UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

Corso di Laurea triennale in
Informatica Musicale

UNA LIBRERIA C PER IL CARICAMENTO E LA MANIPOLAZIONE DI DOCUMENTI IEEE 1599

Relatore: Prof. Luca Andrea Ludovico
Correlatore: Prof. Federico Simonetta

Tesi di Laurea di:
Alessandro Talamona
Matr. Nr. 895744

ANNO ACCADEMICO 2019-2020

Indice

Capitolo 1

Introduzione

Il presente elaborato è organizzato come segue: nel Capitolo 1, dopo aver definito lo scopo del software e aver chiarito il concetto di ‘‘libreria’’, viene trattato il formato IEEE1599 descrivendo le principali caratteristiche. Viene infine accennata una breve introduzione alle Forgeign Function Interface.

Il Capitolo 2 presenta le tecnologie utilizzate per lo sviluppo del codice e l'analisi del materiale informatico sui cui è stato svolto il lavoro.

Nel Capitolo 3 si descrive come vengono gestiti dal software prima il documento XML come file di testo e poi i vari layer previsti dal formato IEEE1599. Infine vengono esposti i principali problemi riscontrati in fase di progettazione.

Nel Capitolo 4 viene analizzato il procedimento con cui è stato sviluppato ciascun modulo in cui è suddiviso il software, ponendo l'attenzione, per ciascuno di essi, sugli aspetti più importanti e problematici e sulle soluzioni più interessanti dell'implementazione.

Nel Capitolo 5 vengono mostrati alcuni risultati ottenuti durante la fase di testing che permettono di confermare il raggiungimento degli obiettivi preposti.

Il Capitolo 6, infine, è dedicato alle conclusioni e propone alcune idee per eventuali sviluppi futuri.

1.1 Scopo del software

Lo scopo di questo elaborato è quello di costruire una libreria software che consenta il caricamento e quindi la manipolazione dei dati contenuti in documenti che rispettino lo standard IEEE1599, del quale verranno descritte le caratteristiche di interesse nella successiva sezione.

Una libreria è costituita, tra le altre cose, da una moltitudine di strutture dati e funzioni che hanno obiettivi diversi e tra queste, nel nostro caso, si vogliono creare

le strutture dati per contenere ogni tipo di informazione ottenibile dai documenti IEEE 1599 e le funzioni che permettano il caricamento dei dati in tali strutture.

Ulteriore obiettivo è quello di consentire l'utilizzo in diversi linguaggi di programmazione della libreria così sviluppata, attraverso le Foreign Function Interface (FFI): un meccanismo per l'interfacciamento tra linguaggi di programmazione diversi di cui verrà introdotta la definizione più avanti.

1.2 Librerie condivise

Le librerie condivise possono essere intese come insiemi di risorse scritte in linguaggio di programmazione che servono a implementare interfacce verso un certo problema, che nel caso particolare del presente progetto di lavoro risulta essere la manipolazione dell'informazione contenuta nei documenti IEEE 1599.

Il vantaggio principale offerto è quello di rendere disponibili queste risorse ad altri software che le vogliono sfruttare riutilizzando lo stesso codice.

Permettono quindi di riorganizzare il modo in cui sono organizzati progetti complessi, diminuendone i tempi di compilazione.

Esistono due tipi di librerie: quelle statiche e quelle dinamiche o condivise. In entrambi i casi si tratta comunque di file compilati che, in fase di linking, possono essere incorporati come un'unica entità nel programma che le utilizza.

Le librerie statiche vengono incorporate nell'eseguibile prima della sua esecuzione e solitamente hanno estensione `lib` o `a`.

Se più programmi eseguiti contemporaneamente utilizzano la stessa libreria statica, ne viene caricata una copia per ognuno di essi, sprecando memoria per contenere lo stesso codice.

Le librerie condivise servono ad ovviare questo problema permettendo di condividere il codice di una singola copia della libreria tra diversi programmi.

Le librerie condivise non vengono inserite direttamente nel file eseguibile. È in fase di run time che queste vengono collegate al programma in esecuzione.

L'estensione utilizzata dipende dal sistema operativo. Ad esempio, Windows utilizza i file `dll`, Linux quelli `so`.

Per lanciare un eseguibile che utilizza librerie condivise, bisogna specificare la posizione di quest'ultime.

La convenienza rispetto all'uso delle librerie statiche è che se la libreria condivisa viene aggiornata, non serve ricompilare il programma che la utilizza per poter applicare le modifiche: basta ricompilare la libreria [1].

1.3 IEEE 1599

IEEE 1599 è un formato basato su XML per la rappresentazione musicale, sviluppato dal Laboratorio di Informatica Musicale (LIM) e divenuto standard nel 2008, che descrive contenuti musicali eterogenei in modo completo [2].

In un unico file, simboli musicali, spartiti, tracce audio, performance computer-driven, metadati per la catalogazione, testi e contenuti grafici relativi a una particolare brano musicale vengono collegati e sincronizzati reciprocamente all'interno della stessa struttura [3]. Contenuti eterogenei vengono organizzati in una struttura multi-livello che supporta diversi formati di codifica e un numero di elementi digitali per ogni livello [4].

I livelli, o *layer*, definiti dal formato sono [5]:

- *General*, che contiene i metadati relativi al brano in oggetto, tra cui le informazioni catalografiche su titolo dell'opera, autori e genere;
- *Logic*, vero nucleo del formato, destinato alla descrizione simbolica dei contenuti musicali;
- *Structural*, che identifica gli oggetti musicali su cui il brano è costruito e permette di evidenziarne i mutui rapporti;
- *Notational*, che contiene le differenti rappresentazioni grafiche della partitura, ad esempio riferibili a diverse edizioni o trascrizioni;
- *Performance*, che è dedicato ai formati per la generazione di esecuzioni sintetiche da parte dell'elaboratore;
- *Audio*, che consente di legare al brano in oggetto le esecuzioni audio/video della partitura.

Figura 1: Struttura del formato IEEE 1599.

1.4 Foreign function interface

In fase di progettazione del software, è stato presa in considerazione l'eventualità di poter scrivere un software utilizzabile in quanti più ambienti di sviluppo possibile.

Con Foreign Function Interface (FFI) si intendono quei meccanismi usati per interfacciare un linguaggio di programmazione con un altro [6]. Questi hanno lo scopo di convertire e adattare la semantica e le routine tra i vari linguaggi.

Solitamente, il procedimento di traduzione prevede alcune strategie comuni, come l'utilizzo di funzioni *wrapper* che fanno da collante tra due linguaggi.

Ogni linguaggio definisce le proprie FFI verso altri. Un esempio può essere JNI (Java Native Interface), un framework che permette a Java di essere compatibile con questo tipo di interazioni verso altri linguaggi e in particolare verso il C/C++.

Esistono poi alcuni strumenti, come il software SWIG, che permettono, tramite le FFI, di collegare diversi linguaggi attraverso un unico sorgente.

Capitolo 2

Tecnologie utilizzate

In questo capitolo vengono presentate le principali tecnologie utilizzate per lo sviluppo del software. Vengono trattati:

- Il linguaggio C, elencando le funzionalità di programmazione elementari più utilizzate.
- XML e DTD, che sono gli strumenti con cui è definito il formato IEEE 1599.
- XPath, il linguaggio che permette di muoversi all'interno di documenti XML.
- libxml2, la libreria che permette l'interazione tra C e XML sui cui si basa l'implementazione del codice del presente elaborato.

2.1 C

Il linguaggio di programmazione scelto è il C in modo da favorire future estensioni verso altri linguaggi, visto che. su quest'ultimo, sono basati molti di quelli di alto livello, come per esempio Python, C++, Java, Javascript e Perl [7].

Una delle funzionalità più utilizzate del linguaggio è stata la possibilità di creare tipi di dati strutturati, o strutture, attraverso la parola chiave `struct`. Queste permettono l'aggregazione di più variabili, ma, a differenza degli array, non in modo ordinato e omogeneo poiché una struttura può contenere variabili di tipo diverso.

É possibile definire nuovi tipi di dato anteponendo la parola chiave `typedef` alla dichiarazione di una struttura e specificando il nome con cui riferircisi. Agli elementi del tipo di dato così creato si può accedere tramite l'operatore ‘‘.’’

Esempio 2.1: Definizione di un tipo di dato combinando `typedef` e `struct`.

```
typedef struct elemento{
```

```
    int informazione;
} nome_tipo;
```

Il linguaggio prevede anche un particolare tipo di dato, `union`, che permette di memorizzare, in istanti diversi, oggetti differenti per dimensione e tipo ma aventi lo stesso ruolo all'interno del programma.

Esempio 2.2: Definizione di un tipo di dato `union`.

```
union nome_union{
    int numero;
    char carattere;
    struct nome_struct;
};
```

Tra i concetti fondamentali del linguaggio C c'è quello di *puntatore*, perché permette, tra le altre cose, di poter astrarre in modo semplice strutture dati complesse.

Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile e si dichiara anteponendo il carattere `*` al nome della variabile.

L'asterisco è anche l'operatore di indirezione, cioè restituisce l'oggetto puntato dal puntatore.

L'operatore `&`, davanti a un nome di variabile, restituisce invece l'indirizzo di quest'ultima.

I puntatori sono strumenti necessari anche per la gestione delle *liste*, ovvero collezioni di un numero di elementi omogenei di cui non si conosce a priori la numerosità, la quale può anche variare nel tempo. Inoltre, al contrario di quanto accade per gli array, gli elementi non occupano posizioni contigue in memoria.

Una lista, in C, può essere vista come una struttura che contiene campi di informazioni e almeno un puntatore a cui viene legato l'elemento successivo.

Esempio 2.3: Definizione di una lista.

```
union nome_union{
    int numero;
    char carattere;
    struct nome_struct;
};
```

Il C permette di gestire l'allocazione della memoria in modo dinamico, assegnando alle variabili solamente la quantità di memoria necessaria.

Per fare ciò, esistono funzioni come `malloc` e `calloc`, adibite all'allocazione della memoria, `realloc`, per consentire la modifica di spazi di memoria precedentemente allocati, e `free` per liberare la memoria allocata.

Nel Capitolo 3, relativo allo sviluppo del software, verranno discussi alcuni esempi in cui verranno utilizzate tutte le funzionalità elencate finora.

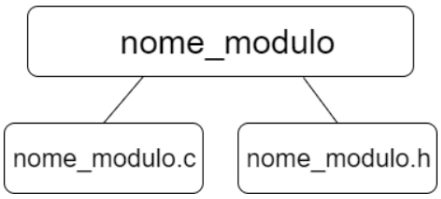

Figura 2: Un modulo è formato dal codice sorgente con il suo file di intestazione.

Nell'esposizione di questo elaborato, si ricorre spesso al termine *modulo*. Con questa espressione si fa riferimento alla coppia di file con estensioni `c` e `h`. Infatti, in C è buona norma suddividere il codice in questa maniera delegando al secondo file, che viene chiamato di *intestazione*, le dichiarazioni delle strutture dati e delle funzioni, lasciando ai file sorgente la loro definizione.

2.2 XML

Extensible Markup Language (XML) è un linguaggio di markup per documenti che contengono informazioni strutturate, quindi definite non solo dal contenuto, ma anche dal ruolo occupato da tale contenuto.

I linguaggi di markup sono insiemi di regole che definiscono dei meccanismi attraverso i quali è possibile identificare una struttura in un documento [8].

In XML, tali meccanismi prevedono l'utilizzo di *tag* di formattazione, cioè di una coppia di marcatori che delimitano una parte del contenuto del documento assegnandogli una semantica.

