UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di laurea in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di software per l'esecuzione di query su partiture musicali

Engineering and development of software designed to execute queries on music scores

Relatore:
Prof. **EMANUELE STORTI**

Tesi di laurea di:
PAOLO CAPPELLETTI

Anno Accademico 2021/2022

Indice

1 Introduzione

Lo sviluppo tecnologico ha modificato, ampliato e migliorato l'interazione tra uomo e arte. Grazie alle varie metodologie di digitalizzazione di suoni e immagini e al World Wide Web, la maggior parte delle opere artistiche sono state rese accessibili da ogni angolo del mondo, e da questa enorme disponibilità stanno nascendo vari progetti volti a sviluppare sistemi sempre più all'avanguardia con lo scopo di mantenere, analizzare, catalogare e studiare tali opere. Molti sono i progetti che cercano di migliorare i sistemi di identificazione e autenticazione di immagini [13] [12], come ad esempio la BIL (Bibliografia Informatizza Leopardiana) una banca dati digitale, o digital repository, che "prevedeva l'ordinamento e la pubblicazione, in un corpus digitale, della sterminata bibliografia leopardiana fino allora disseminata, oltre che nei volumi della Bibliografia leopardiana, in altre bibliografie parziali, repertori, cataloghi"[19].
Questo sviluppo però non è avvenuto omogeneamente se si pensa alle varie forme di arte: alcune, come la scrittura, la pittura e la poesia, hanno avuto uno sviluppo più consistente mentre altre, come la musica, hanno visto questo sviluppo molto più lentamente, per varie cause che verranno studiate in questo lavoro.
La scrittura musicale oggi non ha un formati che ne rendono semplice l'accesso, fruizione e condivisione tra utenti. Inoltre questa difficoltà non permette uno sviluppo agevole di strumenti più sofisticati per supportare tutta una serie di attività quali l'insegnamento e l'apprendimento di quest'arte. Ad oggi gli strumenti di analisi musicale si concentrano più sulla componente sonora e relativa digitalizzazione, che ha una letteratura importante alle spalle e che vede nel Teorema del campionamento di Nyquist-Shannon uno dei suoi capisaldi. Infatti da queste basi sono nati strumenti come Shazam, che raccoglie tutte le

canzoni commerciali "fingerprinted", ovvero ne estraggono e raccolgono in un database vari "hash token"[20].
Questo lavoro di tesi cerca di introdurre strumenti che sia altrettanto potenti, ma basati sulla notazione musicale. Nello specifico riguarda la ricerca, sperimentazione e implementazione di formati di rappresentazione e vocabolari al fine di poter interrogare partiture musicali in formato digitale con lo scopo di supportare l'analisi delle partiture stesse, che comprende sia tutto il lato musicale che il testo delle opere.
Il lavoro svolto unirà sia sezioni di pura ricerca, studio e analisi di linguaggi e ontologie che fasi di ingegnerizzazione, con analisi dei requisiti, implementazione e testing di codice al fine di ottenere il risultato sperato, ed è stato separato in varie fasi che saranno rappresentate dai capitoli di questa tesi che andranno a trattare i seguenti argomenti:

- **Stato dell'arte**, in cui si descrive la situazione attuale dello sviluppo tecnologico nell'archiviazione di partiture musicali digitali e nella loro rappresentazione, con linguaggi come il MIDI e ontologie come il Music Theory Ontology;
- **Requisiti del lavoro**, in cui si va ad introdurre il lavoro effettivamente svolto al fine di ottenere una rappresentazione quanto più esatta delle partiture che permetta di eseguire interrogazioni utili all'analisi delle opere stesse
- **Strumenti utilizzati**, in cui si introducono tutti i software che sono stati utili alla realizzazione di quest'opera;
- **Ontologia**, in cui si presenta l'integrazione all'ontologia MusicXML da me proposta, al fine di catturare aspetti non ancora presi in considerazione delle partiture musicali;
- **Progettazione e implementazione modifiche traduttore**, in cui si andranno a descrivire le due fasi di lavorazione, prima con un focus sulla

divisione del lavoro poi con una descrizione dettagliata delle operazioni svolte, con tanto di nomi delle classi e funzioni modificate;

- **Interrogazioni sulla partitura**, in cui saranno introdotte interrogazioni il cui funzionamento è stato confermato dall'autore stesso;
- **Conclusioni**, in cui si descrivono i risultati raggiunti sia da un punto di vista dello studio dello stato dell'arte che del lavoro svolto nella modifica del traduttore, e si indicano anche dei possibili sviluppi futuri.

2 Stato dell'arte

Le partiture musicali digitali oggi offrono sfide sia dal lato della memorizzazione e catalogazione in digital repository che della rappresentazione e standardizzazione delle opere stesse. Alla base di queste sfide vi è sempre la mancanza di una linea unica, ovvero di standard sia nella memorizzazione che nella rappresentazione da cui partire per sviluppare strumenti più avanzati. Infatti quello che si potrà notare in questo capitolo è la presenza di vari sistemi anche molto eterogenei che però difficilmente riescono a diventare la base da cui sviluppare i lavori futuri.

2.1 Memorizzazione

Per ciò che riguarda la rappresentazione di una partitura, le informazioni raccolte dalla maggior parte dei repository digitali riguardano principalmente i metadati. Essi sono un insieme di dati che descrivono il documento musicale, e che includono ad esempio il titolo dell'opera, l'autore, l'anno di composizione, l'editore, eventuali traduttori e arrangiatori.
Esempio noto è MusicBrainz, ovvero un "community database di metadati musicali e un 'enciclopedia musicale pubblica' a cui gli utenti contribuiscono con informazioni di artisti, pubblicazioni, tracce, e altri aspetti della musica con l'obiettivo di creare un 'sito musicale completo'"[4], quindi un database di aspetti laterali all'opera della quale non viene richiamato nessun aspetto musicale, se non in alcuni casi gli strumenti o musicisti coinvolti.
Successivamente, anche a seguito di tecnologie di cui parleremo nel prossimo paragrafo, i database hanno iniziato a incorporare anche l'opera in sè, rappresentando la partitura in diversi modi, alle volte sfruttando i linguaggi di rappresentazione esistenti, altre volte semplicemente come immagini digitalizzate. Di quest'ultimo gruppo fa parte l'IMSLP o Petrucci Music Library il cui

obiettivo è "mettere insieme tutti i pezzi musicali di pubblico dominio, in aggiunta ai pezzi musicali di compositori contemporanei (o le loro proprietà) che sognano di pubblicarli in maniera gratuita"[8], come anche lo Sheet Music Consortium, che è "un gruppo di biblioteche che lavorano al fine di costruire una collezione pubblica di partiture musicali digitalizzate usando Open Archives Initiative Protocol for Metadata Harvesting"[7]. Entrambe queste opere hanno alla loro base la volontà di rendere quante più opere possibili disponibili alla massa, ed infatti entrambe anche se in maniera diversa coinvolgono gli utenti stessi nel caricamento delle opere stesse, a discapito di una standardizzazione della rappresentazione delle opere, che nella maggior parte dei casi risultano essere file in formato .png o .pdf, quindi inutilizzabili ai fini del nostro lavoro. Esistono repository in grado di soddisfare le necessità del lavoro svolto, come ad esempio il NEUMA[6], libreria digitale che ha lo stesso scopo delle due precendenti ma con una maggiore percentuale di brani in un formato consono, ma l'assenza di una linea comune non ha permesso uno sviluppo uniforme delle tecnologie.

2.2 Formati di rappresentazione

La rappresentazione digitale di partiture musicali è un ambito che negli ultimi 40 anni è stato parecchio esplorato e, se paragonato agli altri ambiti trattati in questo lavoro, è sicuramente quello in cui la ricerca e sviluppo di linguaggi ha raggiunto i migliori risultati. Vari formati oggi sono largamente utilizzati da ampie comunità, e molti strumenti di supporto sono nati attorno a questi linguaggi, tra cui i più noti sono i software che permettono di stampare a schermo la parititura da queste rappresentazioni. In questo capitolo verrano illustrati alcuni dei formati più utilizzati, mettendo in luce pregi e difetti di ognuno di essi.

2.2.1 MIDI

Il Musical Instrument Digital Interface (MIDI) è il più longevo dei formati presentati, infatti è l'unico proveniente dal ventesimo secolo. Presentato nel 1981, il protocollo MIDI rappresenta uno dei più utilizzati protocolli di comunicazione digitale ed ha avuto un grande utilizzo da parte dei costruttori di strumenti come "mezzo per permettere una comunicazione rapida e standardizzata tra generatori di suono e attrezzatura di controllo, registrazione e computer prodotti da diverse compagnie"[2].
In generale MIDI presenta un'interfaccia composta da 3 elementi essenziali: MIDI Out, MIDI in, MIDI Thru. Questi elementi sono porte unidirezionali che svolgono diversi compiti:

- al MIDI Out spetta il compito di inviare i dati ad un altro dispositivo, nello specifico una porta MIDI In;
- al MIDI In spetta il compito di ricevere i dati da una porta MIDI Out o MIDI Thru;
- al MIDI Thru spetta il compito di copiare i dati in ingresso e inviarli ad una porta MIDI In. Uno degli scopi essenziali del MIDI Thru è quello di permettere il collegamento di più dispositivi ad una singolo dispositivo, spesso definito master.

I dati vengono scambiati tra queste porte sotto forma di variabili che descrivono eventi musicali, quali una nota con relativa tono e ottava, ma con caratteristiche meno legate ad aspetti prettamente inerenti alle partiture musicali come il canale assegnato oppure dati relativi al dispositivo che ha attivato l'evento.
Questo appena descritto è uno dei limiti del formato MIDI, infatti il formato è più incentrato sui collegamenti e sulla possibilità di riprodurre la musica stessa, non sull'aspetto di notazione musicale, e questo per quanto abbia interesse e catturi un certo mercato, non si adatta al lavoro presentato.

