciale va: a mia madre che mi ha supportato e spronato, a mio padre che sarebbe sicuramente fiero del mio lavoro, ai miei fratelli e a tutti i miei amici.