Applicato al nostro caso, ciò vale a dire che, per esempio, la struttura di un documento IEEE 1599 come presentata nell'introduzione di questo lavoro corrisponderebbe a:

```
<ieee1599>
        <general>...<\general>
        <logic>...<\logic>
        <structural>...<\structural>
        <notational>...<\notational>
```

```
        <performance>...<\performance>
        <audio>...<\audio>
</ieee1599>
```

I tag, o elementi, non sono l'unica componente utilizzata dai documenti XML. Le informazioni aggiuntive appartenenti agli elementi, ma che non fanno effettivamente parte del contenuto testuale, prendono il nome di *attributi*. Questi sono inseriti all'interno dei tag di apertura e hanno la forma `nome=valore`[1].

2.2.1 DTD

Un Document Type Definition (DTD) definisce la struttura di un documento XML, specificando quali elementi e attributi sono ammessi e come devono essere organizzati.

Un documento XML, per essere *valido*, deve rispettare le regole del DTD relativo oltre a dover essere *ben formato*, ovvero rispettante le regole di sintassi di XML.

Il DTD che definisce la struttura dei documenti IEEE 1599, sul quale si basano le scelte di programmazione effettuate per questo elaborato, è disponibile sul sito ufficiale del Computer Society of the Institute of Electrical and Electronics Engineers [2], attualmente alla versione del 2008.

Il DTD è dichiarato all'interno del documento XML nel tag vuoto `DOCTYPE`. Solitamente è definito in un file esterno e nei documenti viene espresso come riferimento a una risorsa esterna, come mostrato nel seguente estratto di un file IEEE 1599.

Esempio 2.4: Estratto del file gottes_macht.xml

```
<!DOCTYPE ieee1599 SYSTEM
"http://standards.ieee.org/downloads/1599/1599-2008/
ieee1599.dtd"
>
```

2.3 XPath

XPath è un linguaggio attraverso il quale è possibile indirizzare parti di un documento basato su tecnologie XML attraverso l'uso di espressioni [9].

[1] `http://www.cs.unibo.it/~fabio/corsi/einn00/11-XMLSintassi/11-XMLSintassi.pdf`
[2] `https://standards-ieee-org.pros.lib.unimi.it/downloads.html`

Queste vengono composte seguendo una sintassi che consente di accedere ai nodi della struttura ad albero modellata in base alla struttura logica del documento su cui si sta lavorando.

Tali espressioni vengono definite *Location Path* e sono composte a loro volta da *Location Step* separati da carattere, solitamente ‘‘/’’, la cui struttura è:

`axis::node-test[predicate]`

Ad ogni location step, viene effettuata una ricerca tra i nodi che fanno parte del *contesto* attuale. I nodi che soddisfano le condizioni di ricerca formano il nuovo contesto.

La componente *axis* si riferisce alla relazione di parentela tra i nodi cercati ed il nodo corrente; *node-test* specifica il tipo o il nome del nodo da cercare; *predicate* esprime una condizione da valutare rispetto alla ricerca effettuata[3].

Per il presente elaborato, come descritto in seguito, l'utilizzo di questo linguaggio è servito ad estrapolare per ogni layer del formato IEEE 1599 il relativo contesto.

2.4 Libxml2

Il software creato si appoggia sulla libreria libxml2 [4] per il parsing dei file XML generici.

Sul sito è presente un tutorial [5] su come utilizzare i metodi principali del linguaggio dal quale sono stati estrapolati alcuni algoritmi ricorrenti per le operazioni di parsing.

Questa libreria, tra le varie disponibili, è una delle più utilizzate perché può facilitare la programmazione essendo basata sulla creazione in memoria della struttura ad albero derivata dal documento XML, supporta XPath e permette la validazione tramite DTD durante il parsing.

[3] `https://www.html.it/pag/31760/xpath/`
[4] `http://www.xmlsoft.org/`
[5] `http://xmlsoft.org/tutorial/`

Capitolo 3

Il Software

In questo capitolo vengono descritti i principi su cui si basa il software e il metodo con cui è stato strutturato e organizzato il codice, iniziando a presentare alcune funzioni di utilità generale.

Viene discusso anche l'approccio con cui è stata affrontata la gestione della struttura a livelli del documento IEEE 1599, estrapolando uno schema comune a tutti i layer.

Infine vengono trattate le principali problematiche e questioni aperte riscontrate durante la progettazione.

3.1 Gestione del documento

3.1.1 Controllo degli input

Per prima cosa bisogna analizzare le caratteristiche del tipo di documento su cui bisogna lavorare. Il software deve ricevere in input un file XML e ci sarà quindi bisogno di controllare che il file sia ben formato, ma visto che si ha a disposizione un DTD, sarebbe ancora meglio lavorare su un file già validato.

La libreria libxml2 mette a disposizione il metodo `xmlValidateDTD` per verificare che un documento XML sia valido e su questo è stato sviluppata la funzione `isValid`, che restituisce il valore ‘‘1’’ in caso il documento sia valido e ‘‘0’’ altrimenti.

Questa funzione, insieme alle altre di ruolo simile e classificabile come gestione del file, sono state raggruppate nel modulo `common`.

Per consentire l'operazione di validazione è necessario disporre dei file DTD ieee1599.dtd[1], svg11.dtd[2], midixml.dtd[3] e MIDIEvents10.dtd[4].

Questi file vanno collocati al percorso indicato dalla variabile `dtd_root_folder`, di default è ‘‘./File/DTD’’ e parte dala directory di lavoro del software.

Sono disponibili anche i metodi `setDtdRootFolder` e `getDtdRootFolder` per modificare e recuperare il valore di tale variabile.

`setFileRootFolder` e `getFileRootFolder` servono invece a modificare e recuperare il valore della variabile `file_root_folder`, che di default vale ‘‘./File’’ e indica il percorso che può essere utilizzato per ricercare i documenti da elaborare.

3.1.2 Funzioni generali

Le prime operazioni basilari sviluppate sono state quelle per il caricamento del documento XML in memoria e per la navigazione del suo contenuto tramite XPath.

Il primo compito viene svolto dalla funzione `getDoc`, che riceve come input il percorso al documento IEEE 1599 da elaborare e il parametro `force_validation` di tipo intero.

Utilizza la funzione `xmlParseFile` di libxml2 per creare in memoria una struttura ad albero che rispecchia quella del documento XML e fornisce l'indirizzo del puntatore alla radice di tale struttura.

Quando `force_validation` è diverso da zero, `getDoc` carica in memoria solo i documenti validi. Altrimenti, carica anche quelli non validi avvisando l'utente su quali sono i problemi riscontrati.

La funzione `getNodeset`, partendo da un documento caricato in memoria, permette di estrapolare un contesto o nodeset a partire dall'espressione XPath fornitagli come input. Anche in questo caso, il codice si appoggia su una funzione di libxml2, nello specifico utilizza il metodo `xmlXPathEvalExpression`.

Le implementazioni di entrambe queste funzioni utilizzano algoritmi e funzioni di libreria ampiamente documentati sul sito libxml2, che sono stati adattati per le esigenze del caso.

In particolare sono stati utili gli esempi del tutorial per la lettura del contenuto degli elementi[5] e per l'utilizzo di XPath al fine di ottenere il contenuto degli elementi [6].

Queste funzioni fanno parte del modulo `common`.

[1] `https://www.lim.di.unimi.it/IEEE/ieee1599.dtd`
[2] `https://www.w3.org/Graphics/SVG/1.1/svgdtd.html`
[3] `https://www.musicxml.com/for-developers/musicxml-dtd/`
[4] `https://www.midi.org/dtds/MIDIEvents10.dtd.html`
[5] `http://xmlsoft.org/tutorial/ar01s04.html`
[6] `http://xmlsoft.org/tutorial/ar01s05.html`

Altre funzioni utili contenute in questo modulo sono quelle relative alla gestione delle stringhe, ovvero `concat`, che permette di concatenare due stringhe, `xmlCharToInt` e `xmlCharToDouble`. Le ultime due vengono utilizzate ogniqualvolta ci sia il bisogno di caricare nelle strutture dati valori intrinsecamente numerici contenuti nel documento XML.

Codice 3.1: Definizione di *Common attributes parameter entities* e *Common Elements* nel DTD.

```
<!-- Common attributes parameter entities -->

<!ENTITY % spine_ref
        "event_ref IDREF #REQUIRED">

<!ENTITY % spine_start_end_ref
        "start_event_ref IDREF #REQUIRED
    end_event_ref IDREF #REQUIRED">

<!ENTITY % accidental
        "(none | double_flat | flat_and_a_half | ...)">

<!ENTITY % formats
        "(application_excel | video_mp4 | ...)">


<!-- Common Elements -->

<!ELEMENT rights EMPTY>
<!ATTLIST rights
   file_name CDATA #REQUIRED>
```

Come mostra il Codice 3.1, nel DTD sono definiti quelli che vengono identificati come *Common attributes paramter entitis*, ovvero `accidental`, `format`, `spine_ref` e `spine_start_end_event_ref`.

I primi due sono utilizzati spesso perché servono a collegare allo spine gli elementi che hanno bisogno di essere sincronizzati tra loro. Queste entità servono solamente a indicare quali attributi fanno parte degli elementi in cui sono inserite.

Le altre, invece, elencano i possibili valori che l'attributo può assumere.

Nel modulo `common`, questi valori sono stati memorizzati nelle enumerazioni che ereditano lo stesso nome.

In *Common Elements* viene definito l'elemento `rights`, che viene utilizzato dai

layer notational, performance e audio e viene utilizzato per linkare a file esterni contenenti i diritti relativi al contenuto cui si riferiscono.

La struct `rights`, che lo rappresenta, e la funzione `loadRights`, per il suo caricamento, sono state inserite anch'esse in questo modulo.

3.2 Gestione dei layer

L'idea alla radice è quella di tradurre ogni elemento presente nella gerarchia XML in una struct le cui variabili interne sono gli attributi e le entità contenute, cercando ove possibile di mantenere le corrispondenze tra i nomi.

Per rendere il codice più leggibile, nella definizione delle struct si è deciso di ordinare le variabili al loro interno in questo modo:

- All'inizio sono elencate quelle accessorie, tipicamente si tratta di variabili contatore;
- Successivamente vengono collocate quelle derivate dagli attributi previsti dall'elemento da cui sono derivate;
- Seguono le istanze delle strutture, delle variabili o dei puntatori alle strutture dei sotto-elementi;
- Infine, ove necessario, viene dichiarato il puntatore della struttura a sé stessa.

Quest'ultimo elemento è presente nel caso in cui la struttura dovesse essere utilizzata per la creazione di una lista da parte di un elemento superiore della gerarchia.

Esempio 3.2: Ordinamento delle variabili contenute nella struct notehead.

```
struct notehead{
    int n_printed_accidentals;

    xmlChar* id;
    xmlChar* staff_ref;
    xmlChar* style;

    struct pitch pitch;
    struct fingering fingering;
    struct printed_accidental* printed_accidentals;
    xmlChar* printed_accidentals_shape;
    int tie;
```

```
    struct notehead* next_notehead;
};
```

In ogni caso, non tutti gli elementi citati nel DTD hanno subito lo stesso procedimento di traduzione: spesso è stato possibile sostituire un elemento con una lista di suoi figli. Per chiarire questo punto, l'Esempio 3.3 mostra come vengono tradotti gli elementi `general`, `related_files` e `related_file`. La struct che sarebbe dovuta derivare dall'elemento nel mezzo viene ridotta a una lista presente nell'elemento `general`, che lo precede nella gerarchia.

Esempio 3.3: Traduzione in struct dell'elemento `general`.

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

<!ELEMENT related_files (related_file+)>

<!ELEMENT related_file EMPTY>
<!ATTLIST related_file
        ...
        ...
>


struct general{
    int n_related_files;
    int n_analog_media;

    struct description description;
    struct related_file* related_files;
    struct analog_medium* analog_media;
    xmlChar* notes;
};
```

In generale, le struct impostate come liste sono state utilizzate anche per tutti quegli elementi che, secondo il DTD, possono comparire più volte come figli di un dato elemento, ovvero sono contrassegnati dai simboli ‘‘+’’ o ‘‘*’’.

Di seguito è mostrato il caso di `measure`, in cui `voice` deve comparire almeno una volta. Per quanto detto finora, nella struct derivata da questo elemento dovrà

essere presente una lista per contenere i valori delle varie occorrenze del suo sotto-elemento.

Codice 3.4: Definizione di `measure`.