2.2.2 MEI

Il Music Encoding initiative (MEI) è un "sforzo open-source della comunity al fine di definire un sistema di codificazione di documenti musicali in una struttura leggibile dai calcolatori"[9], ovvero un formato in grado di codificare partiture musicali su cui hanno lavorato esperti di ogni settore, da bibliotecari a storici a esperti del settore informatico.
Il formato si basa su su eXtensible Markup Language (XML), quindi fa uso di tag e relativi attributi per definire la propria struttura. I tag sicuramente di maggiore importanza sono <measure> e <note>, ovvero la struttura base attorno alla quale si sviluppano i brani musicali. La <measure>, in italiano battuta, sta a rappresentare un semplice contenitore in cui vengono inserite direttamente tutte le strutture che possono contenere le note: accordi, <beam> o le note stesse. Se per accordi e note le definizioni da un punto di vista musicale è banale, con il tag <beam> il linguaggio va ad intendere un gruppo di node le cui code vanno collegate.

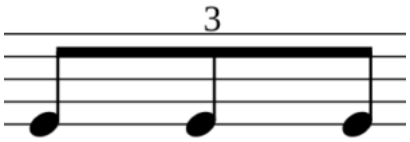

Figura 1: Esempio di <beam>

Per quanto riguarda il tag <note>, questo è ricco di vari tag che ne vanno a descrivere tutte le caratteristiche, dalle più banali come durata, tono e ottava, ad altre come la direzione del corpo e se la nota è o meno un'acciaccatura. Da notare che il formato MEI differenzia note e pause, infatti è presente il tag <rest>.
Uno dei punti di maggiore forza del MEI è che lo sforzo non si è concentrato solamente sulle partiture che seguono lo standard occidentale moderno, ma ha anche abbracciato altri standard maggiormente diffusi in epoca rinascimen-

tale, come la notazione mensurale, e medievale, come la notazione neumatica. Questo aspetto permette al formato, rispetto agli altri presentati, ne dà una maggiore rilevanza per quanto riguarda il bacino di brani che è in grado di rappresentare, e quindi ne amplia le possibilità di utilizzo. Inoltre il formato MEI è molto utilizzato in ambito puramente accademico per la sua capacità di memorizzare molti metadati legati alla singola opera.
Uno degli aspetti più negativi del linguaggio e che ne limita in parte la diffusione è la separazione tra le note e la chiave: il linguaggio prevede che la definizione della chiave, che coinvolge sia il tipo di chiave (violino, basso...) che la scala oltre ad altre indicazioni, non avviene nel momento in cui appare nel brano, bensì è definita a priori e richiamata tramite l'attributo n del tag <staff>, mentre la definizione con tutte le caratteristiche descritte avviene nel tag <staffDef>. Questo crea delle difficoltà di lettura a chi non è un addetto dei lavori e quindi complica la possibilità di diffusione del formato.

2.2.3 MusicXML

Il MusicXML è un formato il cui scopo è quello di semplificare lo scambio di informazioni relative alle partiture musicali. Il linguaggio, inventato negli anni 2000 da Michael Good, nasce con l'idea di rappresentare partiture musicali scritte in notazione moderna occidentale con la possibilità, anche se non ancora espressa, di estendersi alla rappresentazione di altre notazioni musicali. Come descritto dallo stesso creatore Michael Good nell'articolo "MusicXML for Notation and Analysis"[3], il linguaggio nasce con lo scopo di semplificare e, di conseguenza, aumentare la circolazione di informazioni relative alle partiture musicali tra esperti del settore e non, quindi il linguaggio è strettamente sufficiente a raggiungere questo obiettivo. Questo implica che esisteranno altri linguaggo che avranno comportamenti migliori di MusicXML in altri ambiti (come il MIDI, già introdotto in questo lavoro, per la riproduzione della

musica), ma è ottimizzato per lo scambio di informazioni, e a dimostrazione di ciò si basa su uno dei linguaggi che storicamente rappresentazione lo scambio di informazioni: XML. Qui possiamo vedere un punto di contatto col formato MEI, ma subito si discosta da quest'ultimo per l'approccio che ha: se il formato MEI ha come punto di partenza i metadati e si concentra su di essi e sul loro scambio, il formato MusicXML mette in risalto la musica e la sua scrittura, andando ad interpretare in maniera più approfondita e accurata ogni suo aspetto, andando poi ad aggiungere ad essa tutti i metadati presenti.
Come nel formato MEI, anche nel formato MusicXML ci sono dei tag che rappresentano ogni aspetto della partitura, ma a differenza dell'altro formato si ha un uso minore degli attributi e maggiore dei tag. Aspetti come direzione del corpo della nota, altezza e tono non sono più attributi, ma tag specifici. Specificato il tag <note>, che conterrà tutti i dettagli della nota, il formato MusicXML prevede dei tag figli tra cui <stem>, <beam> e <pitch> che a sua volta prevede dei tag figlie quali <step> che indica la nota, <alter> che indica della alterazioni (diesis, bemolle...) e <octave> che indica l'ottava. Un tag figlio del tag <note> da evidenziare è il tag <lyric>, in cui vengono raccolte informazioni relative al testo che quella nota rappresenta. Quest'ultimo ha tag figli quali <text>, che indicano il testo vero e proprio, e <syllabic>, che dà informazioni relative al testo, indicando se è un parola completa o se sono sillabe iniziali finali o intermedie. Questo aspetto è importante sia da un punto di vista meramente grafico, poichè in una partitura tra sillabe della stessa parola a cui corrispondono note diverse viene aggiunto un tratto d'unione, che di rappresentazione dell'informazione stessa, poiché permette di ricostruire il testo completo del brano partendo dalla rappresentazione in MusicXML.
Per quanto riguarda la battuta, rappresentata dal tag <measure>, in essa sono contenute tutte le note, col relativo tag <note>, sotto forma di tag figli, mentre altri tag che ne descrivono le caratteristiche restanti. Di particolare interesse

```
<part id="P1">
  <measure number="1">
    <attributes>
      <divisions>1</divisions>
      <key>
        <fifths>-1</fifths>
        <mode>major</mode>
      </key>
      <time symbol="cut">
        <beats>2</beats>
        <beat-type>2</beat-type>
      </time>
      <clef>
        <sign>C</sign>
        <line>3</line>
      </clef>
    </attributes>
    <sound tempo="120"/>
    <note>
      <pitch>
        <step>C</step>
        <octave>4</octave>
      </pitch>
      <duration>4</duration>
      <voice>1</voice>
      <type>whole</type>
      <lyric number="1">
        <syllabic>single</syllabic>
        <text>Qui</text>
      </lyric>
    </note>
    <barline location="right">
      <bar-style>none</bar-style>
    </barline>
  </measure>
```

Figura 2: esempio di parte di file MusicXML

è il tag <attribute> che contiene dati relativi a chiavi da rappresentare a inizio battuta, rappresentata dal tag <clef>, le indicazioni di tempo, rappresentate nel tag <time> e specificando nei tag figli <beats> e <beat-type> la frazione che rappresenta il tempo, la tonalità, rappresentata nel tag <key>, contenuta nello specifico nel tag <fifths> e <mode> che indica se maggiore o minore, e quali strumenti devono suonare, rappresentato nel tag <instruments>. Si può trovare una particolarità del MusicXML nella rappresentazione della tonalità, poiché invece di rappresentarla banalmente tramite la prima nota dell'accordo viene

rappresentata con un numero reale che rappresenta i diesis o bemolle presenti, dando ai diesis valore positivo e ai bemolle valore negativo. Altri aspetti rilevanti e che rappresentano forse anche una delle maggiori critiche al linguaggio MusicXML sono contenuti nel tag <direction> figlio del tag <measure>. Infatti in questo tag, più nello specifico all'interno del tag <direction-type> figlio di <direction>, sono contenute le informazioni relative ai crescendo e diminuendo, nel tag <wedge>, e alle dinamiche (piano, pianissimo, forte...), nel tag <dynamics>. Queste caratteristiche in realtà non sono proprie della battuta, bensì di specifiche note, e andrebbero trattate al pari di altri abbellimenti musicali giustamente agganciati alle note specifiche. Per esempio i crescendo e diminuendo dovrebbero avere un trattamento simile alle legature, rappresentate da due tag differenti, <tied> per legature tra due note di due battute diverse e <slur> per le restanti occasioni. Il tag <slur> a sua volta può assumere due valori: start e stop. Usando questo metodologia si potrebbero rappresentare in maniera più corretta i crescendo e diminuendo, legandoli alle note di inizio e fine dinamica.
Per quanto riguarda le dinamiche esse potrebbero ricevere un trattamento simile alle articolazioni musicali, che sono rappresentate nel tag <articulations> figlio di <note> come un tag vuoto. Quind avremo i tag <staccato />, <tenuto />, che rappresentano le varie articolazioni. Lo stesso si potrebbere fare spostando il tag <dynamics> come figlio di <note> e usando i tag <p />, <f /> già esistenti.
Alle rappresentazioni di battute e note si aggiungono altre sovrastrutture comunque inerenti la paritura musicale, come la <part> ovvero la parte vera e propria, intesa come semplice contenitore di battute, a cui si può aggiungere un identificatore, e la <score-partwise>, tag radice dei file MusicXML. Oltre a contenere tutte le parti, a questo tag vengono associati i vari metadati sfruttando altri tag figli come <identification>, <work> o <credit>. In <identifica-

tion> sono contenuti tutti i metadati relativi all'autore e ai detentori dei diritti di copyright, nel tag <work> aspetti relativi all'opera, come nome e numero, e il tag <credit>, che non ha cardinalità singola a differenza degli altri due, gestisce i dati relativi uno specifico esperto che ha lavorato all'opera, come un arrangiatore, un compositore o un autore.
Nonostante le criticità riguardo la rappresentazione di alcuni aspetti della partitura musicale, il formato MusicXML si dimostra il migliore per il lavoro da svolgere, ed infatti per tutto il resto del lavoro saranno prese in considerazione solo opere in formato MusicXML.