```
<!ELEMENTO measure
    (voice+ | multiple_rest | measure_repeat?)
>
```

Per quanto riguarda i tipi di dato utilizzati all'interno delle struct, per la maggior parte sono state utilizzate stringhe, dato che la natura delle informazioni contenute nei documenti XML è prevalentemente testuale.

Le stringhe sono tutte gestite con il tipo `xmlChar` per mantenere omogeneità tra i tipi usati dal software e quelli usati dalla libreria libxml2 che lo definisce. Questo tipo di dato utilizza caratteri unsigned codificati in UTF-8.

In alcuni casi sono stati utilizzati altri tipi di dato più adatti ai casi specifici.

Per esempio sono stati adottati tipi numerici per quei valori che si prestano ad essere adoperati per svolgere operazioni aritmetiche: un esempio sono `num` e `den` dell'elemento `duration`, che rappresentano le frazioni indicanti la durata delle note, oppure il `timing` di `event` espresso in `VTU`.

In altri casi ancora, valori binari sono stati sostituiti con un intero che assume valori ``1'' e ``0'' per emulare un tipo di dato booleano.

Esempio 3.5: Utilizzo del tipo intero nella struttura duration

```
struct duration{
    int n_tuplet_ratios;

    int num;
    int den;
    struct tuplet_ratio* tuplet_ratio;
};
```

Analizzando i valori ammessi dal DTD per alcuni attributi,è stato possibile definire alcune enumerazioni utilizzabili per sviluppi futuri del software: per esempio possono rivelarsi utili per i controlli relativi all'input di certi valori, limitando la scelta ai soli valori enumerati se si decide di sviluppare funzioni che permettono di modificare le strutture caricate.

Esempio 3.6: Utilizzo di enum per enumerare i valori di `accidentals`.

```
enum accidentals{double_sharp,sharp_and_a_half,sharp,
```

```
demisharp , natural , demiflat , flat ,
flat_and_a_half , double_flat };
```

Lo scheletro del software è stato modellato suddividendo il codice in diversi moduli per rispecchiare la gerarchia ad albero definita dal DTD.

Tale gerarchia è stata ricostruita innestando le strutture partendo dalla radice del documento e arrivando fino all'ultimo elemento.

La struttura radice quindi, chiamata `ieee1599`, contiene quelle `general`, `logic`, `structural`, `notational`, `performance` e `audio`, come mostrato dal seguente schema, e a ciascuna di queste è associato il rispettivo modulo che si occuperà della gestione degli elementi sottostanti.

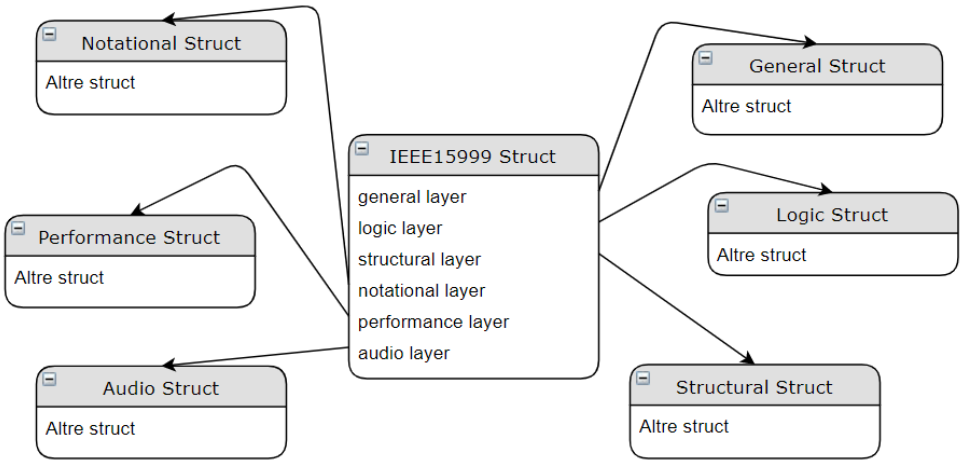

Figura 3: La gerarchia di struct rispecchia quella del formato IEEE 1599.

3.2.1 Algoritmo di caricamento

Per implementare la funzionalità di caricamento nelle strutture dati delle informazioni contenute nel documento, è stato utilizzato un algoritmo di partenza il cui principio di funzionamento rimane identico in tutte le funzioni di questo tipo pur venendo adattato per gestire le peculiarità di ciascun elemento.

La funzione `loadMeasureValue`, il cui codice è riportato in basso, fornisce un esempio concreto che permette di trattare gli aspetti principali necessari a illustrare il funzionamento di questo algoritmo.

Esempio 3.7: Definzione della funzione `loadMeasureValue`.

```
struct measure* loadMeasureValue(xmlNodePtr cur){
    xmlAttr* attributes;
    xmlNodePtr temp_cur;
    struct measure* value=(struct measure*)calloc(1,sizeof
(struct measure));
    value->next_measure=NULL;

    struct voice* voice_temp=NULL;
    struct voice* voice_head=NULL;
    struct voice* voice_p=NULL;
    value->n_voices= 0;

    value->next_measure=NULL;
    attributes=cur->properties;
    while(attributes!=NULL){
        if(!xmlStrcmp(attributes->name,
(const xmlChar*)"number")){
            value->number=xmlCharToInt(xmlGetProp(cur,
attributes->name));
        }
        else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"id")){
            value->id=xmlGetProp(cur,attributes->name);
        }
        else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"show_number")){
            if(!xmlStrcmp(xmlGetProp(cur,attributes->name)
,(const xmlChar*)"yes"))
                value->show_number= 1;
            else
                value->show_number= 0;
        }
        else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"numbering_style")){
            value->numbering_style=
xmlGetProp(cur,attributes->name);
        }
        attributes=attributes->next;
    }
```

```
    temp_cur=cur->xmlChildrenNode;
    while(temp_cur!=NULL){
        if(!xmlStrcmp(temp_cur->name,
(const xmlChar*)"voice")){
            voice_temp= (struct voice*)
calloc(1,sizeof(struct voice));
            voice_temp=loadVoiceValue(temp_cur);
            voice_temp->next_voice=NULL;
            if(voice_head==NULL){
                voice_head=voice_temp;
            }
            else{
                voice_p=voice_head;
                while(voice_p->next_voice!=NULL)
                    voice_p=voice_p->next_voice;
                voice_p->next_voice=voice_temp;
            }
            value->n_voices++;
        }
        else if(!xmlStrcmp(temp_cur->name,
(const xmlChar*)"multiple_rest")){
            attributes=temp_cur->properties;
            while(attributes!=NULL){
                if(!xmlStrcmp(attributes->name,
(const xmlChar*)"number_of_measures")){
                    value->
multiple_rest.number_of_measures=
xmlCharToInt(xmlGetProp(temp_cur,attributes->name));
                }
                else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"event_ref")){
                    value->multiple_rest.event_ref=
xmlGetProp(temp_cur,attributes->name);
                }
                attributes=attributes->next;
            }
        }
        else if(!xmlStrcmp(temp_cur->name,
(const xmlChar*)"measure_repeat")){
            attributes=temp_cur->properties;
```

```
                while(attributes!=NULL){
                    if(!xmlStrcmp(attributes->name,
(const xmlChar*)"number_of_measures")){
                        value->
measure_repeat.number_of_measures=
xmlCharToInt(xmlGetProp(temp_cur,attributes->name));
                    }
                    else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"event_ref")){
                        value->measure_repeat.event_ref=
xmlGetProp(temp_cur,attributes->name);
                    }
                    attributes=attributes->next;
                }
            }
            temp_cur=temp_cur->next;
        }
        value->voices=voice_head;

        return value;
}
```

In principio viene inizializzata la nuova variabile `value` che serve a contenere le informazioni caricate che la funzione deve ritornare al termine del procedimento.

Partendo dal nodo corrente puntato da `cur`, argomento passato dalla funzione chiamante, il primo ciclo while scandisce i suoi attributi, selezionando solo quelli possibili con le istruzioni `if/else` e salvandoli in `value` utilizzando la funzione `xmlGetProp`.

Al termine di questo ciclo viene eseguita l'istruzione

```
temp_cur=cur->xmlChildrenNode
```

che definisce un puntatore ai nodi figli dell'elemento puntato da `cur`.

Si entra in un altro ciclo while, che questa volta scandisce i tipi di elemento che `cur` può contenere.

Se sono previste occorrenze multiple di un sotto-elemento, come nel caso di `voice`, vengono preparati i tre puntatori per gestire la lista corrispondente all'inizio della funzione, in questo caso si tratta di `voice_temp`, `voice_head` e `voice_p`.

Quando viene intercettato un elemento `voice`, in `voice_temp` vengono caricati i valori di attributi e sotto-elementi chiamando la funzione `loadVoiceValue`, che

applica ricorsivamente il funzionamento dell'algoritmo in analisi. Successivamente vengono effettuate le istruzioni che lavorano sui puntatori `voice_head` e `voice_p` per costruire la lista di elementi `voice`.

Se la lista è vuota, ovvero `voice_head` non è definita, questa assume il valore di `voice_temp`. Altrimenti, se la lista non è vuota, si fa puntare `voice_p` all'ultimo elemento della lista e lo si trasforma in penultimo facendolo puntare a `voice_temp`. Quest'ultimo è sempre l'ultimo elemento della lista perché punta a `NULL`.

3.3 Problemi riscontrati

3.3.1 DTD esterni

Il formato IEEE 1599, per definire alcuni degli elementi che lo compongono, utilizza i DTD esterni di SVG e MIDI.

Codice 3.8: Dichiarazione di DTD esterni nel file *ieee1599.dtd*.

```
<!-- Import of external DTDs -->

<!ENTITY % svg SYSTEM "svg11.dtd" >
%svg;

<!ENTITY % ChannelRequired "#REQUIRED">

<!ENTITY % midixml SYSTEM "MIDIEvents10.dtd" >
%midixml;
```

SVG è l'acronimo di Scalable Vector Graphics ed è un formato XML utile a rappresentare immagini, oggetti grafici e interazioni grafiche [10].

Gli elementi in cui viene utilizzato questo tipo di rappresentazione sono:

- `custom_articulation`
- `slur`
- `custom_hsymbol`
- `layout_shapes`

Tutti questi elementi prevedono il sotto-elemento `svg`. Questo infatti serve a definire come l'oggetto in questione deve essere rappresentato graficamente.

Un esempio di quella che potrebbe essere una descrizione di questo tipo è dato dal seguente estratto di un documento IEEE1599.

Esempio 3.9: Utilizzo nella gerarchia di `horizontal_symbols` dell'elemento `svg`.