2.3 Vocabolari e Ontologie

L'aspetto di vocabolari in grado di standardizzare la rappresentazione sia dei metadati inerenti un'opera che l'opera stessa l'aspetto più critico della digitalizzazione delle partiture musicali.
L'assenza di questo materiale ha sicuramente rallentato tutto il processo di ricerca e analisi che invece è avvenuto per le altre arti, questo per una ragione principale: l'assenza di vocabolari ha reso più difficile l'interoperabilità tra dati, e questo ha creato difficoltà a raccogliere grandi quantità di dati. Per fare ciò si è stati costretti a uniformare manualmente diversi dataset cambiando, a seconda delle condizioni, l'approccio alla conversione dei dati. Questo processo, soggetto a errori e dispendioso in termini di tempo, è uno degli aspetti più rilevanti del processo che stiamo descrivendo.
Negli ultimi anni sono nate alcune soluzioni di standardizzazione dei metadati, anche grazie alla spinta di Open e FAIR Data. Per Open Data si intende una filosofia che prevede la libertà dei dati da diritti di copyright. Come ben spiegato nell'articolo "The State of Open Data"[1] di Katrin Braunschweig, Julian Eberius, Maik Thiele e Wolfgang Lehner, l'obiettivo dell'iniziativa Open Data è di rendere pubblici tutti i dati non personali e non a fini commerciali,

soprattutto (ma non esclusivamente) quelli raccolti e processati da organizzazioni governative"[14], dunque una spinta verso la liberalizzazione dei dati soprattutto se detenuto da organizzazioni governative.
Per FAIR Data invece si intendono dati che siano "Findable, Accessible, Interoperable, and Reusable"[10]: Per Findable si intende che siano facilmente trovabili sia a persone che macchine, quindi una buona indicizzazione dei dati e strutturazione dei dati. Per Accessible si intende che i dati, una volta trovati, devono essere resi accessibili tramite metodi di autenticazione e autorizzazione. Interoperable intende l'esistenza di uno scheletro che permetta a dati omogenei nella sostanza di essere integrati, quindi omogenei anche nella forma. Infine per Reusable si intende la semplicità all'uso e riutilizzo dei dati, quindi una buona documentazione di dati e metadati.
A seguito dello sviluppo di queste filosofie si sono sviluppate delle soluzioni per quanto riguarda la creazione di dizionari di dati per le partiture musicali, ed alcune di esse saranno descritte all'interno di quest'opera, inoltre saranno inseriti dei commenti relativi alla possibilità di usufruire dei vocabolari descritti ai fini di questo lavoro.

2.3.1 MIDI Linked Data Cloud

Il MIDI Linked Data Cloud è un dataset che ha lo scopo di "rappresentare molte collezioni di opere musicali digitalizzate nel formato standard MIDI come Linked Data usando il nuovo midi2rdf"[15], quindi sfruttare la tecnologia dei Linked Data per standardizzare la rappresentazione fornita dal formato MIDI. Nello specifico, i Linked Data sono una soluzione trovata all'interno del web, infatti Pascal Hitzler e Krzysztof Janowicz nell'editoriale "Linked Data, Big Data, and the 4th Paradigm"[5] li definiscono nel seguente modo: "I Linked Data prendono l'idea del World Wide Web di identificari globali e collegamenti e li applica ai dati grezzi, non solo ai documenti". Come mostrato in figura

3, che mostra un estratto dell'ontologia, si può notare come al centro del progetto sia messo il concetto di evento, che è estraneo alla partitura musicale, e che il linguagggio di riferimento è il MIDI, di cui si sono già discussi i limiti relativi a questo. Questi aspetti non permettono di prendere in considerazione la presente ontologia, mentre potrebbe essere di interesse per altri lavori che si sposano meglio col formato MIDI.

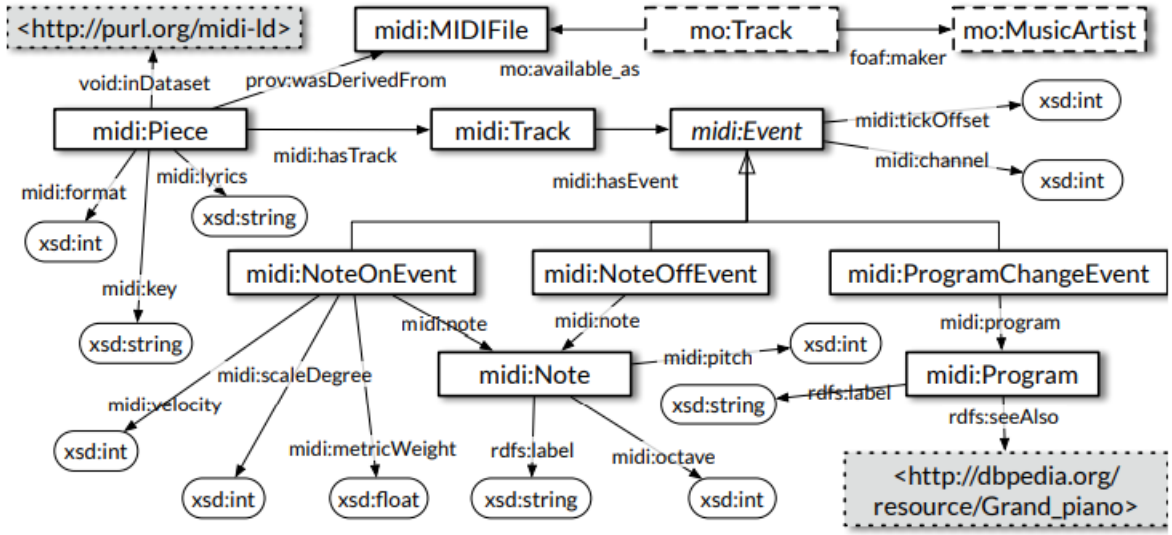

Figura 3: Estratto dell'ontologia MIDI:pezzi, tracce, eventi e i loro attributi[15]

2.3.2 HaMSE

L'ontologia HaMSE (Harmonic, Melodic, Structural Emotional), svulippata come estensione al MIDI Linked Data Cloud, si propone come obiettivo quello di unire due tipologie diverse di rappresentazione della musica: audio e partiture. Inoltre , come descritto dall'articolo con cui i creatori hanno spiegato il loro progetto, "L'ontologia proposta servirà a codificare formalmente informazioni musicali usando lo una rappresentazione standard interoperabile, chiamata RDF/OWL Knowledge Graphs"[17]. L'uso dei Knowledge Graph, già visto anche se non esplicitato nel MIDI Linked Data Cloud, permette di rappresentare delle informazioni attraverso dei grafi e ne permette delle rappre-

sentazioni grafiche di alto impatto. Nella figura 4 troviamo una parte dell'ontolgia, rappresentante quello che viene definito evento simbolico, con cui si intende sia la nota che la pausa, e i possibili aspetti che la potrebbero caratterizzare. Per quanto l'ontologia sia di base di buona fattura, l'ereditare dal formato MIDI di cui si sono già descritti i limiti all'applicabilità in questo lavoro ne inficia la valutazione complessiva.

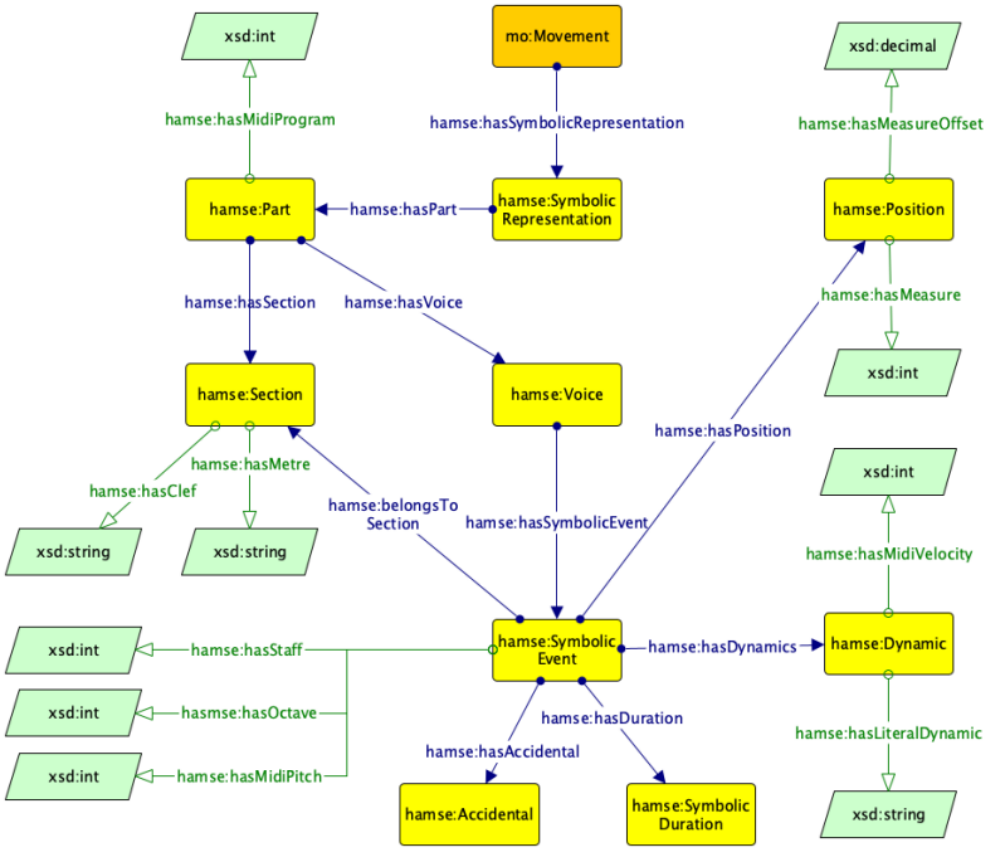

Figura 4: Dettaglio dell'ontologia in cui si rappresenta l'evento musicale simbolico[17]

2.3.3 Music Theory Ontology

Music Theory Ontology è un progetto nato con lo scopo di creare un vocabolario che partendo da lavori precedenti ampliando il numero di aspetti teorici trattati. Nella presentazione del progetto gli autori specificano che si sono concentrati sulla musca occidentale moderna in quanto a notazione e che

"le classi sono organizzate gerarchicamente in termini di inclusione usando rdfs:subClassOf. Includiamo le proprietà di annotazione rdfs:label e rdfs:comment per ogni classe e proprietà dell'ontologia"[18]. Per quanto il primo aspetto può risultare una limitazione, in quanto alcuni formati teoricamente prevedono la possibilità di rappresentare opere con diverse notazioni, nella realtà esso rappresenta al momento una semplice limitazione anche dovuta all'assenza di formati che veramente prevedono la rappresentazione di opere scritte in notazione neumatica o mensurale. Altro aspetto introdotto anche in questo linguaggio è la stretta dipendenza dal Resource Description Framework, strumento sviluppato da W3C che, per definizione stessa data da W3C "RDF è un modello standard per lo scambio di dati nel web. RDF ha proprietà che facilitano l'unione di dati anche se sottostanno a schemi differenti, nello specifico supporta l'evoluzione degli schemi nel tempo senza dover modificare i dati stessi"[11]. Si tratta di uno strumento potente che si adatta perfettamente alla situazione, e verrà ulteriormente approfondito in seguito. Tra i principali aspetti introdotti dal Music Theory Ontology sono le chiavi, le alterazioni e l'indicazione del tempo, oltre a specificare la durata delle note e delle pause e introdurre aspetti teorici riguardanti gli accordi(maggiore/minore, accordo di sesta...). Come descritto dalle conclusioni dell'articolo "A Music Theory Ontology" scritto dai creatori dell'ontologia stassa, il punto debole del Music Theory Ontology è essere incompleto: aspetti importanti già trattati nei linguaggi di rappresentazione come le dinamiche sono non pertinenti il lavoro svolto, aspetti che sono stati trattati da altre ontologie.