```
<horizontal_symbols>
    <custom_hsymbol
          start_event_ref="event_01"
          end_event_ref="event_02"
     >
        <svg width="100%" height="100%" version="1.1">
          <polyline
               points="0,0 0,20 20,20 20,40 40,40 40,60"
               style="fill:white;stroke:black;stroke-width:2"
          />
        </svg>
    </custom_hsymbol>
</horizontal_symbols>
```

In modo analogo, anche `music_font` e `text_font` devono essere definite dall'elemento `font`, anch'esso appartenente a SVG.

MIDI (Musical Instrument Digital Interface), invece, è un protocollo per lo scambio di informazione musicale e segnali di controllo, pensato per la comunicazione tra apparecchi musicali digitali.

Viene gestito dal livello performance in quanto i messaggi che è capace di veicolare servono a descrivere il procedimento necessario ad ottenere il suono [11].

Torna utile evidenziare che i messaggi MIDI vengono suddivisi in due categorie: *Channel Message* e *System Message*.

Infatti, per quanto riguarda il formato IEEE 1599, all'interno dell'elemento `midi_event` ci possono essere più occorrenze di quelli che il DTD chiama `MIDIChannelMessage`, che corrispondono alla prima categoria. Questi comprendono tutti quei messaggi che si riferiscono a canali MIDI specifici.

I *System Message* invece sono quelli che vengono inviati su tutti i canali e, tra questi, il formato IEEE 1599 gestisce i *System Exclusive* tramite l'elemento `SysEx`. Si tratta di messaggi definiti dalle case produttrici dei dispositivi stessi che servono alla loro comunicazione e possono avere significati diversi anche per apparecchi simili.

La versione attuale del presente software non si occupa della gestione di queste due tipologie di elementi esterni al DTD.

Se in futuro si volesse introdurre anche questa possibilità, si potrebbero includere una libreria per SVG e una per la codifica XML del MIDI se non si vuole implementare manualmente il parsing di tali elementi.

Ad ogni modo, anche senza ulteriori miglioramenti, il software consente ugualmente di caricare documenti che al loro interno contengono elementi provenienti

da DTD esterni. Se questo accade, questi saranno ignorati, ma gli elementi che li contengono vengono comunque gestiti.

3.3.2 Altri problemi

Altri problemi presentatisi durante l'implementazione sono stati riscontrati durante l'analisi del DTD del formato IEEE 1599.

Alla riga 767 viene introdotta l'entità parametrica `added_feature_object_classes`, che serve a conservare dati per poterli riutilizzare richiamandoli in modo sintetico attraverso il nome assegnatogli.

L'utilizzo di entità parametriche si rivela particolarmente utile, ad esempio, per richiamare i nomi di una lista di sotto-elementi comuni a elementi diversi senza doverli rielencare ad ogni evenienza [12].

Codice 3.10: Dichiarazione dell'entità parametrica `added_feature_object_class`.

```
<!ENTITY % added_feature_object_classes "">
```

Nel DTD, però, l'entità in questione è vuota e quindi attualmente non risulta di alcuna utilità. Per questo motivo, all'interno del codice, la sua presenza è stata ignorata.

Nel caso di future implementazioni, comunque, sarebbe utile ricordarsi di controllare le eventuali nuove versioni del DTD per verificare che questa componente non trovi effettiva applicazione, richiedendo, di conseguenza, un adattamento delle funzioni di parsing per gli elementi che la utilizzano.

Alla riga 970, invece, è presente quello che durante lo sviluppo di questo software è stato considerato come un refuso.

Codice 3.11: Dichiarazione dell'elemento `mpeg4_score`.

```
<!ELEMENT mpeg4_score (csound_spine_event+, rights?)>
```

L'elemento `mpeg4_score`, infatti, contiene il sotto-elemento `csound_spine_event`.

È ragionevole ritenere che questa sia una scorrettezza perché all'interno del DTD viene definito anche l'elemento `mpeg4_spine_event`, che non avrebbe altrimenti relazioni di discendenza con nessun altro elemento, pur essendo logicamente collegato e quindi riconducibile a sotto-elemento di `mpeg4_score`.

Capitolo 4

Implementazione dei moduli

In questo capitolo vengono descritti uno per uno i moduli in cui è suddiviso il software, soffermandosi sulle peculiarità e i punti di maggior difficoltà incontrati per ciascuno di essi, facendo riferimento a quello che viene definito nel DTD.

4.1 Radice del documento

L'elemento radice viene gestito dal modulo `managerDocument` e nel DTD viene definito come mostrato di seguito:

Codice 4.1: Definizione dell'elemento `ieee1599` secondo il DTD.

```
<!ELEMENT ieee1599
    (general, logic, structural?,notational?,
     performance?, audio?)
>
<!ATTLIST ieee1599
        version CDATA #REQUIRED
        creator CDATA #IMPLIED>
```

La struct adibita al suo caricamento è chiamata `ieee1599` ed è istanziata in una variabile globale chiamata `ieee1599_root`. Da questa è possibile accedere a qualsiasi altra informazione relativa al formato IEEE 1599 ed è ottenibile tramite il metodo `getIEEE1599Root`.

Codice 4.2: Definizione della struct `ieee1599`.

```
struct ieee1599{
    xmlChar* file_name;
```

```
    xmlChar* version;
    xmlChar* creator;

    struct general general_layer;
    struct logic logic_layer;
    struct structural structural_layer;
    struct notational notational_layer;
    struct performance performance_layer;
    struct audio audio_layer;
};
```

Tra le variabili che compongono la struct radice è compresa `file_name`. Questa non è stata dedotta dal DTD, ma è una stringa accessoria che serve a recuperare il nome del documento a cui i dati caricati in memoria si riferiscono.

Le funzioni principali appartenenti a questo modulo sono `loadDocument`, `freeDocument` e `printDocument`. La prima si occupa del caricamento dei dati, la seconda di liberare la memoria allocata dinamicamente per contenere le strutture e la terza per la stampa a video di alcune delle informazioni sul documento caricato in memoria. Tutte contengono i richiami alle analoghe funzioni appartenenti ai vari layer.

La funzione `loadDocument`, in più, introduce un algoritmo per il caricamento degli attributi, in questo caso `version` e `creator`, il cui principio rimane simile a quelli utilizzati da tutte le altre funzioni di caricamento.

Codice 4.3: Estratto di codice della funzione `loadDocument`.

```
...

xpath=(xmlChar *)"/ieee1599";
result=getNodeset(doc,xpath);

if(!xmlXPathNodeSetIsEmpty(result->nodesetval)){
    nodeset=result->nodesetval;
    cur=nodeset->nodeTab[0];
    attributes=cur->properties;
    while(attributes!=NULL){
     if(!xmlStrcmp(attributes->name,
(const xmlChar*)"version"))
     {
       ieee1599_root.version=
```

```
xmlGetProp(cur, attributes->name);
      }
      else if(!xmlStrcmp(attributes->name,
(const xmlChar*)"creator"))
      {
        ieee1599_root.creator=
xmlGetProp(cur, attributes->name);
      }
      attributes=attributes->next;
    }
}
...
ieee1599_root.notational_layer=loadNotational();
...
```

L'estratto di codice di cui sopra mostra come, dopo aver passato l'espressione XPath a `getNodeset` per iniziare a percorrere il documento dalla radice, il contesto contenuto nella variabile `nodeset`, che è un puntatore a una gerarchia di nodi, viene scandito nel ciclo `while` per estrarre il valore degli attributi di ciascun nodo.

Attributi diversi vengono isolati tramite il costrutto `if/else`, comparando i nomi dei tag grazie a `xmlStrcmp` e leggendone il valore con `xmlGetProp`, entrambe funzioni di libxml2.

La funzione `getNodeset` può ritornare più nodi che soddisfino l'espressione indicata come argomento e che sono puntati dagli elementi dell'array `nodeTab`.

Se ci fossero più nodi chiamati ‘‘ieee1599’’, l'algoritmo lavorerebbe soltanto sul primo nodo incontrato.

Lo stesso codice mostra infine, prendendo `notational` come esempio, come vengono assegnati i valori alle struct interne alla variabile `ieee1599_root` associate ai layer.

Le funzioni di stampa che vengono chiamate da `printDocument` utilizzano la variabile globale di tipo intero `N_DISPLAY`: il suo valore indica, per ciascuna lista presente nell'intera gerarchia delle struct, quanti elementi possono essere stampati al massimo.

4.2 General layer

I livello general è gestito nel modulo `managerGeneral` e nel DTD viene rappresentato dall'elemento `general`, la cui definizione è mostrata in basso.

Codice 4.4: Definizione dell'elemento `general` secondo il DTD.

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

Come anticipato nel Capitolo 3.2, in cui è stato presentato l'esempio riguardante `related_files`, possiamo notare come anche `analog_media` è stato sostituito con una lista dei suoi figli `analog_medium`. Infatti, dal Codice 4.5 si può osservare che la sua funzione è solamente quella di contenitore di uno o più elementi, come indicato dal simbolo ''+''.

Codice 4.5: Definizione di `analog_media` e `analog_medium` secondo il DTD.

```
<!ELEMENT analog_media (analog_medium+)>

<!ELEMENT analog_medium EMPTY>
<!ATTLIST analog_medium
        description CDATA #REQUIRED
        copyright CDATA #IMPLIED
        notes CDATA #IMPLIED>
```

La struct `general` contiene anche i due contatori `n_related_files` e `n_analog_media` per tenere il conto del numero di oggetti caricati nelle rispettive liste associate.

Codice 4.6: Dichiarazione della struct `general`.

```
struct general{
    int n_related_files;
    int n_analog_media;

    struct description description;
    struct related_file* related_files;
    struct analog_medium* analog_media;
    xmlChar* notes;
};
```

L'elemento `notes`. invece, dovendo contenere solamente del teso, viene tradotto in una stringa.

L'elemento `description` è il primo caso incontrato in cui sono previsti altri sotto-elementi che necessitano di una traduzione in struct, tra cui `main_title`, `authors` e `other_title` che possono comparire più di una volta.

In questi casi, il processo di traduzione visto finora continua ad applicarsi ricorsivamente, fino a raggiungere gli ultimi livelli di profondità nella gerarchia.

Codice 4.7: Dichiarazione della struct `description`.

```
struct description{
  int n_authors;
  int n_other_titles;
  int n_dates;
  int n_genres;

  xmlChar* main_title;
  xmlChar* number;
  xmlChar* work_title;
  xmlChar* work_number;
  struct author* authors;
  struct other_title* other_titles;
  struct date* dates;
  struct genre* genres;
};
```

Il contenuto di questo intero livello viene istanziato nella variabile statica `general_layer` accessibile da ogni metodo del modulo.

```
static struct general general_layer;
```

Per quanto riguarda le funzioni, per ogni sotto-elemento di `general` sono state sviluppate quelle per il caricamento dei dati nelle strutture, per liberare la memoria allocata dinamicamente e per la stampa a video.

Quelle del primo tipo utilizzano espressioni XPath per ottenere il contesto relativo all'elemento di cui si occupano e, scansionando i nodi ottenuti, estrapolano i dati da assegnare ai campi adatti della variabile `general_layer`.

Il contenuto di quest'ultima viene ritornato dalla funzione `loadGeneral`.

4.3 Logic layer

Il livello logic è gestito dal modulo `managerLogic` e nel DTD viene rappresentato dall'elemento `logic`.

Codice 4.8: Definizione dell'elemento `logic` secondo il DTD.

```
<!ELEMENT logic (spine, los?, layout?)>
```

Data la mole di informazioni previste dal contenuto di questo layer, e in particolare della sua componente LOS (Logically Organized Symbols), è stato deciso di suddividere ulteriormente il codice per gestire in modo più ordinato alcuni degli elementi di quest'ultima nel sotto-modulo `managerLosElements`.

Codice 4.9: Dichiarazione della struct `logic`.