2.3.4 MusicOWL

L'ontologia MusicOWL, sviluppata da Jim Jones, Diego de Siqueira Braga, Kleber Tertuliano e Tomi Kauppinen, è un'ontologia con lo scopo di descrivere il contenuto di partiture musicali di musica occidentale moderna. Il proget-

to, sviluppato con una solida base di partiture fornite dall'università di Munster, si è inserito come collante tra varie altre ontologie, andando poi a sviluppare quasi ogni aspetto di una partitura musicale: come si vede nella figura 5, l'ontologia si interfaccia con altre, come la Music Ontology che si prende cura degli aspetti più di alto livello quali la partitura stessa e i movimenti, l'ontologia Chord, che esprime in maniera esaustiva la rappresentazione di una nota, e la Tonality Ontology, che come dice il nome standardizza la rappresentazione delle tonalità. Le ontologie appena presentate non sono presenti in questo capitolo poiché come anche dimostrato dal lavoro svolto per la realizzazione dell'ontologia MusicOWL esse sono estremamente specifiche e verticali su un singolo aspetto della musica. Questo aspetto unito al fatto che l'ontologia MusicOWL riprende molti termini utilizzati ha portato alla conclusione di non inserire nello stato dell'arte le ontologie Chord, Music Ontology e Tonality Ontology.
Nello specifico l'ontologia si occupa di standardizzare la rappresentazione di tutto il resto, andando dalla parte stessa, intesa come partitura del singolo strumento, il pentagramma, le battuta, gli accordi, che prendono il nome di "noteset", alterazioni, dinamiche e durata delle note, chavi ed altri aspetti. A dimostrazione dell'esaustività del lavoro svolto vi è la rappresentazione dei ritornelli, aspetto ignorato dagli altri vocabolari trattati. Alcune criticità sono presenti, infatti sono assenti sia una standardizzazione del testo di un'opera, caratteristica però comune a tutte le ontologie presentate, che delle legature o delle corone, aspetto invece trattato in alcune di esse. All'ontologia però, a differenza delle altre ontologie, viene associato un software di traduzione che accetta vari formati in ingresso, tra cui il MusicXML, e produce file anch'essi in vario formato ma sempre basato su una rappresentazione RDF. Quest'ultimo aspetto aumenta di gran lunga la fruibilità dell'ontologia e di conseguenza amplifica di molto l'utilizzo dell'ontologia stesso. L'ontologia quindi, per quanto possa ri-

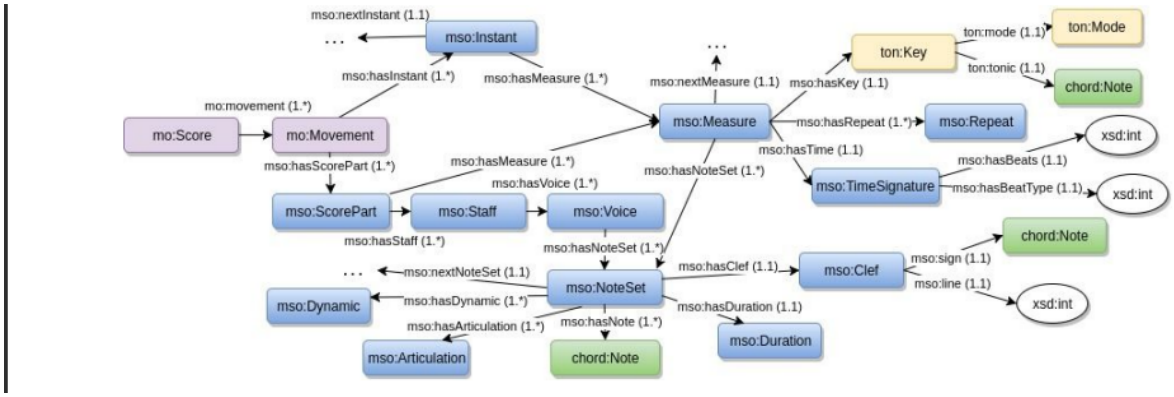

Figura 5: MusicOWL: Panoramica generale

sultare incompleta, è un ottimo punto di inizio per trovare un'efficace metodo per rappresentare le partiture in un formato che ne renda possibile l'esecuzione di query.

3 Requisiti del lavoro

A seguito dell'analisi avvenuta con l'illustrazione dello stato dell'arte, si è illustrato come l'ontologia MusicOWL, assieme ad altre ontologie, sia la migliore soluzione per rappresentare partiture musicali, inoltre è già presente un convertitore MusicXML-RDF che si appoggia proprio sull'ontologia MusicOWL il cui codice è stato reso pubblico dal team di sviluppo. Di conseguenza l'obiettivo generale sarà estendere l'ontologia con ulteriori termini che rappresentino aspetti delle partiture musicali non trattati da altre ontologie, implementare la nostra estensione all'interno del traduttore e analizzare i risultati ottenuti, fornendo inoltre una serie di esempi di query che possono essere effettuate su partiture in RDF.
Per quanto riguarda l'estensione, sulla base delle documentazioni delle altre ontologie, si possono notare i seguenti aspetti che sono stati tralasciati:

- **Testo**: con questo requisito si intende standardizzare la rappresentazione del testo sia nella sua interezza, che come sillabe associate alle note

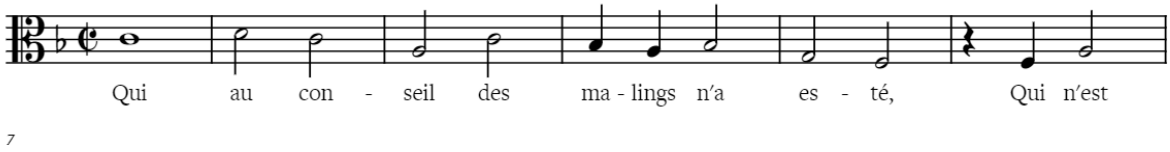

Figura 6: Esempio di spartito con testo

- **Legatura**: per legatura si intendono tutti i tipi di legatura che la musica conosce, sia di valore, di portamento o di frase, e in generale è una figura musicale che unisce due o più note consecutive andando ad indicare al musicista di suonare quelle note nella maniera più fluida possibile (ad esempio per gli strumenti a fiato indica un'unica emissione di fiato per il gruppo di note). Una legatura di valore è presente tra note della stessa altezza e ne determina il valore, mentre legatura di frase e portamento diffe-

riscono solamente per la lunghezza della legatura stessa, infatti la legatura di frase abbraccia tutta una frase musicale mentre quella di portamento solo una porzione.

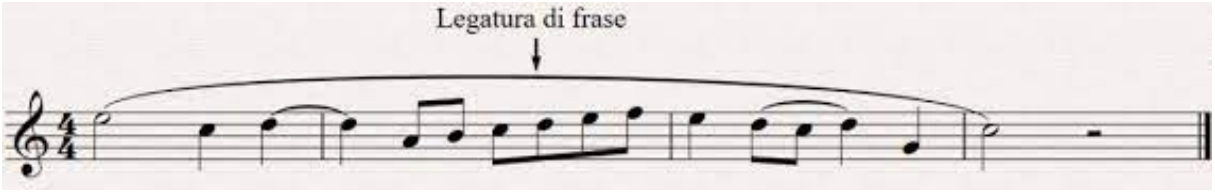

Figura 7: Esempio di legatura di frase con legature di valore(prima/seconda battuta) e di portamento(terza battuta)

- **Trillo**: per trillo si intende una figura musicale in cui una nota si alterna rapidamente con la nota ausiliaria superiore per grado congiunto. In parole povere la nota si alterna con la nota immediatamente successiva nella scala definita dalla tonalità.

Figura 8: Esempio di trillo

- **Dinamica**: per dinamica si intende una figura musicale in cui un gruppo di note sono coinvolte in un cambiamento graduale dell'intensità del suono. Ne esistono due tipi: crescendo, dove l'intensità aumenta, e diminuendo, dove l'intensità diminuesce. È buona norma indicare l'intensità iniziale e finale sulle note che rappresentano l'inizio e la fine della dinamica.

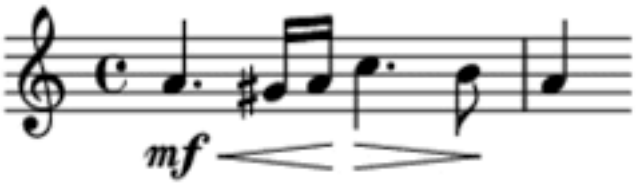

Figura 9: Esempio di crescendo e diminuendo

- **Corona**: per corona si intende una figura musicale in cui una nota vede il suo valore prolungato. In caso di esibizione singola, la durata è decisa dall'esecutore, altrimenti spetterà al direttore d'orchestra deciderne la durata.

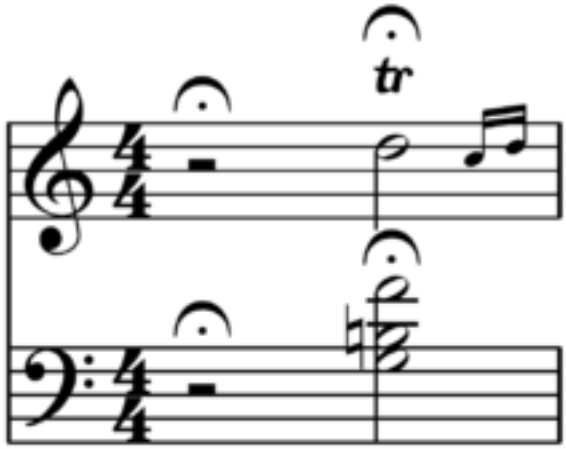

Figura 10: Esempio di corona sia di una pausa che di una nota

Una volta terminato lo sviluppo dell'ontologia, si dovranno portare modifiche al convertitore proposto da MusicOWL al fine di integrare anche la nostra estesione proposta.

4 Strumenti utilizzati

In questo capitolo verranno esposti gli strumenti utilizzati per lo sviluppo delle modifiche del convertitore, nello specifico:

- **Music Score to RDF Converter**, il convertitore fornito dal team che ha sviluppato l'ontologia MusicOWL;
- **Eclipse** come IDE per la programmazione;
- **GraphDB** per testare i risultati ottenuti dalla conversione dei file.

Inoltre verranno introdotti due linguaggi che verranno utilizzati all'interno del progetto: RDF e SPARQL.

4.1 RDF

Il Resource Descriptor Framework è un modello di rappresentazione delle informazioni. Grazie alla sua spiccata flessibilità "supporta modelli di dati generici basati sui grafi e formati di rappresentazione dati per descrivere cose, includendole relazioni con altri oggetti"[21]. Proposto da W3C, RDF è uno strumento molto potente, in grado di fondere dati che utilizzano schemi differenti con relativa facilità. Questa caratteristica viene implementata attraverso l'utilizzo di URI per rappresentare le informazioni e una struttura fissa dello schema che verrà presentata successivamente, rendendo l'RDF uno dei formati più utilizzati nella rappresentazione di dati il cui schema può cambiare nel tempo. È questo il caso delle partiture musicali, dove le variazioni di schema sono dovute sia a possibili modifiche del lingaggio stesso (basti pensare al cambiamento avvenuto nel passaggio tra tetragramma e pentagramma) sia all'assenza di uno schema ufficiale, bensì alla presenza di vari schemi diversi ognuno con una propria ragion d'essere. Il modello RDF si basa sugli statement, ovve-

ro triple del tipo soggetto-predicato-oggetto dove un soggetto è una risorsa, un predicato una proprietà e un oggetto in valore. Nello specifico:

- una risorsa è l'unità fondamentale rappresentata dal modello RDF. Essa viene sempre rappresentata da un URI (Unified Resource Identifier);
- una proprietà è un legame tra una risorsa e un valore e viene anch'essa espressa attraverso un URI;
- un valore è un tipo di dato primitivo. Esso può essere una stringa, un numero o un URI che fa riferimento ad un'altra risorsa.

Un esempio di tripla è la seguente:
chord:Note chord:natural note:Rest
In cui chord:Note indica l'URI di una risorsa che in questo caso rappresenta il soggetto, chord:natural è l'uri rappresentante la proprietà mentre note:Rest è un valore espresso sotto forma di URI. Concettualmente il modello RDF nasce con lo scopo di rappresentare modelli di dati non relazionali,bensì a grafo, dove le varie risorse sono i nodi del grafo mentre le linee sono rappresentarappresentarrappresentate dalle proprietà.

4.2 Music Score to RDF Converter

Il "Music Score to RDF Converter" è il convertitore che permette di trasformare un file MusicXML in RDF basandosi sulll'ontologia MusicOWL. Scritto in linguaggio Java, esso prende in ingresso file di tipo XML, per produrre output di vari formati, come il Turtle, N-Triples, RDF/XML e RDF/JSON, accomunati dall'utilizzare la notazione RDF. In generale alla chiamata del convertitore si possono passare vari parametri, quali:

- "setInputFile", in cui si specifica il percorso del file codificato come MUSICXML 3.0 che deve essere convertito.

- "setOutputFile", in cui si specifica il percorso e nomedel file di output, ma non il formato.
- "setThumbnail", in cui si specifica la thumbnail (miniatura) della partitura convertita (Opzionale).
- "setScoreIdentifier", in cui specifica l'URI della partitura convertita. (Opzionale)
- "setScoreTitle", in cui si specifica il titolo dell'opera.
- "issued", in cui si specifica la data di rilascio. I formati accettati sono yyyy, yyyyMM, yyyyMMdd. (Opzionale)
- "setCollection" , in cui si specifica una categorizzazione, ad esempio temporale, dell'opera. Alcuni esempi sono "Compositori ottocenteschi" o "Partiture avanzate".
- "isVerbose", parametro booleano prettamente tecnico per mostrare nella console lo stato della conversione, sotto forma di log.
- "setOutputFormat", in cui si specifica solamente il formato del file di ouput, tra quelli già mostrati nella parte introduttiva di questo capitolo.
- "addPerson", in cui si specifica una persona che ha contribuito all'opera. Consiste in un URI, un nome ed un ruolo, che può avere determinati valori prederminati quali Role.ARRANGER, Role.COMPOSER o Role.TRANSLATOR.
- "addResource", in cui si specificano risorse strettamente legate all'opera, come sue rappresentazioni in formati audio. come .mp3, o strettamente visivi, come il .pdf. Per ogni risorsa devono essere presenti 3 parametri: link all'opera, una descrizione testuale e il MIME Type.

Da un punto di vista della progettazione del codice, esso si divide in 4 parti principali:

- converter, ovvero l'insieme di tutte le classi dedite alla conversione. Di notevole importanza sono la presenza del metodo main nella classe Converter.java e la class MusicXML2RDF.java, che rivedremo nei prossimi capitoli;
- core, contenente l'insieme delle classi utili a rappresentare tutte le strutture in cui è suddivisa una partitura musicale. Classi come Note.java o Measure.java saranno di notevole importanza nelle fasi di implementazione delle modifiche;
- util, contenente una classe con vari metodi che permettono al convertitore di funzionare correttamente. Un esempio è un metodo il cui scopo è quello di decomprimere un file XML compresso;
- vocabulary, contenente l'insieme delle classi che hanno l'obiettivo di rappresentare le varie ontologie o vocabolari utilizzati da MusicOWL, andando a trasformare tutti i vari soggetti, predicati e oggetti rappresentanti le triple RDF in Property o Resource.

Nell'ultimo aspetto si nota come all'interno del convertitore venga utilizzata la libreria Apache Jena per leggere e scrivere grafi RDF in Java.
Sostanzialmente il convertitore prende in input dei file MusicXML 3.0 e li converte in uno stadio intermedio, andando a rappresentare la partitura musicale tramite classi Java. Troveremo quindi una istanza della classe MusicScore che conterrà all'interno dell'attributo parts di tipo array di ScorePart tutte le singole parti. Allo stesso modo ogni istanza di ScorePart conterrà tutti i movimenti al suo interno e così via fino ad arrivare alle note. Una volta ottenuta questa nuova rappresentazione dell'opera, esse verrà tradotta in un grafo RDF andan-

do a sfruttare strutture come il foreach per navigare nella partitura e creando per ogni aspetto musicale la corrispondente tripla RDF.
All'interno del traduttore stesso inoltre vengono forniti vari esempi di file sia come input (ovvero vari file in formato .xml) che come output (ad esempio in formato .ttl), che permettono di approcciare il lavoro del convertitore inizialmente come "black box", per poi passare ad un'analisi approfrondita del codice e di come i singoli aspetti vengono trattati.
Aspetto negativo del traduttore al momento è l'assenza di compatibilità col formato .musicxml, probabilmente per semplificare la prima traduzione in classi Java.

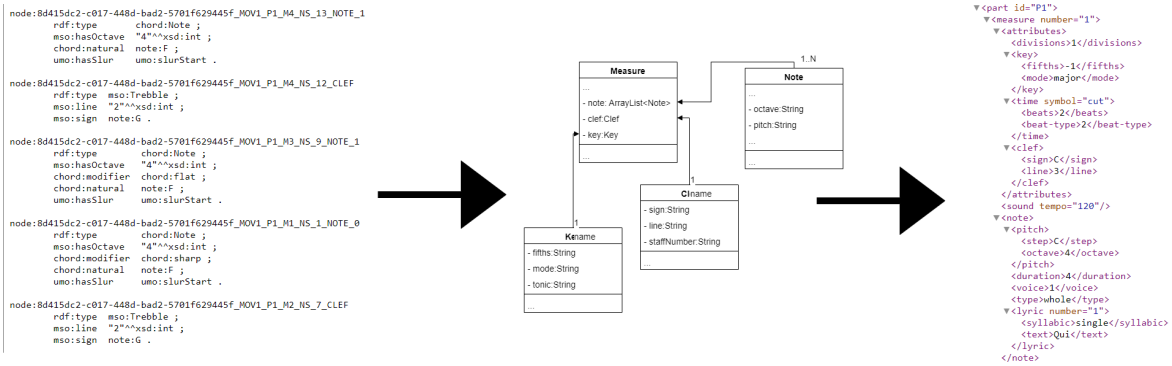

Figura 11: Funzionamento del traduttore, da MusicXML a classi Java a RDF

4.3 Eclipse

Eclipse è un IDE (Integrated Development Environment, ovvero ambiente di sviluppo integrato), multi-linguaggio e multipiattaforma ideato da un consorzio di colossi dell'informatica quali Ericsson, HP , IBM, Intel e distribuito gratuitamente sotto i termini della Eclipse Public License.
Come ogni IDE, Eclipse supporta lo sviluppo di codice in vari linguaggi, partendo dalla scrittura fino alle fasi di debugging e testing. Per riuscire a fare ciò fa uso di vari tool quali:

- un editor di testo in cui scrivere il codice;

- un compilatore, con cui tradurre il codice in codice macchina;
- un interprete, che esegue il codice senza tradurlo;
- un debugger, per identificare vari errori nella scrittura del codice e testare il progetto;
- un tool di building automatico, che permettono migliori prestazioni nella costruzione delle build per alcune fasi molto ripetitive e semplici;
- un sistema di analisi in tempo reale della sintassi, per cui l'IDE riesce a evidenziare le keyword del linguaggio che si sta usando.