```
struct logic{
    int n_events;

    struct event* spine;
    struct los los;
    struct layout layout;
};
```

L'elemento `spine` è uno dei più importanti del formato: consiste in una lista di `event` che permette di sincronizzare tutti gli eventi temporali descritti anche dagli altri layer.

Nella libreria, questa componente è stata tradotta in una lista chiamata `spine` che ha elementi di tipo struct `event`.

Inoltre, i suoi attributi sono un esempio di come è stato utilizzato il tipo intero per memorizzare alcuni valori. Si tratta di `hpos` e `timing`, che indicano rispettivamente un'unità di misura della posizione orizzontale, espressa in Virtual Pixels (VPX), e una coordinata temporale, espressa in Virtual Time Unit (VTU).

Tra gli elementi che compongono `los`, invece, l'elemento `agogics` è un caso in cui il tipo intero viene impiegato per simulare il funzionamento delle variabili booleane.

La traduzione dell'elemento `staff_list`, che contiene soltanto due sotto-elementi, è stata effettuata sostituendolo con altrettante liste: una per `brackets` e una per `staff`.

Anche il suo attributo `bracketed`, che secondo il DTD può assumere i valori ‘‘yes’’o ‘‘no’’, viene tradotto in un intero.

Quindi, in fase di parsing, alla variabile viene assegnato il valore ‘‘1’’ nel primo caso o ‘‘0’’ altrimenti.

Per quanto riguarda la componente `layout` non emergono particolari differenze rispetto a quanto affrontato finora, se non per la presenza degli elementi esterni di cui si è discusso nel Capitolo 3.3.1.

Per caricare i dati all'interno di tutte le strutture di questo layer, la funzione `loadLogic` richiama le altre funzioni di loading più interne e, al termine delle operazioni di parsing, ritorna la variabile `logic_layer` popolata.

4.3.1 LOS sublayer

Il modulo `managerLosElements` gestisce gli elementi di `staff`, `part`, `horizontal_symbols` e `ornaments`.

Codice 4.10: Definizione dell'elemento `los` nel DTD.

```
<!ELEMENT los
        (agogics*, text_field*, metronomic_indication*,
        staff_list, part+, horizontal_symbols?,
        ornaments?, lyrics*)
>
```

Le strutture che plasmano le liste degli elementi `horizontal_symbols` e `ornaments` sono gestite in modo leggermente diverso da come visto fin ora.

Per cominciare, al nome delle struct ereditato da quello di corrispondenti elementi del DTD viene aggiunto il suffisso `_list`.

Inoltre, queste contengono un elemento di tipo `horizontal_symbol` o `ornament` in aggiunta al solito puntatore.

Tali tipi sono stati definiti in questo modulo combinando l'uso di `typedef` e `union`.

Nell'estratto di Codice 4.11 viene mostrato, prendendo `turn` da esempio, come ciascun sotto-elemento di `ornaments` viene tradotto nella corrispondente struttura per poi essere racchiuso in un'unica union, dalla quale viene creato il tipo `ornament`.

Codice 4.11: Definizione di `ornaments` nel DTD, dichiarazione della struct `turn` e definizione del tipo `ornament`.

```
<!ELEMENT ornaments
  (acciaccatura | baroque_acciaccatura | appoggiatura |
  baroque_appoggiatura | mordent | tremolo | trill | turn)*
>
```

```
struct turn{
    xmlChar* ornament_name;
```

```
    xmlChar* id;
    xmlChar* event_ref;
    xmlChar* type;
    xmlChar* style;
    xmlChar* upper_accidental;
    xmlChar* lower_accidental;
};

typedef union{
    struct acciaccatura acciaccatura;
    struct baroque_acciaccatura baroque_acciaccatura;
    struct appoggiatura appoggiatura;
    struct baroque_appoggiatura baroque_appoggiatura;
    struct mordent mordent;
    struct tremolo tremolo;
    struct trill trill;
    struct turn turn;
    int empty;
}ornament;
```

In questo layer sono presenti gli elementi vuoti `repetition` e `tie`.

Questi, anziché essere tradotti in struct, sono stati trasformati in valori interi che assumono ‘‘1’’ o ‘‘0’’ per segnalare rispettivamente la loro presenza o assenza.

L'elemento `key_signature` dovrebbe contenere i sotto-elementi `sharp_num` o `flat_num`, anch'essi elementi vuoti con l'attributo `number` in comune. È stato deciso di tradurli in un'unica struttura con l'aggiunta della variabile `num_type` che indica a quale dei due elementi originali ci si riferisce.

Codice 4.12: Definzione dell'elemento `articulation` e dei suoi componenti secondo il DTD.

```
<!ELEMENT articulation
    (normal_accent | staccatissimo | ... )*
>

<!ELEMENT normal_accent EMPTY>

<!ELEMENT staccatissimo EMPTY>
```

```
struct articulation{
    xmlChar* articulation_sign;

    struct articulation* next_articulation;
}
```

Come mostrato dal Codice 4.12, l'elemento `articulation` appartenente a `chord`, è un contenitore di diversi tipi di elementi vuoti e senza attributi.

Ad ogni `chord` corrisponde un `articulation` che può contenere più oggetti.

Si è deciso di organizzare tutto ciò facendo in modo che nella struct di `chord` compaia una lista di `articulation` nella cui struttura è presente la stringa `articulation_sign` che serve a indicare, tramite il nome dell'elemento, il tipo di oggetto cui si riferisce.

I valori possibili per `articulation_sign` sono elencati nell'enumerazione `articulation_signs` definita nel modulo `common`.

Codice 4.13: Definzione dell'elemento `notehead` e dei suoi componenti secondo il DTD.

```
struct notehead{
    int n_printed_accidentals;

    xmlChar* id;
    xmlChar* staff_ref;
    xmlChar* style;

    struct pitch pitch;
    struct fingering fingering;
    struct printed_accidental* printed_accidentals;
    xmlChar* printed_accidentals_shape;
    int tie;

    struct notehead* next_notehead;
};
```

In modo simile viene gestito anche l'elemento `printed_accidentals`.

Questo però prevede l'attributo `shape` che, nel processo di traduzione, viene ereditato come variabile dall'elemento padre `notehead` nella variabile `printed_accidentals_shape`, mostrata nel Codice 4.13.

È possibile fare questa operazione senza generare ambiguità per il fatto che `notehead` può contenere al massimo un elemento `printed_accidentals`.

Un' altra differenza rispetto a `articulation` è che i suoi sotto-elementi non sono privi di attributi. Al contrario, questi hanno `parenthesis` in comune, che viene tradotto in variabile intera nella struttura associata.

Anche in questo caso, il tipo di oggetto cui la struttura si riferisce, come nel caso precedente, è contenuto in una stringa, `printed_accidental_type` , che assume come valore il nome del rispettivo sotto-elemento.

I possibili nomi di tali sotto-elementi sono enumerati nella variabile `accidentals` definita nel modulo `common`.

Questo sotto-modulo non definisce una variabile per contenere i dati caricati. Quindi, tutte le funzioni di caricamento esistono per essere chiamate dal modulo del layer logic, al quale ritornano i contenuti di ciascuna istanza delle struct.

4.4 Structural layer

Il livello structural è gestito dal modulo `managerStructural` e nel DTD viene rappresentato dall'elemento `structural`.

Codice 4.14: Definizione dell'elemento `structural` secondo il DTD.

```
<!ELEMENT structural
    (chord_grid*, analysis*, petri_nets*, mir*)
>
```

La funzione di caricamento `loadGeneral` ritorna il valore della variabile `structural_layer`, istanza della struct `structural` presente nel Codice 4.15 che rappresenta il layer.

Codice 4.15: Dichiarazione della struct `structural`.

```
struct structural{
    int n_chord_grids;
    int n_analysis;
    int n_petri_nets;
    int n_mir_models;

    struct chord_grid* chord_grid;
    struct analysis* analysis;
    struct petri_net* petri_nets;
    struct mir_model* mir;
};
```

Questa struttura contiene le quattro liste che servono a contenere gli altrettanti tipi di sotto-elemento.

Tra di essi è presente `analysis` nella cui gerarchia è previsto `feature_object`, che coinvolge l'entità parametrica `added_feature_object_classes` di cui si è discusso nel Capitolo 3.3.

4.5 Notational layer

Il livello notational è gestito dal modulo `managerNotational` e nel DTD viene rappresentato dall'elemento `notational`.

Codice 4.16: Definizione dell'elemento `notational` secondo il DTD.

```
<!ELEMENT notational
    (graphic_instance_group | notation_instance_group)+
>
```

La funzione `loadNotational` ritorna la variabile `notational_layer` dopo aver caricato al suo interno le liste dei due possibili tipi di sotto-elemento previsti da questo layer, che sono `graphic_instance_group` e `notation_instance_group`.

Nel modo in cui si articolano le gerarchie di questi ultimi due, si può individuare una struttura simile. In Figura 4 viene illustrata questa caratteristica, evidenziata anche dalla simmetria dello schema.

Ogni parte deve però essere tradotta in una strutture dati diversa perché vengono definiti attributi differenti per ognuna, tranne per il caso dell'elemento in comune `rights`.

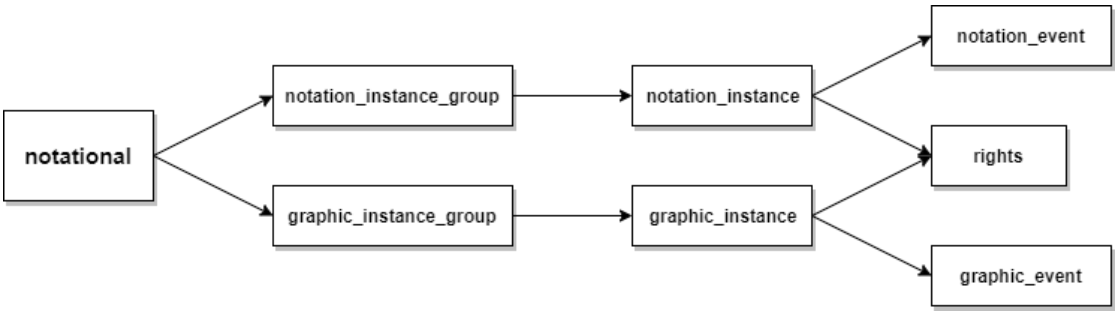

Figura 4: Gerarchia interna all'elemento `notational`.

4.6 Performance layer

Il livello performance è gestito dal modulo `managerPerformance` e nel DTD viene rappresentato dall'elemento `performance`.

Codice 4.17: Definizione dell'elemento `performance` secondo il DTD.