Ripartendo dall'ultimo punto si può introdurre uno degli aspetti più interessanti di Eclipse, ovvero l'essere multipiattaforma. Andando ad inserire e togliere determinati plugin, Eclipse riesce a supportare scrittura, interpretazione e/o compilazione e testing di vari linguaggi, tra i quali Java, Python, C, Ruby e molti altri.

4.4 GraphDB

GraphDB è un DBMS per gestire grafi RDF. Esso implementa l'interfaccia del framework RDF4J (framework responsabile di archiviazione, query e analisi di dati RDF), il W3C SPARQL Protocol specification (documento che descrive il passaggio dalle query in SPARQL dal lato cliente al processore effettivo) e supporta tutti i formati di serializzazione RDF.
Alla base dell'utilizzo di GraphDB c'è la GraphDB Workbench (che per semplicità chiameremo Workbench), che è il tool web-based di amministrazione di GraphDB. Esso sfrutta un'intefaccia simile a quella di RDF4J Workbench Web Application, ma aggiunge molte funzionalità, come la possibilità di bloccare query che stanno richiedendo troppo tempo, l'import di server files e un maggior numero di formati disponibili per l'esportazione. La Workbench si compone principalmente di 4 parti:

- la sezione di import, in cui è possibile importare file sia basati sulla notazione RDF che in forma tabulare, quindi in formati come il .csv o .xls. Aspetto importante dell'importazione è il grafo a cui si andranno ad aggiungere i dati inseriti, che può essere quello fornito di default dal programma oppure altri, di cui si deve inserire il nome;
- la sezione di esplorazione, che permette di visualizzare i dati caricati da vari punti di vista: il primo di default, che inizialmente mostra a schermo i vari grafi e successivamente tutte le triple RDF contenute, una visualizzazione gerarchica delle classi e una in cui vengono messe alla luce le relazioni tra le classi;
- la sezione di SPARQL, in cui si ha a disposizione un editor di testo con syntax highlighter (sistema di evidenziamento delle parole chiavi, simile a quello introdotto per Eclipse) e che permette la scrittura, elaborazione e visualizzazione dei risultati di una query SPARQL;
- la sezione di monitor, che tiene traccia di tutte le query che sono state lanciate e che sono in elaborazione, con statistiche interessanti quali il testo della query stessa e il tempo dal lancio e la possibilità di abortire le query.

4.5 SPARQL

SPARQL è un linguaggio di interrogazione dati rappresentati in formato RDF. Come spiegato nell'articolo "Semantics and Complexity of SPARQL"[16] scritto da Jorge Pèrez, Marcelo Arenas e Claudio Gutierrez, una query SPARQL è formata da tre parti:

- la "pattern matching part", in cui si va ad applicare filtri ai valori, unire dei pattern o annidare dei controlli. Inoltre in questa parte si può scegliere la fonte in cui ricercare pattern;

- la "solution modifier part" in cui si applicano al risultato della prima parte operazioni come la proiezione, l'ordinamento e la limitazione nel numero delle triple accettate come risultato;
- la "output part" dove viene visualizzato il risultsto di una query, che può essere una lista di valori come un valore booleano.

A livello sintattico, una query SPARQL ha alcune similitudini con query SQL. Keyword come SELECT, WHERE, ORDER BY, GROUP BY e HAVING sono comuni ad entrambi i linguaggi. Le differenze sono molte e derivano dalla differenza che sta alla base dei linguaggi: essendo SQL basato su database relazionali la cui rappresentazione naturale sono tabelle, il focus di una query SQL sono le colonne delle tabelle che vogliamo visualizzare, mentre in una query SPARQL il risultato principalmente è un insieme di risorse rappresentate da variabili (riconoscibili perché nella forma ?nome) che devono apparire in triple RDF secondo le indicazioni date dalla clausola WHERE. Ad esempio, se vogliamo trovare tre risorse collegate da una proprietà, in SPARQL basterà creare 3 variabili, una per risorsa, e dichiarare le proprietà che legano le risorse nella WHERE, mentre in SQL per trovare 3 istanze collegate da un vincolo di chiave esterna bisognerà effettuare due join ed esplicitare il vincolo, rendendo il processo estremamente più complesso sia computazionalmente che nella visualizzazione.

```
SELECT ?window ?wall
WHERE{
  ?window a ifc:IfcWindow .
  ?window qrw:isPlacedIn ?wall .
  ?wall a ifc:IfcWall .
  FILTER NOT EXISTS {
    ?wall qrw:isContainedIn ?storey .
    ?window qrw:isContainedIn ?storey .
    ?storey a ifc:IfcBuildingStorey .
  }
}
```

Figura 12: Esempio di query SPARQL

```sql
SELECT customerName, customercity, customermail, ordertotal,salestotal
FROM onlinecustomers AS c
   INNER JOIN
   orders AS o
   ON c.customerid = o.customerid
   LEFT JOIN
   sales AS s
   ON o.orderId = s.orderId
   WHERE s.salesId IS NULL
```

Figura 13: Esempio di query SQL

5 L’estensione presentata

In questo capitolo verranno presentate tutte le voci che compongono l’integrazione all’ontologia MusicOWL che abbiamo sviluppato. Formalmente prende il nome di UNIVPM Music Ontology, il cui prefisso URI è "umo" e per ogni voce verrà descritto quale aspetto musicale viene rappresentato e in quale maniera integrando il ruolo che svolge nella rappresentazione RDF e le possibili relazioni con altre voci di MusicOW.

- **RISORSE**
  - **umo:trill** viene utilizzato per rappresentare un trillo;
  - **umo:fermata** viene utilizzato per rappresentare;
  - **umo:slurStart** viene utilizzato per rappresentare l’inizio di una legatura nella partitura musicale;
  - **umo:slurStop** viene utilizzato per rappresentare la fine di una legatura nella partitura musicale;
  - **umo:crescendo** viene utilizzato per rappresentare uncrescendo nella partitura musicale;
  - **umo:diminuendo** rappresenta un diminuendo nella partitura musicale.
- **PROPRIETÀ**
  - **umo:hasTrill** rappresenta la presenza di un trillo come abbellimento di una nota, è una proprietà della risorsa chord:Note e accetta come valore una risorsa di tipo umo:trill;
  - **umo:hasFermata** rappresenta la presenza di una corona come abbellimento di una nota, è una proprietà della risorsa chord:Note e accetta come valore una risorsa di tipo umo:fermata;

- **umo:hasSlur** rappresenta la presenza di una legatura che coinvolge la nota interessata, è una proprietà della risorsa chord:Note e accetta come valore una risorsa di tipo umo:slurStart o una di tipo umo:slurStop. Nel primo caso si descrive la nota di inizio legatura, nel secondo la nota di fine legatura;
- **umo:hasWedge** rappresenta la presenza di una dinamica nella battuta, è una proprietà della risorsa mso:Measure e accetta come valore una risorsa di tipo umo:crescendo o di tipo umo:diminuendo;
- **umo:hasText** rappresenta la presenza di testo legato alla singola nota, è una proprietà della risorsa chord:Note e accetta come valore una stringa di testo.
- **umo:textToSing** rappresenta la presenza di testo nell'opera, è una proprietà della risorsa mo:Score e accetta come valore una stringa di testo.

Ad esempio, nella figura 14 mostriamo una serie di note con una legatura e subito dopo la rappresentazione rdf corrispondente delle singole note

Figura 14: Esempio di legatura

node:60528b09-2c26-407f-b005-52b2eb6f56f1_MOV1_P3_M2_NS_23_NOTE_7
rdf:type chord:Note ;
mso:hasOctave "4"8sd:int ;
chord:modifier chord:flat ;
chord:natural note:A ;
umo:hasSlur umo:slurStart .

node:60528b09-2c26-407f-b005-52b2eb6f56f1_MOV1_P3_M2_NS_24_NOTE_8
rdf:type chord:Note ;
mso:hasOctave "4"8sd:int ;
chord:modifier chord:flat ;
chord:natural note:B .

node:60528b09-2c26-407f-b005-52b2eb6f56f1_MOV1_P3_M2_NS_25_NOTE_9
rdf:type chord:Note ;
mso:hasOctave "4"8sd:int ;
chord:natural note:B .

node:60528b09-2c26-407f-b005-52b2eb6f56f1_MOV1_P3_M2_NS_26_NOTE_10
rdf:type chord:Note ;
mso:hasOctave "5"8sd:int ;
chord:natural note:C ;
umo:hasSlur umo:slurStop .

6 Progettazione e implementazione modifiche traduttore

6.1 Progettazione modifiche traduttore

In questa fase principalmente si divideranno e espliciteranno i vari aspetti su cui si andrà a lavorare.

1. Inizialmente si andrà ad integrare l'estensione qui proposta all'interno del convertitore creando una classe apposita in cui si andranno ad inserire proprietà e risorse di cui si è discusso nel capitolo precedente.
2. Successivamente si modificheranno le classi Note.java e Measure.java, affinché riescano a catturare anche gli aspetti di nostro interesse.
3. In seguito si andrà a mettere mano alla prima conversione, ovvero da MusicXML a Java: modificando una funzione del convertitore, ovvero "createMusicScoreDocument", si potranno raccogliere più dati dai formati MusicXML e convertirli in attributi delle classi sopra citate.
4. Infine si andrà a mettere mano alla seconda conversione, ovvero da Java a RDF: modificando un'altra funzione del convertitore, ovvero "createRDF", si potranno riportare tutti i dati contenuti nelle classi Java nello schema RDF, quindi comprendere anche quelli implementati dall'ontologia UNIVPM Music Ontology.

6.2 Implementazione modifiche traduttore

In questo capitolo seguiremo i punti mostrati nella progettazione e si discuteranno in maniera più approfondita le modalità di lavoro:

1. Per integrare l'estensione abbiamo creato una classe nella repository scr/main /java/de/wwu/music2rdf/vocabulary col nome UNIVPMMusicOntology.java. Sfruttando il framework Jena Apache le proprietà sono diventate degli attributi di tipo Property mentre le risorse attributi di tipo Resource. Inoltre tutti gli attributi sono contrassegnati dalla parola chiave "final" al fine di evitare la sovrascrizione di una delle proprietà. Nella figura seguente viene mostrato il codice della classe.

```
1  package de.wwu.music2rdf.vocabulary;
2
3  import org.apache.jena.rdf.model.Model;
7
8  public class UNIVPMMusicOntology {
9      public static final String NS="https://kdmg.dii.univpm.it/music/ontology#";
10     private static Model univpmmusicontology = ModelFactory.createDefaultModel();
11
12     public static final Property hasText=univpmmusicontology.createProperty(NS+"hasText");
13     public static final Property hasTrill=univpmmusicontology.createProperty(NS+"hasTrill");
14     public static final Property HasFermata=univpmmusicontology.createProperty(NS+"hasFermata")
15     public static final Property HasSlur=univpmmusicontology.createProperty(NS+"hasSlur");
16     public static final Property HasWedge=univpmmusicontology.createProperty(NS+"hasWedge");
17     public static final Property HasTextToSing=univpmmusicontology.createProperty(NS+"hasTextTo
18
19     public static final Resource Trill=univpmmusicontology.createResource(NS+"trill");
20     public static final Resource Fermata=univpmmusicontology.createResource(NS+"fermata");
21     public static final Resource SlurStart=univpmmusicontology.createResource(NS+"slurStart");
22     public static final Resource SlurStop=univpmmusicontology.createResource(NS+"slurStop");
23     public static final Resource Crescendo=univpmmusicontology.createResource(NS+"crescendo");
24     public static final Resource Diminuendo=univpmmusicontology.createResource(NS+"diminuendo")
25 }
26
```

Figura 15: codice di UNIVPMMusicOntology.java

2. Per maggiore chiarezza divideremo il lavoro fatto su Note.java e Measure.java:

   - **Note.java**

In Note.java si ha bisogno di tenere tracca del testo della singola nota, la presenza di un trillo e di una corona. Per il primo si utilizza un attributo di tipo String che tiene conto anche della sillabazione, che ricordiamo essere inserita nel tag <syllabic>. Nel caso in cui il testo sia la parte iniziale o intermedia di una parola il testo sarà inserito normalmente mentre se il testo è la parte finale di una parola o una parola intera, alla fine della sillaba verrà inserito uno spazio. Per gli ultimi due attributi si avranno una variabile booleana ciascuno per rappresentare la presenza o meno del trillo e della corona. Inoltre sono stati implementati metodi getter e setter per i 3 attributi. Le modifiche nello specifico sono mostrate nella figura 16.

```
private String text;
private boolean fermata;
private boolean trill;
public boolean isTrill() {
    return trill;
}

public void setTrill(boolean trill) {
    this.trill = trill;
}


public boolean isFermata() {
    return fermata;
}

public void setFermata(boolean fermata) {
    this.fermata = fermata;
}

public String getText() {
    return text;
}

public void setText(String text) {
    this.text = text;
}
```

Figura 16: Modifiche alla classe Note.java

- **Measure.java**

  In Measure.java si ha bisogno di tenere traccia delle dinamiche. Dato che non si ha la certezza che ci possa essere solamente una dinamica per battuta, come mostrato nella figura 9, queste vengono rappresentate come un array di stringhe contenenti solamente due valori: "crescendo" o "diminuendo", andando quindi a rappresentare in forma te-

stuale il tipo di dinamica presente. Inoltre è stato introdotto il metodo getter per ottenere l'array e poi effettuare sia operazioni di inserimento che di lettura.

Come si può notare, non è presente nessuna rappresentazione del testo intero, poiché verrà ricostruito nella fase finale partendo dai testi delle note. Inoltre nel codice della classe Note.java era già presente una rappresentazione per le legature soddisfacente(sotto forma di stringa contenente i termini "middle", "start" o "stop), dunque non è stata necessaria nessuna implementazione.

3. Per ogni aspetto trattato nel punto precedente(quindi escludendo il testo completo) sono state inserite alcune righe di codice al fine di popolare, quando necessario, gli attributi inseriti nel punto 2. Nello specifico:

   - **testo**: Per catturare il testo e inserirlo in un oggetto di tipo Note è sufficiente inserire nel ciclo for in cui vengono iterate le caratteristiche delle note presenti nel file MusicXML un blocco condizionale che confermi la presenza del testo. In caso negativo non accade nulla, mentre se risulta presente del testo si va a controllare il tipo di sillaba e si seguono le indicazioni date nel punto precedente. Lo stesso discorso è valido anche per corone e trilli, in cui non avviene il secondo check ma viene inserito il valore true ai rispettivi attributi. Nell'immagine qui sotto il codice di quello che abbiamo appena descritto.

```
if(elementNotes.getElementsByTagName("text").item(0)!=null)
    if(elementNotes.getElementsByTagName("syllabic").item(0).getTextContent().equals("begin")||
            elementNotes.getElementsByTagName("syllabic").item(0).getTextContent().equals("middle"))
        note.setText(elementNotes.getElementsByTagName("text").item(0).getTextContent());
    else
        note.setText(elementNotes.getElementsByTagName("text").item(0).getTextContent()+" ");
if(elementNotes.getElementsByTagName("fermata").item(0)!=null) note.setFermata(true);
if(elementNotes.getElementsByTagName("trill-mark").getLength()>0) note.setTrill(true);
```

Figura 17: Alcune modifiche al metodo "createMusicScoreDocument"

- Per quanto riguarda le dinamiche è stato sufficiente inserire all'interno del ciclo che itera sulle caratteristiche delle battute un'istruzione condizionale che controlli se la caratteristica che si sta analizzando è una dinamica. In caso negativo non accade nulla, mentre in caso positivo si popola l'array dell'oggetto di tipo Measure come descritto nel punto precedente.

Anche in questo caso non sono stati trattati testo completo e legature, per la ragione spiegata nel punto precedente, mentre per le legature è già presente il sistema di cattura da MusicXML.

4. La funzione "parseRDF" ha ricevuto varie modifiche, anche in questo caso dividibili in base alla tipologia di aspetto:

   - **Testo, legature, corone e trilli**: Nel ciclo in cui vengono iterate le note presenti in una battuta sono stati inseriti un'istruzione condizionale per aspetto al fine di capire, per ogni nota, se presentasse testo, legature corone e/o trilli. Per chiarezza in questo punto tratteremo il testo, ma lo stesso ragionamento sarà valido per gli altri aspetti con delle variazioni riguardo il valore della tripla RDF. Nel caso in cui il testo non fosse presente non accadrebbe nulla, mentre in caso positivo sarebbe stata creata una tripla RDF in cui il soggetto sarebbe stata la nota presa in considerazione, predicato la proprietà dell'estensione relativa, quindi umo:hasText e valore la stringa restituita dal metodo getter "getText". Nell'immagine successiva è presentato il codice che implementa quello che è stato descritto per ogni aspetto;

```
if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getText()!=null) {
    model.add(model.createLiteralStatement(resNote, UNIVPMMusicOntology.hasText, score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getText()));
    text+=score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getText();
}
if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).isFermata()) {
    model.add(model.createLiteralStatement(resNote,  UNIVPMMusicOntology.HasFermata, UNIVPMMusicOntology.Fermata));
}
if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).isTrill()) {
    model.add(model.createLiteralStatement(resNote,  UNIVPMMusicOntology.hasTrill, UNIVPMMusicOntology.Trill));
}
if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getSlurElementType()!=null) {
    if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getSlurElementType().equals("start")) {
    model.add(model.createLiteralStatement(resNote, UNIVPMMusicOntology.HasSlur, UNIVPMMusicOntology.SlurStart));
}
    else if(score.getParts().get(i).getMeasures().get(j).getNotes().get(k).getSlurElementType().equals("stop")){
        model.add(model.createLiteralStatement(resNote, UNIVPMMusicOntology.HasSlur, UNIVPMMusicOntology.SlurStop));
    }
}
```

Figura 18: Alcune modifiche apportate alla funzione "parseRDF"

- **Testo completo**: Nella figura 18, nel codice dell'if per controllare la presenza del testo appare una variabile text a cui viene concatenato il testo della nota corrente. Quella variabile viene utilizzata per inserire il testo completo attraverso una tripla RDF che prende come soggetto la partitura, come predicato umo:hasTextToSing e come valore il valore della variabile text al termine di tutti i cicli che iterano sulle parti, battute e note.
- **Dinamiche**: Nel ciclo in cui vengono iterate le battute presenti in una parte, viene inserito un'istruzione condizionale per controllare se l'array delle dinamiche non è vuoto. In caso positivo, allora viene creato un ciclo sull'array al fine di inserire tutte le dinamiche all'interno del grafo RDF con triple il cui soggetto è la battuta, predicato umo:hasWedge e valore la risorsa corrispondente al valore testuale presente nell'array, in caso negativo non accade nulla.

A questo punto il convertitore è perfettamente in grado di effettuare il lavoro che faceva precedentemente includendo inoltre tutti gli aspetti trattati dall'estensione proposta.

7 Interrogazioni di una partitura

Al termine del lavoro svolto il traduttore è stato utilizzato e sono state create partiture in formato RDF che comprendessero l'estensione proposta. Su di esse sono state provate delle query per convalidare e verificare il lavoro svolto. In questo capitolo saranno presentate tutte le query che sono state testate sui file offerti dal traduttore stesso a seguito della modifica mostrata nel capitolo precedente. Le interrogazioni devono essere prese come step iniziale per capire come ottenere risultati interessanti sfruttando l'estensione proposta, e per questo motivo abbiamo deciso che per ogni query ritornerà lo stesso tipo di variabili, ovvero il nome dell'opera e, nelle situazioni in cui è stato reputato utile, il numero della battuta. L'unica eccezione è la query numero 3.