```
<!ELEMENT performance
        (midi_instance | csound_instance | mpeg4_instance)+
>
```

In modo simile al layer notational, è possibile individuare una strutturazione comune per gli elementi della gerarchia di `csound_instance` e `mpeg4_instance`.

La differenza sta nel fatto che, in questo caso, gli elementi simmetrici condividono anche gli stessi attributi, fatta eccezione per `instrument_number` di `csound_instrument_mapping` che diventa `instrument_name` in `mpeg4_instrument_mapping`.

Per evitare ripetizioni di codice è stata creata una sola struttura per ogni coppia di elementi abbinabili. I nomi delle struct sono stati adattati come mostrato nello schema in basso. Quelli delle variabili interne rimangono identici a quelli degli attributi relativi, tranne per la stringa `instrument_info` che ospita i valori di `instrument_number` o `instrument_name` a seconda dei casi.

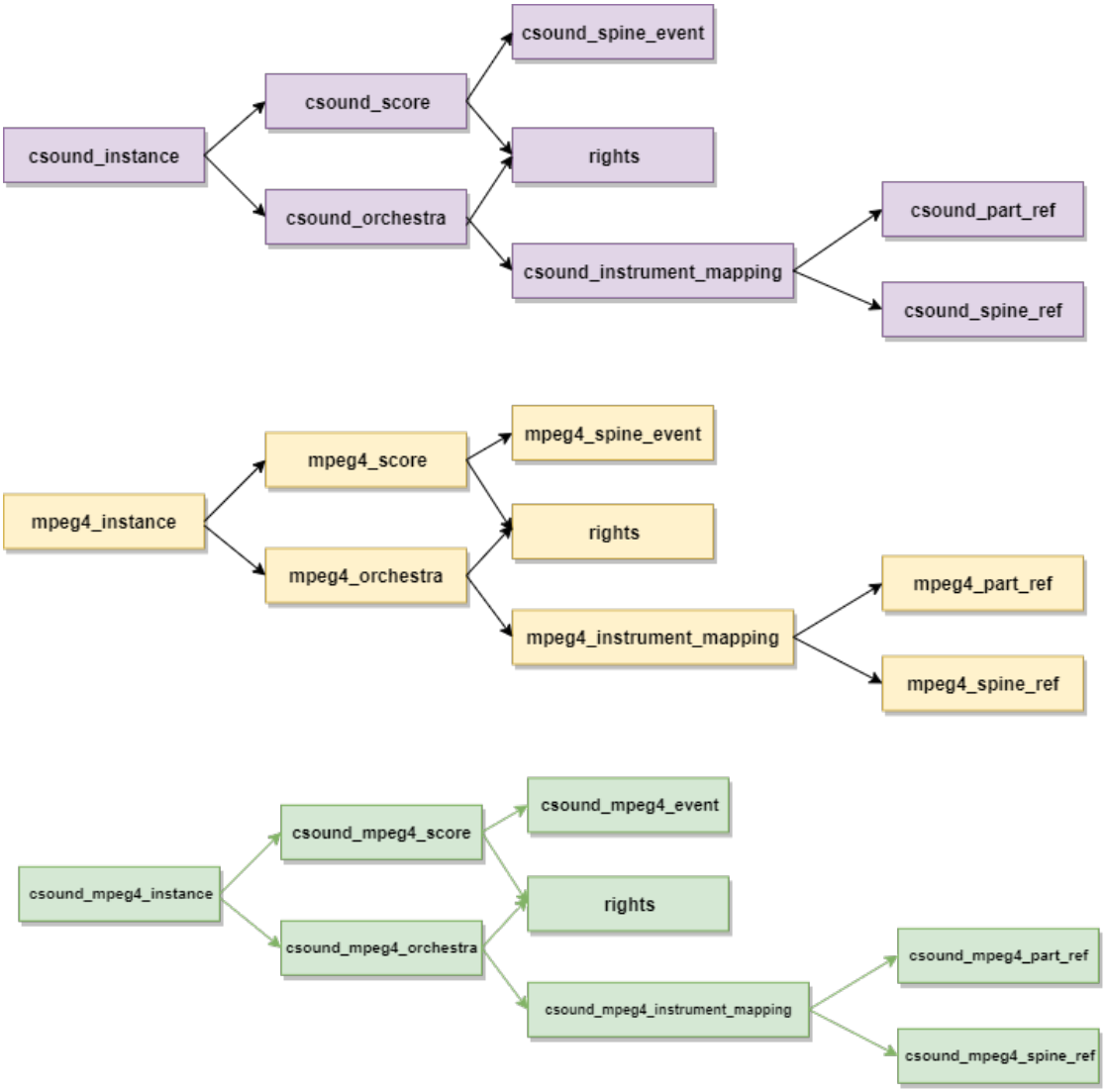

Figura 5: Gerarchie interne agli elementi `csound_instance` e `mpeg4_instance` fuse in quella di `csound_mpeg4_instance`.

Nella struct `performance`, la cui istanza `performance_layer` che contiene i dati associati a questo layer, sono dichiarate tre liste, tante quanti sono i sotto-elementi di `performance` nominati dal DTD, Una contiene le istanze di `midi_instance`, le altre due sono istanze di `csound_mpeg4_instance`, una chiamata `csound_instance` e l'altra `mpeg4_instance`.

La funzione adibita al caricamento dei dati dell'intero layer è `loadPerformance`, che a sua volta richiama `loadMidiInstance`,

`loadCsoundInstance` e `loadMpeg4Instance` per il parsing degli elementi più in profondità.

4.7 Audio layer

Il livello audio è gestito dal modulo `managerAudio` e nel DTD viene rappresentato dall'elemento `audio`.

Codice 4.18: Definizione degli elementi `audio` e `track` secondo il DTD.

```
<!ELEMENT audio (track+)>

<!ELEMENT track (track_general?, track_indexing?, rights?)>
<!ATTLIST track
        file_name CDATA #REQUIRED
        file_format %formats; #REQUIRED
        encoding_format %formats; #REQUIRED
        md5 CDATA #IMPLIED>
```

L'elemento `audio` è un contenitore di `track`, componenti definite da quelli che il DTD chiama `General Sub-layer` e `Indexing Sub-layer`, rappresentati rispettivamente dagli elementi `track_general` e `track_indexing`.

La funzione `loadAudio` è adibita al popolamento della variabile `audio_layer`, istanza della struct `audio`.

Quest'ultima è la struttura dati che rappresenta l'intero livello ed è costituita solamente da una lista di `track` ed un contatore che ne segna il numero di occorrenze.

`track_general` contiene `genres`, già incontrato nel layer general. Anche se questo elemento non è stato indicato tra i *Common Elements* del DTD, si tratta comunque della stessa componente utilizzata in più occasioni.

Per questo motivo, sia la dichiarazione della sua struct che la sua funzione di caricamento sono state inserite nel modulo `common`.

Capitolo 5

Testing

Per verificare il corretto funzionamento di quanto sviluppato, si è deciso di implementare un semplice programma in C che utilizzi le funzioni rese disponibili dall'interfaccia della libreria.

A questo scopo, il codice sorgente dei moduli implementati è stato compilato in una libreria composta dalla coppia di file `lib` e `dll` insieme ai vari header.

I passi da eseguire per potere ottenere tali file in Windows utilizzando Visual Studio 2019 sono elencati nel file *Tutorial Compilazione Libreria* conservato nel repository *GitHub* in cui si trova l'intero progetto [1], dove è presente anche il file *Tutorial Utilizzo Libreria*: una guida per la configurazione di Visual Studio 2019 per predisporre all'utilizzo della libreria.

È stato utilizzato anche un file batch da eseguire ogni volta che viene compilato il sorgente della libreria che organizza automaticamente in modo convenzionale i file necessari al suo utilizzo da terze parti.

La libreria viene in questo modo confezionata in un package la cui struttura è mostrata dalla seguente Figura 6 .

[1] `https://github.com/LIMUNIMI/Manager1599`

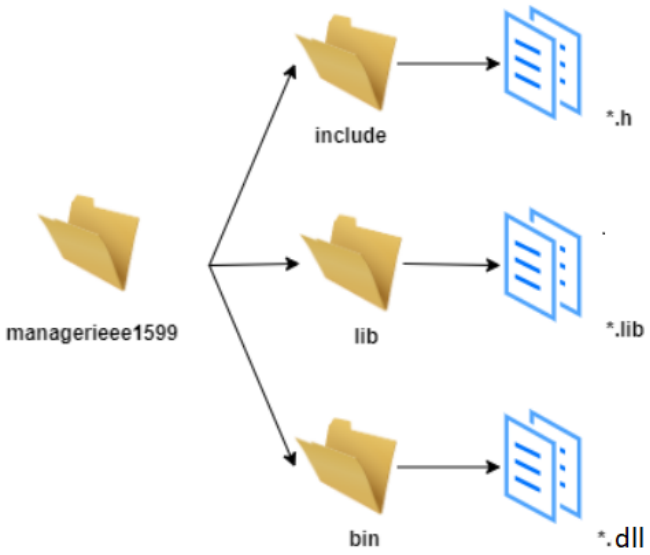

Figura 6: Organizzazione dei file che compongono la libreria Manager IEEE1599.

Per poter accedere alle principali funzioni di caricamento e lettura dei dati, il progetto client che la utilizza deve importare l'header "managerDocument.h".

Per comodità, è stato sviluppato il modulo `fileChooser`, il cui scopo è quello di introdurre delle procedure per potere consentire all'utente la scelta di un documento da caricare.

In particolare comprende il metodo `showFiles`, che elenca tutti i file XML presenti in una data cartella associando a ciascuno un numero incrementale. Per memorizzare il nome del file scelto, si passa come input il suo numero a `readFileName`.

Queste ultime funzioni utilizzano strutture dati esterne che sono state importate dalla libreria `dirent`.

I file XML presi in considerazione da questo modulo sono quelli che si trovano nella cartella indicata da `file_root_folder`. Al suo interno, sono stati depositati tutti i file IEEE 1599 scaricabili dall'archivio musicale del sito del LIM [2].

Il main del software sviluppato consiste in un semplice ciclo `while` che ad ogni iterazione mostra all'utente i file XML disponibili e richiede la scelta di uno di essi tramite l'inserimento del relativo numero associato. Dopodiché, il documento viene elaborato e l'utente può scegliere se terminare l'esecuzione o elaborarne uno nuovo.

Per caricare l'interno documento, viene chiamato il metodo `loadDocument` della libreria.

[2] `https://ieee1599.lim.di.unimi.it/music_collection.php`

I dati caricati in memoria vengono stampati a video dalla funzione `printDocument` avendo impostato `N_DISPLAY` a ‘‘3’’.

Per ultimo, viene chiamato il metodo `freeDocument` per permettere di liberare la memoria allocata dinamicamente.

Analizzando il contenuto dei file utilizzati per il testing, si può osservare che alcuni elementi previsti dal DTD e addirittura intere componenti di alcuni layer non vengono mai utilizzati. Risulta perciò impossibile riferirsi a un singolo caso pratico che permetta di verificare il corretto funzionamento della libreria per ogni singola entità prevista dal DTD.

Si può comunque assumere che, se il processo funziona correttamente per gli elementi più ricorrenti, funzioni anche per quelli inutilizzati, poiché la loro gestione si fonda per la maggior parte delle volte su algoritmi che seguono lo stesso ragionamento e implementazione simile.

Se si vuole dimostrare che ogni tipo di informazione viene caricata correttamente, bisogna comprare ogni volta l'output del programma client con il contenuto del file in elaborazione.

Nel prosieguo di questo capitolo, a sostegno di questa tesi, vengono presentati alcuni esempi di stampa a video presi a campione tra le elaborazioni di documenti differenti, scegliendo quelli che presentano contenuti significativi, in modo da coprire, per quanto possibile, tutti i vari aspetti di cui si occupa il formato IEEE 1599.

5.1 Caricamento dei file

Dalla Figura 7 si può osservare la lista di file XML indicizzata mostrata all'utente per la scelta del documento da elaborare.

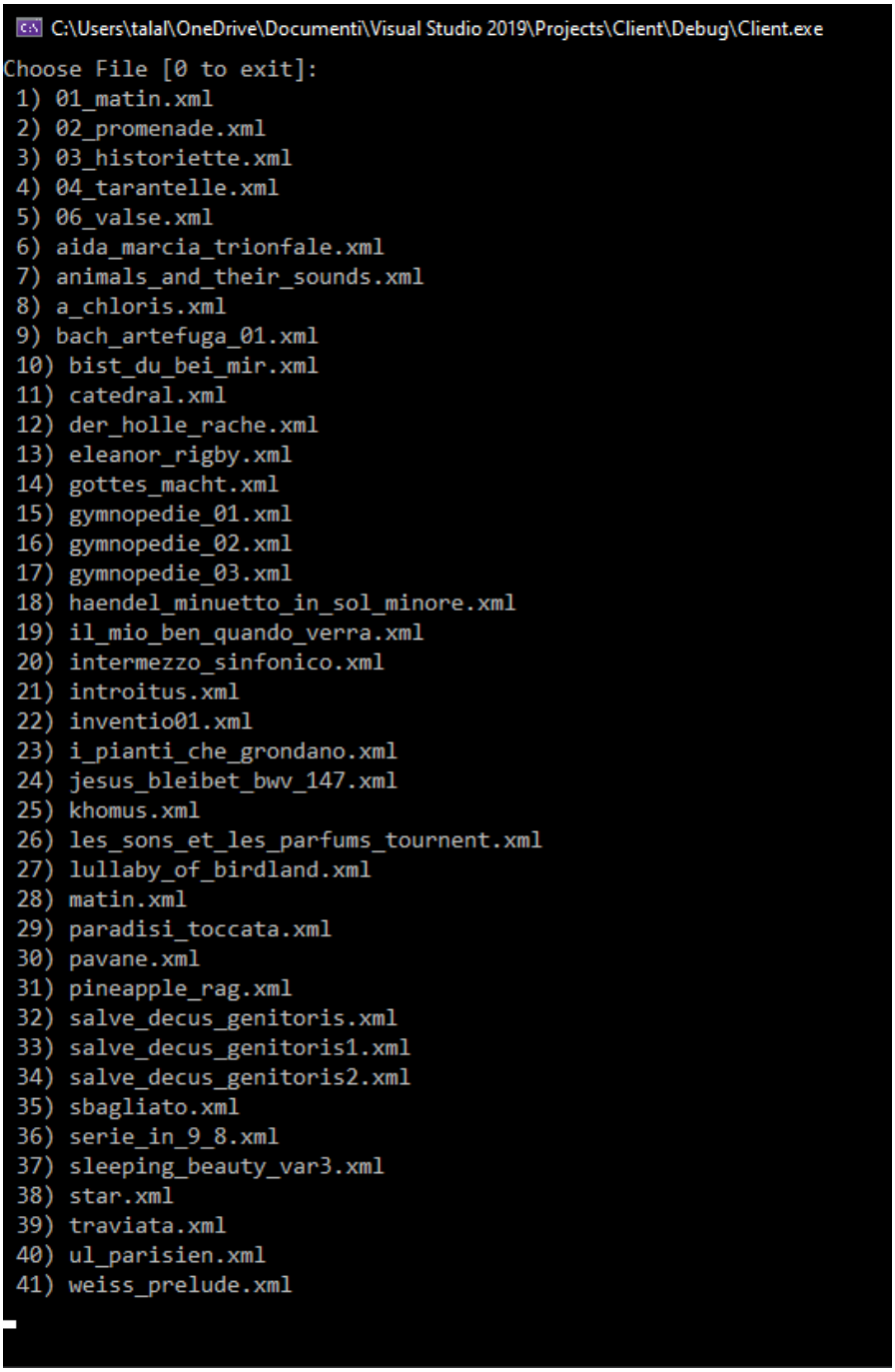

Figura 7: Esempio di lista di documenti XML contenuti nella cartella indicata dalla variabile `file_root_folder`.

Dopo aver scelto un file e averlo caricato in memoria, vengono eseguite le

istruzioni XPath per iniziare a esplorare la struttura XML parendo dai nodi più esterni.

Il tipo di messaggi mostrato qui di seguito è utile soprattutto per lo sviluppatore che, nel caso si verifichino errori durante il parsing, può dedurre in quale dei layer si verifica il problema:

```
Loaded ./File/aida_marcia_trionfale.xml
Executed xpath:/ieee1599
Executed xpath:/ieee1599/general/description
Executed xpath:/ieee1599/general/related_files/related_file
Executed xpath:/ieee1599/general/analog_media/analog_medium
Executed xpath:/ieee1599/general/notes
Executed xpath:/ieee1599/logic/spine
Executed xpath:/ieee1599/logic/los
Executed xpath:/ieee1599/logic/layout
Executed xpath:/ieee1599/structural/chord_grid
Executed xpath:/ieee1599/structural/analysis
Executed xpath:/ieee1599/structural/petri_nets/petri_net
Executed xpath:/ieee1599/structural/mir/mir_model
Executed xpath:/ieee1599/notational/graphic_instance_group
Executed xpath:/ieee1599/notational/notation_instance_group
Executed xpath:/ieee1599/performance/midi_instance
Executed xpath:/ieee1599/performance/csound_instance
Executed xpath:/ieee1599/performance/mpeg4_instance
Executed xpath:/ieee1599/audio/track
```

5.2 Output general layer

Per controllare i risultati relativi al layer general è stato scelto il file *aida_marcia_trionfale* perché, tra i documenti analizzati, è fra quelli che contengono una descrizione tra le più complete per questo livello.

L'output qui riportato permette di affermare che tutte le informazioni riguardanti `description` e `related_files` vengono immagazzinate correttamente:

```
###Document Info###
File name: ./File/aida_marcia_trionfale.xml
[version=1.0]

###General Layer###
```

```
#Description#
Title: Marcia trionfale
Work Title: Aida
2 authors
composer: Giuseppe Verdi
librettist: Antonio Ghislanzoni

#Related Files#
15 related files
File: name=Aida_Marcia_trionfale/iconografico/bozzetto1.jpg
    format=image_jpeg encoding=image_jpeg description=Bozzetto:
        Palazzo del Re
File: name=Aida_Marcia_trionfale/iconografico/bozzetto2.jpg
    format=image_jpeg encoding=image_jpeg description=Bozzetto:
        Tempio di Vulcano
File: name=Aida_Marcia_trionfale/iconografico/bozzetto3.jpg
    format=image_jpeg encoding=image_jpeg description=Bozzetto:
        Appartamento di Amneris
...
```

5.3 Output logic layer

Le componenti del layer logic sono state osservate separatamente.

Lo spine è stato estratto dal file *animals_and_their_sounds* e già dai primi elementi stampati a video è possibile affermare che tutte le informazioni vengono memorizzate correttamente:

```
###Logic Layer###

#Spine#
8 events
Event: id=event_cow
Event: id=event_pig timing=1 hpos=1
Event: id=event_dog timing=1 hpos=1
...
```

Tra tutti gli elementi che riguardano il los, i più utilizzati sono sicuramente quelli che fanno parte di `part` e `staff_list` ,che sono infatti presenti in tutti i documenti presi in considerazione.

Il file *gottes_macht*, utilizzato come prossimo esempio, rappresenta il caso più esauriente poiché contiene, in aggiunta, informazioni che riguardano gli elementi `agogics`, `horizontal_symbols` e `lyrics`:

```
#LOS#
1 agogics
Agogics: event_ref=timesig_1 value=Mit Kraft und Feuer.

3 staves
Staff: id=staff_1
    1 clefs
    Clef: ( shape=G step=2 event_ref=clef_1 )
    1 key signatures
    Key Signature: ( event_ref=keysig_1 num_type=flat_num )
    1 time signatures
    Time Signature: ( event_ref=timesig_1 visible=yes )
        Time Indications: ( num=2 den=2 vtu=8 )
Staff: id=staff_2
    1 clefs
    Clef: ( shape=G step=2 event_ref=clef_2 )
    1 key signatures
    Key Signature: ( event_ref=keysig_2 num_type=flat_num )
    1 time signatures
    Time Signature: ( event_ref=timesig_2 visible=yes )
        Time Indications: ( num=2 den=2 vtu=8 )
Staff: id=staff_3
    1 clefs
    Clef: ( shape=F step=6 event_ref=clef_3 )
    1 key signatures
    Key Signature: ( event_ref=keysig_3 num_type=flat_num )
    1 time signatures
    Time Signature: ( event_ref=timesig_3 visible=yes )
        Time Indications: ( num=2 den=2 vtu=8 )

2 parts
Part: id=soprano
    1 voice items
    Voice List: ( id=soprano_voice1 staff_ref=staff_1 )
    18 measures
    Measure: ( number=1 )
    1 voices
```

```
    Voice: ( voice_item_ref=soprano_voice1 )
    1 chords
        Chord: ( event_ref=v1_e1 duration=1/2
            [ Noteheads:
                staff_ref=staff_1  ( Pitch: step=C octave=6 ) ] )
    1 rests
        Rest: ( event_ref=v1_e0 staff_ref=staff_1 duration=1/2 )


    Measure: ( number=2 )
    1 voices
    Voice: ( voice_item_ref=soprano_voice1 )
    2 chords
        Chord: ( event_ref=v1_e2 duration=1/2
            [ Noteheads:
                staff_ref=staff_1 ( Pitch: step=B octave=5 ) ] )
        Chord: ( event_ref=v1_e3 duration=1/2
            [ Noteheads:
                staff_ref=staff_1 ( Pitch: step=G octave=5 ) ] )

    Measure: ( number=3 )
    1 voices
    Voice: ( voice_item_ref=soprano_voice1 )
    2 chords
        Chord: ( event_ref=v1_e4 duration=1/2
            [ Noteheads:
                staff_ref=staff_1 ( Pitch: step=A octave=5 ) ] )
        Chord: ( event_ref=v1_e5 duration=1/4
            [ Noteheads:
                staff_ref=staff_1 ( Pitch: step=B octave=5 ) ] )

   ...
Part: id=piano
    2 voice items
    Voice List: ( id=r_hand_voice staff_ref=staff_3 )
                ( id=l_hand_voice staff_ref=staff_3 )
    18 measures
    Measure: ( number=1 )
    2 voices
    Voice: ( voice_item_ref=r_hand_voice )
```

```
1 chords
    Chord: ( event_ref=v2_e1 duration=1/2
        [ Noteheads:
            staff_ref=staff_2 ( Pitch: step=C octave=6 )
            staff_ref=staff_2 ( Pitch: step=C octave=5 ) ] )
1 rests
    Rest: ( event_ref=v2_e0 staff_ref=staff_2 duration=1/2 )

Voice: ( voice_item_ref=l_hand_voice )
1 chords
    Chord: ( event_ref=v3_e0 duration=1/1
        [ Noteheads:
            staff_ref=staff_3 ( Pitch: step=C octave=4 )
            staff_ref=staff_3 ( Pitch: step=C octave=3 ) ] )

Measure: ( number=2 )
2 voices
Voice: ( voice_item_ref=r_hand_voice )
2 chords
    Chord: ( event_ref=v2_e2 duration=1/2
        [ Noteheads:
            staff_ref=staff_2 ( Pitch: step=B octave=5 )
            staff_ref=staff_2 ( Pitch: step=B octave=4 ) ] )
    Chord: ( event_ref=v2_e3 duration=1/2
        [ Noteheads:
            staff_ref=staff_2 ( Pitch: step=G octave=5 ) ] )
Voice: ( voice_item_ref=l_hand_voice )
1 chords
    Chord: ( event_ref=v3_e1 duration=1/1
        [ Noteheads:
            staff_ref=staff_3 ( Pitch: step=G octave=4 ) tie
            staff_ref=staff_3 ( Pitch: step=G octave=3 ) tie ]
            )

Measure: ( number=3 )
2 voices
Voice: ( voice_item_ref=r_hand_voice )
2 chords
    Chord: ( event_ref=v2_e4 duration=1/2
        [ Noteheads:
```

```
                staff_ref=staff_2 ( Pitch: step=A octave=5 )
                staff_ref=staff_2 ( Pitch: step=A octave=4 ) ] )
        Chord: ( event_ref=v2_e5 duration=1/4
            [ Noteheads:
                staff_ref=staff_2 ( Pitch: step=B octave=5 )
                staff_ref=staff_2 ( Pitch: step=B octave=4 ) ] )
    Voice: ( voice_item_ref=l_hand_voice )
    2 chords
        Chord: ( event_ref=v3_e2 duration=1/2
            [ Noteheads:
                staff_ref=staff_3 ( Pitch: step=G octave=4 ) tie
                staff_ref=staff_3 ( Pitch: step=G octave=3 ) tie ]
                )
        Chord: ( event_ref=v3_e3 duration=1/2
            [ Noteheads:
                staff_ref=staff_3 ( Pitch: step=F octave=4 )
                staff_ref=staff_3 ( Pitch: step=F octave=3 ) ] )

   ...

10 horizontal symbols
Horizontal Symbol: dynamic staff_ref=staff_2
                   event_ref=v2_e1 value=f

Horizontal Symbol: dynamic staff_ref=staff_2
                   event_ref=v2_e10 value=sf

Horizontal Symbol: dynamic staff_ref=staff_2
                   event_ref=v2_e14 value=f

...
1 lyrics
Lyrics: 30 syllables part_ref=soprano
    30 syllables
    Syllables: ( start_event_ref=v1_e1
                end_event_ref=v1_e1 value=Gott )
        ( start_event_ref=v1_e2
            end_event_ref=v1_e2 value=ist )
        ( start_event_ref=v1_e3
            end_event_ref=v1_e3 value=mein )
```

```
        ...
```

Il file *inventio01*, invece, risulta essere l'unico a utilizzare la componente `ornaments`, nonostante sfrutti solamente un solo tipo di elemento, ovvero `mordent`:

```
6 ornaments
Ornament: mordent event_ref=p1v1_10 type=upper length=normal

Ornament: mordent event_ref=p1v1_58 type=lower length=normal

Ornament: mordent event_ref=p1v1_81 type=upper length=normal

...
```

5.4 Output notational layer

Del layer notational non sono presenti casi in cui viene utilizzata la componente `notation_instance_group`. Al contrario, `graphic_instance_group` è presente in tutti i file presi in considerazione e di seguito viene mostrato il suo contenuto riferendosi al documento *intermezzo_sinfonico*:

```
 ###Notational Layer###
1 graphic instance groups
Graphic Instance Group description=Edizione a stampa :
    3 graphic instances
    Graphic Instance: position_in_group=1
    file_name=score_files/edizione_a_stampa/
intermezzo_sinfonico_pag1.jpg
    file_format=image_jpeg encoding_format=image_jpeg
    measurement_unit=pixels
        360 graphic events
        Graphic Events: ( event_ref=violino10_meas1_voice1_ev1
        upper_left_x=258.0 upper_left_y=592.0
        lower_right_x=278.0 lower_right_y=625.0 )
        ( event_ref=violino10_meas1_voice1_ev3
        upper_left_x=307.0 upper_left_y=581.0
        lower_right_x=327.0 lower_right_y=614.0 )
        ( event_ref=violino10_meas2_voice1_ev1
        upper_left_x=350.0 upper_left_y=582.0
        lower_right_x=370.0 lower_right_y=615.0 )
        ...
```

```
    Graphic Instance: position_in_group=2
    file_name=score_files/edizione_a_stampa/
intermezzo_sinfonico_pag2.jpg
    file_format=image_jpeg encoding_format=image_jpeg
    measurement_unit=pixels
        485 graphic events
        Graphic Events: ( event_ref=oboe4_meas18_voice1_ev1
        upper_left_x=177.0 upper_left_y=126.0
        lower_right_x=193.0 lower_right_y=167.0 )
        ( event_ref=oboe4_meas18_voice1_ev2
        upper_left_x=196.0 upper_left_y=142.0
        lower_right_x=208.0 lower_right_y=169.0 )
        ( event_ref=oboe4_meas18_voice1_ev3
        upper_left_x=210.0 upper_left_y=127.0
        lower_right_x=230.0 lower_right_y=163.0 )
        ...
    Graphic Instance: position_in_group=3
    file_name=score_files/edizione_a_stampa/
intermezzo_sinfonico_pag3.jpg
    file_format=image_jpeg encoding_format=image_jpeg
    measurement_unit=pixels
        499 graphic events
        Graphic Events: ( event_ref=flauto1_meas48_voice1_ev4
        upper_left_x=1026.0 upper_left_y=642.0
        lower_right_x=1044.0 lower_right_y=666.0 )
        ( event_ref=flauto1_meas48_voice1_ev2
        upper_left_x=983.0 upper_left_y=642.0
        lower_right_x=1001.0 lower_right_y=666.0 )
        ( event_ref=flauto1_meas48_voice1_ev1
        upper_left_x=957.0 upper_left_y=632.0
        lower_right_x=979.0 lower_right_y=663.0 )
        ...
```

5.5 Output performance layer

Il layer performance viene utilizzato unicamente dal file *gottes_macht* e soltanto in minima parte:

```
###Performance Layer###
1 MIDI instances
```

```
MIDI Instance: file_name=beethoven_gottes_macht.mid format=1
    1 MIDI mappings
    MIDI Mapping: part_ref=soprano track=1 channel=1
        1 MIDI event sequences
        MIDI Event Sequence (
                                division_type=timecode
                                division_value=1024
                                measurement_unit=ticks
                            )
             1 midi events
            MIDI events: [ timing=0 event_ref=v1_e0 ] )
```

Nei file esaminati, delle componenti `csound_instance` e `mpeg4_instance` dello stesso layer non è stato fatto nessun utilizzo.

Allo stesso modo, non vi è traccia nemmeno dell'intero layer structural.

5.6 Output audio layer

I layer audio viene utilizzato in quasi ogni documento. Il prossimo esempio mostra l'output di questo livello prendendo come campione il file *serie_in_9_8*:

```
###Audio Layer###
6 tracks
audio_files/serie_9_8_sito.mp3
file_format=audio_mp3 encoding_format=audio_mp3
    Track General
    -Recordings: ( date=1988 studio=LIM )
    -Performers: ( name=Antonio José Rodriguez Selles
                    type=computer music )
                ( name=Goffredo Haus type=computer music )
    -Computer music master produced at LIM by
        Antonio José Rodriguez Selles
        and Goffredo Haus (1988)
    Track Indexing ( seconds )
    -Track Events:
        ( start_time=0.5 event_ref=Flauto_1_voice0_measure1_ev0 )
        ( start_time=0.5 event_ref=Marimba_2_voice0_measure1_ev0 )
        ( start_time=0.5 event_ref=
Violoncello_3_voice0_measure1_ev0 )
        ...
```

```
audio_files/serie_9_8_contempo_studio.wav
file_format=audio_wav encoding_format=audio_wav
    Track General
    -Recordings: ( date=1989 )
    -Performers: ( name=Contempo Ensemble type=ensemble )
    -Studio performance by Contempo Ensemble (1989)
    Track Indexing ( seconds )
    -Track Events:
        ( start_time=2.32 event_ref=Flauto_1_voice0_measure1_ev0 )
        ( start_time=2.32 event_ref=Marimba_2_voice0_measure1_ev0 )
        ( start_time=2.32 event_ref=
Violoncello_3_voice0_measure1_ev0 )
        ...

audio_files/serie_9_8_concerto_x_cim.wav
file_format=audio_wav encoding_format=audio_wav
    Track General
    -Recordings: ( date=1993 )
    -Performers: ( name=Angela and Nicoletta Feola
                    type=4 hands piano )
                ( name= type=computer music tape )
                ( name=Angelo Paccagnini type=arranger )
    -Performance at the X Colloquium on Musical Informatics by
        Angela and Nicoletta Feola (4 hands piano)
        and computer music tape;
        arranged by Angelo Paccagnini (1993)
    Track Indexing ( seconds )
    -Track Events:
        ( start_time=0.29 event_ref=Flauto_1_voice0_measure1_ev0 )
        ( start_time=0.29 event_ref=Marimba_2_voice0_measure1_ev0 )
        ( start_time=0.29 event_ref=
Violoncello_3_voice0_measure1_ev0 )
        ...
...
```

Capitolo 6

Sviluppi futuri e conclusioni

6.1 Sviluppi futuri

Lo sviluppo di questa libreria non si conclude con quanto implementato finora. Una libreria, per essere quanto più possibile utile e completa, dovrebbe poter permettere di svolgere diversi tipi di operazioni sulle informazioni di cui si occupa.

Questo lavoro si è concentrato principalmente sullo sviluppo delle strutture dati e sulla lettura del documento IEEE1599, che sono gli strumenti fondamentali per poter accedere ai dati da manipolare, ma altre operazioni in egual misura importanti sono la scrittura e modifica dei contenuti.

Per questo, un primo miglioramento potrebbe essere l'introduzione di una gerarchia di funzioni, strutturata in modo analogo a quanto fatto per le funzioni di caricamento, che permetta di preparare per la stampa, a video o su file, le informazioni caricate in memoria.

Alcune funzioni di stampa a video sono già state introdotte per svolgere i test di funzionamento discussi nel Capitolo 5. Queste possono essere migliorate facendo in modo che ritornino come valore la stringa contenente ciò che dovrebbe essere stampato anziché stamparlo direttamente a video.

Inoltre, il codice potrebbe essere infoltito aggiungendo varie tipologie di funzioni accessorie. Le prime funzioni da implementare potrebbero essere, ad esempio, le funzioni `get` e `set` per ogni tipo di struttura dati presente.

Le funzioni `get`, quelle che ritornano i valori contenuti nelle strutture, potrebbero rivelarsi particolarmente utili per lo sviluppo per le funzionalità di scrittura e stampa.

Le funzioni `set`, invece, sono quelle attraverso cui i valori contenuti nelle istanze delle strutture possono essere modificati. Queste funzioni sono fondamentali per qualsivoglia altro metodo che tra le sue operazioni necessiti questo tipo di operazione.

Dato il massiccio utilizzo delle liste e della numerosità di strutture che le utilizzano, sarebbe bene introdurre anche le funzioni `create`, `remove` e `add` per la creazione, rimozione e aggiunta di elementi ad una lista, in modo da poter agire direttamente sulla struttura gerarchica.

Ne consegue la necessità di introdurre le funzioni di ricerca degli elementi, che si è soliti sviluppare per potere utilizzare tali tipi di metodi.

Queste sono solo alcune delle funzionalità più basilari che solitamente sono rese disponibili dalle librerie normalmente utilizzate.

6.2 Conclusioni

A conclusione della fase di testing trattata nel Capitolo 5, si può affermare che la libreria è capace di leggere e caricare dai documenti IEEE 1599 in modo corretto tutti gli elementi principali e più utilizzati.

Il software soddisfa le aspettative essendo in grado di svolgere le operazioni che ci si era preposti di implementare durante la fase di progettazione:

- Tutti gli elementi, ad eccezione di quelli provenienti da DTD esterni, sono gestiti dalle struct della libreria.
- Tutti i layer del formato vengono caricati separatamente e incorporati in un' unica struct radice, ottenibile tramite la funzione `getIEEE1599Root`.
- Le applicazioni esterne possono utilizzare le principali funzioni della libreria includendo `managerDocument` e caricando interi documenti con `getDoc`.

La libreria risulta quindi essere pronta già da subito per essere utilizzata da applicazioni che necessitano di operare sulle informazioni contenute nei documenti in formato IEEE 1599.

Bibliografia

[1] Andrew S Tanenbaum and Herbert Bos. *I moderni sistemi operativi*. Pearson, 2016.

[2] Luca A Ludovico. Ieee 1599: A multi-layer approach to music description. *Journal of Multimedia*, 4(1), 2009.

[3] Elisa Russo. Multimedia representation of music pieces encoded in symbolic format: an approach based on csound, mpeg-4, supercollider, cml, chuck and ieee 1599. *Journal of Multimedia*, 4(1), 2009.

[4] Luca A Ludovico. Key concepts of the ieee 1599 standard. In *Proceedings of the IEEE CS Conference The Use of Symbols To Represent Music And Multimedia Objects, IEEE CS, Lugano, Switzerland*, pages 15–26, 2008.

[5] Adriano Baratè, Luca A Ludovico, and Davide A Mauro. Tecnologie basate su xml per la fruizione avanzata dei contenuti musicali. In *Conferenza GARR_09 Selected papers*, page 43, 2010.

[6] Shigeru Chiba. Foreign language interfaces by code migration. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 1–13, 2019.

[7] Fabio Salerno Piero Gallo. *Linguaggio C e C++*. Mondadori Education S.p.A., 2009.

[8] Norman Walsh. What is xml. *XML. commune*, 1998.

[9] James Clark, Steve DeRose, et al. Xml path language (xpath), 1999.

[10] Kurt Cagle. *SVG programming: the graphical web*. Apress, 2008.

[11] Joseph Rothstein. *MIDI: A comprehensive introduction*, volume 7. AR Editions, Inc., 1992.

[12] Elliotte Rusty Harold, W Scott Means, and Katharina Udemadu. *XML in a Nutshell*, volume 3. O'reilly Sebastopol, CA, 2004.

Progetto sviluppato presso il Laboratorio di Informatica Musicale
`https://www.lim.di.unimi.it`