1. Ottenere tutte le opere con almeno una corona
   ```
   select distinct ?title where {
   ?score dc:title ?title. ?score mo:movement ?movement.
   ?movement mso:hasScorePart ?scorepart.
   ?scorepart mso:hasMeasure ?measure.
   ?measure mso:hasNoteSet ?noteset.
   ?noteset mso:hasNote ?note.
   ?note umo:hasFermata umo:fermata.
   }
   ```

2. Ottenere tutte le partiture con almeno una battuta con crescendo e una nota con corona
   ```
   select distinct ?title where{
   ?score mo:movement ?movement. ?score dc:title ?title.
   ?movement mso:hasScorePart ?scorepart.
   ?scorepart mso:hasMeasure ?measure.
   ?measure umo:hasWedge umo:crescendo.
   ```

?measure mso:hasNoteSet ?noteset.
?noteset mso:hasNote ?note.
?note umo:hasFermata umo:fermata.
}

3. Ottenere tutto il testo ordinato di un determinato brano (sostituire il titolo alla variabile title)

select ?text where {
?score dc:title ?title. ?score mo:movement ?movement.
?movement mso:hasScorePart ?scorepart.
?scorepart mso:hasMeasure ?measure.
?measure dbo:order ?order.
?measure mso:hasNoteSet ?noteset.
?noteset mso:hasNote ?note.
?note umo:hasText ?text.
}
order by xsd:integer(?order) ?noteset

4. Ottenere tutte le partiture e le relative battute al cui interno inizia e finisce una legatura

select distinct ?title ?number where{
?score mo:movement ?movement. ?score dc:title ?title.
?movement mso:hasScorePart ?scorepart.
?scorepart mso:hasMeasure ?measure.
?measure mso:hasNoteSet ?noteset1.
?measure mso:hasNoteSet ?noteset2.
?measure dbo:order ?number.

?noteset1 mso:hasNote ?note1.
?noteset2 mso:hasNote ?note2.
?note1 umo:hasSlur umo:slurStart.
?note2 umo:hasSlur umo:slurStop.
?noteset1 mso:nextNoteSet+ ?noteset2.
}
order by xsd:integer(?number)

5. Ottenere tutte le partiture con all'interno una parola o frase desiderata (inserire l'espressione all'interno degli apici doppi)
   select distinct ?score where{
   ?score umo:hasTextToSing ?text
   filter regex(?text, "")
   }

6. Ottenere tutte le partiture con la seguente battuta e stamparne anche il numero
   select ?title ?number where{
   ?score dc:title ?title. ?score mo:movement ?movement.
   ?movement mso:hasScorePart ?scorepart.
   ?movement mso:hasTime ?time.
   ?time mso:hasBeatType "4".
   ?time mso:hasBeats "4".
   ?scorepart mso:hasMeasure ?measure.
   ?measure dbo:order ?number.
   ?measure mso:hasNoteSet ?noteset1.
   ?measure mso:hasNoteSet ?noteset2.
   ?measure mso:hasNoteSet ?noteset3.
   ?noteset1 mso:hasNote ?note1.
   ?noteset1 mso:nextNoteSet ?noteset2.

?noteset1 mso:hasDuration ?duration1.
?noteset1 mso:hasClef ?clef.
?clef a mso:Trebble.
?duration1 a mso:Quarter.
?note1 mso:hasOctave "4".
?note1 chord:natural note:G.
?note1 umo:hasText "De".
?noteset2 mso:hasNote ?note2.
?noteset2 mso:nextNoteSet ?noteset3.
?noteset2 mso:hasDuration ?duration2.
?noteset2 mso:hasClef ?clef.
?duration2 a mso:Quarter.
?note2 mso:hasOctave "5".
?note2 chord:natural note:B.
?note2 umo:hasText "si".
?noteset3 mso:hasNote ?note3.
?noteset3 mso:hasDuration ?duration3.
?noteset3 mso:hasClef ?clef.
?duration3 a mso:Half.
?note3 mso:hasOctave "4".
?note1 chord:natural note:D.
?Note1 umo:hasText "derium ".
}

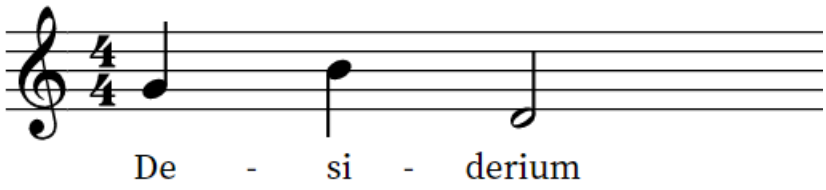

Figura 19: Battuta cui fa riferimento la query numero 6

Riprendendo la query numero 6, è risultato interessante automatizzare il passaggio da una forma grafica come quella presentata ai vincoli della query, ovvero il contenuto della clausola WHERE. Esistono online molti strumenti gratuiti, come MuseScore, che permettono la scrittura su pentagramma e, terminata la propria opera, producono un file in vari formati, tra cui il MusicXML. Quindi si è deciso di produrre un'ulteriore estensione del traduttore che permetta, dato sempre in ingresso un file MusicXML, di ottenere in output una query che cerchi tutte le opere, con riferimento alla battuta da cui inizia la corrispondenza, che contengono esattamente quanto descritto nel file di input. Possiamo prendere come esempio la query numero 6, che è stata scritta utilizzando proprio questo mezzo. Dando la rappresentazione in MusicXML della figura 19 come input, il traduttore è stato in grado di produrre la query che cerca tutte le opere che contengono quella battuta, andano a descriverla nella clausola WHERE.
Per implementare questa modifica è stato aggiunto un parametro alla chiamata, denominato "query", che accetta valori booleani e di default prende valore negativo. Se impostato a falso il traduttore produrrà un grafo RDF, quindi svolgerà il lavoro che abbiamo già descritto e su cui sono state apportate le prime modifiche, mentre se "query" avrà come valore vero, allora produrrà la query nelle modalità descritte. Lo strumento attualmente sviluppato non è molto potente, in quanto all'utente non è data la possibilità di decidere a monte il contenuto della clausola SELECT, ma è una prima idea interessante che potrebbe permette un maggiore utilizzo del traduttore, e di conseguenza dell'ontologia MusicOWL.

8 Conclusioni

Al termine dell'opera svolta possono essere fatte varie considerazioni:

- L'attuale stato dell'arte nella rappresentazione delle partiture volte all'analisi e all'approfondimento è indietro rispetto agli altri settori, sia per un minore interesse, poiché le partiture musicali hanno una minore popolarità rispetto a quadri o opere di scultura, ed avendo correnti interne che spingono verso direzioni differenti non sembra potersi riallineare nel breve periodo. Nonostante ciò le spinte divergenti stanno permettendo uno sviluppo più orizzontale, infatti da una parte troviamo l'ambiente MIDI-HaMSE che riesce a catturare in maniera efficace partiture con lo scopo della riproduzione mentre l'ambiente MusicXML-MusicOWL, con i limiti già esplicitati, permette di ottenere un'efficace rappresentazione in formato RDF delle opere, e quindi a possibilità di analisi della partitura efficaci sia da un punto di vista dei metadati che del lato musicale.
- L'integrazione sviluppata allo strumento offerto dal team di MusicOWL permette attualmente di svolgere attività di analisi abbastanza avanzate sia dal punto di vista dei dati musicali che dei metadati. La grande potenza dello strumento è sicuramente dovuta all'utilizza dei grafi RDF, che rispondono nella miglior maniera alla rappresentazione di partiture musicali, ma possono essere svolti ulteriori lavori di miglioramento del mezzo, come una documentazione più chiara degli aspetti trattati, ma parte dei problemi è anche dovuta ad uno scarsa popolarità del mezzo. Inoltre quest'opera, in quanto svolta da un non esperto di teoria musicale, potrebbe aver tralasciato aspetti delle partiture che necessiterebbero di un'ulteriore integrazione per poter essere catturati e di conseguenza immagazzinati nelle digital repository.

Riferimenti bibliografici

[1] Katrin Braunschweig et al. «The state of open data». In: *Limits of current open data platforms* (2012).

[2] Charles E Collyer, Seth S Boatright-Horowitz e Sari Hooper. «A motor timing experiment implemented using a musical instrument digital interface (MIDI) approach». In: *Behavior Research Methods, Instruments, & Computers* 29.3 (1997), pp. 346–352.

[3] Michael Good. «MusicXML for notation and analysis». In: *The virtual score: representation, retrieval, restoration* 12.113-124 (2001), p. 160.

[4] Jess Hemerly. «Making metadata: The case of MusicBrainz». In: *Available at SSRN 1982823* (2011).

[5] Pascal Hitzler e Krzysztof Janowicz. «Linked Data, Big Data, and the 4th Paradigm.» In: *Semantic Web* 4.3 (2013), pp. 233–235.

[6] «http://neuma.huma-num.fr/home/presentation». In.

[7] «https://digital.library.ucla.edu/sheetmusic/aboutProject.html». In.

[8] «https://imslp.org/wiki/IMSLP:Goals». In.

[9] «https://music-encoding.org/about/». In.

[10] «https://www.go-fair.org/fair-principles/». In.

[11] «https://www.w3.org/RDF/». In.

[12] Seonhee Hwang et al. «Optical measurements of paintings and the creation of an artwork database for authenticity». In: *Plos one* 12.2 (2017), e0171354.

[13] Ruichen Jin e Jongweon Kim. «Artwork Identification method and database construction». In: *Proceedings of the 2017 International Conference on Information Technology*. 2017, pp. 112–115.

[14] Rob Kitchin. *The data revolution: Big data, open data, data infrastructures and their consequences*. Sage, 2014.

[15] Albert Meroño-Peñuela et al. «The MIDI linked data cloud». In: *International Semantic Web Conference*. Springer. 2017, pp. 156–164.

[16] Jorge Pérez, Marcelo Arenas e Claudio Gutierrez. «Semantics and Complexity of SPARQL». In: *International semantic web conference*. Springer. 2006, pp. 30–43.

[17] Andrea Poltronieri e Aldo Gangemi. «The HaMSE Ontology: Using Semantic Technologies to support Music Representation Interoperability and Musicological Analysis». In: *arXiv preprint arXiv:2202.05817* (2022).

[18] Sabbir M Rashid, David De Roure e Deborah L McGuinness. «A music theory ontology». In: *Proceedings of the 1st International Workshop on Semantic Applications for Audio and Music*. 2018, pp. 6–14.

[19] Magherini Simone. *BIL Bibliografia Informatizzata Leopardiana 1815-1999: manuale d'uso ver. 1.0*. Firenze University Press, 2003.

[20] Avery Wang et al. «An industrial strength audio search algorithm.» In: *Ismir*. Vol. 2003. Citeseer. 2003, pp. 7–13.

[21] Marcin Wylot et al. «RDF data storage and query processing schemes: A survey». In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–36.