Universidad
Carlos III de Madrid

Departamento de Ingeniería Telemática

PROYECTO FIN DE CARRERA
Ingeniería de telecomunicación

Análisis armónico de composiciones musicales

Autor: Francisco Javier Rodríguez Donado
Tutor: Raquel Mª Crespo García

Leganés, Julio de 2010

TÍTULO: *Análisis armónico de composiciones musicales*

AUTOR: Francisco Javier Rodríguez Donado

TUTOR: Raquel Mª Crespo García

EL TRIBUNAL

Presidente: _____

Secretario: _____

Vocal: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de Julio de 2010 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

Presidente **Secretario** **Vocal**

A mis padres

Cuando no me ve nadie, como ahora, gusto de imaginar a veces si no será la música la única respuesta posible para algunas preguntas.
Antonio Buero Vallejo

Resumen

Desde hace ya varias décadas han surgido numerosos proyectos y estudios que, de diversas maneras, crean puntos de unión entre la música y la computadora.

Algunos de estos trabajos toman como entrada de datos las propias partituras musicales, es decir, información simbólica sobre la obra musical a analizar. Mediante el desarrollo de algoritmos específicos, llegan a conclusiones de lo más variadas, como puedan ser clasificaciones en torno a distintos criterios o la obtención de melodías principales, por mencionar algunos ejemplos.

Este proyecto entra dentro de este grupo, ya que tiene por objeto realizar una aplicación de análisis de partituras musicales basado en un conocimiento sobre armonía musical. Así, se logra que la computadora llegue a conclusiones *inteligentes* sobre la obra musical, es decir, resultados similares a aquellos que tendría una persona con conocimientos específicos de armonía musical. Además, tras la realización del análisis, la aplicación genera arreglos de forma automática, es decir, altera la progresión armónica detectada, proponiendo una progresión distinta pero equivalente a la original.

Palabras clave: armonía musical, acorde, intervalo, partitura, arreglo musical, progresión armónica, melodía, inteligencia artificial, sistemas expertos, sistemas basados en reglas, Drools, Java, MusicXML.

Abstract

For several decades, numerous projects and studies have emerged creating, in various ways, connection points between music and computational science.

Some of these works take sheet music as input data, that is to say, symbolic information on the musical work to analyze. Through the implementation of specific algorithms, they can get many different results, such as various classifications or extraction of main melodies just to mention some examples.

This project falls within this group, since it carries out an analysis of music scores based on knowledge of musical harmony. Thus, it makes the computer come to conclusions about the musical work, which are close to those from a person with special knowledge of musical harmony. In addition, after the analysis, the computer automatically generates arrangements, in other words, it alters the harmonic progression detected, suggesting a different but equivalent progression to the original one.

Keywords: musical harmony, musical score, chord, interval, melody, musical arrangement, harmonic progression, artificial intelligence, expert systems, rule-based systems, Drools, Java, MusicXML.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Introducción y motivación

A lo largo de las últimas décadas, han surgido numerosos proyectos comerciales, así como estudios académicos, que relacionan de múltiples maneras la informática con la música [1], [4]. Entre los primeros destacan aquellas aplicaciones que explotan el potencial de la computadora como una herramienta que simplifica y facilita determinadas tareas, como pueden ser la grabación, la reproducción, la transcripción musical, la composición o la producción de trabajos musicales. Por otro lado, también han surgido estudios que, en vez de utilizar al ordenador como una mera herramienta que facilite la realización de ciertas tareas, buscan que éste presente resultados *inteligentes*, propios de una persona con un conocimiento y entrenamiento específicos en alguna rama de la música [3]. El proyecto que aquí se describe pertenece a este último grupo.

En esta memoria, se describe detalladamente un proyecto software que ha consistido en el desarrollo de una aplicación que, basándose en un conocimiento extraído de la teoría musical (concretamente teoría sobre armonía), analiza partituras musicales extrayendo información sobre la progresión armónica[1] subyacente, la cual no es obvia en una primera lectura para una persona que no posea conocimientos específicos sobre armonía. Por otro lado, además del desarrollo del motor de análisis armónico, también se ha desarrollado un módulo que hace uso de los resultados obtenidos por el motor de análisis para generar arreglos de forma automática, esto es, alterar la secuencia de acordes reconocida eliminando, sustituyendo o añadiendo nuevos acordes, de forma que imita la labor de un arreglista musical experimentado.

Dado que no se presupone en el lector conocimiento alguno sobre teoría musical, para que una persona no iniciada pueda comprender en qué consiste extraer la progresión armónica, y con objeto también de clarificar exactamente qué conocimientos teóricos han sido los que se han seleccionado para la aplicación de análisis, lo primero que se hará será detallarlos en un apartado específico sobre teoría musical (apartado 2.1) para, posteriormente, explicar cómo la aplicación los ha incorporado.

Para el desarrollo de la aplicación, en lo que respecta a recursos materiales ha sido necesario un ordenador personal que disponía del entorno de desarrollo Eclipse, así como conexión a Internet, el paquete Office de Microsoft para la documentación, y el programa Microsoft Project para la planificación. En lo que respecta a los recursos humanos, sólo se ha dispuesto de un ingeniero de telecomunicación, que ha desempeñado los distintos roles necesarios para el diseño y la implementación de la aplicación, así como las pruebas y la documentación de la misma. Tanto las fases de desarrollo como los recursos y los roles necesarios para el proyecto son descritos en detalle en el apartado 5.

Por último, la motivación del proyecto es, por un lado, mostrar de forma automatizada e inteligente una información fundamental para comprensión teórica de la obra musical, la cual puede ser además de gran utilidad al intérprete humano de la obra

[1] Se entiende por progresión armónica subyacente a la secuencia de acordes que sirven de base armónica para una obra musical dada. Consultar apartado 2.1 para más información.

en cuestión. Por otro lado, intenta emular la labor de un arreglista experimentado mediante la modificación automática de la progresión armónica detectada. Además de todo esto, pretende servir como punto de partida para poder en realizar en el futuro una serie de proyectos, los cuales se detallarán en el apartado *6.Conclusiones*, que realicen estudios automatizados sobre partituras musicales, bien sea con objeto de profundizar en el análisis teórico de las composiciones o bien tengan por objetivo deducir cualquier otro dato sobre las mismas (y para ello se apoyen en el conocimiento que se extrae del motor de análisis).

1.2. Objetivos

El objetivo fundamental de la aplicación de análisis musical es el desarrollo de un motor de análisis que sirva como punto de encuentro entre la obra, la computadora y la teoría musical, así como un módulo que, valiéndose del análisis armónico obtenido realice arreglos musicales, es decir, proponga una progresión armónica distinta pero equivalente a la detectada por el motor de análisis armónico.

Los datos de entrada para la aplicación estarán contenidos en una partitura en formato simbólico, es decir, no mediante una mera imagen o mapa de bits, sino mediante un formato de representación específico para partituras, a ser posible estandarizado, que contenga información sobre todos los elementos de la partitura (notas, ligados, alteraciones, duraciones de las notas, compás, silencios, anotaciones…) para, mediante un sistema de representación, poder mostrarla en pantalla. Un ejemplo de la entrada de datos sería la partitura que se muestra a continuación:

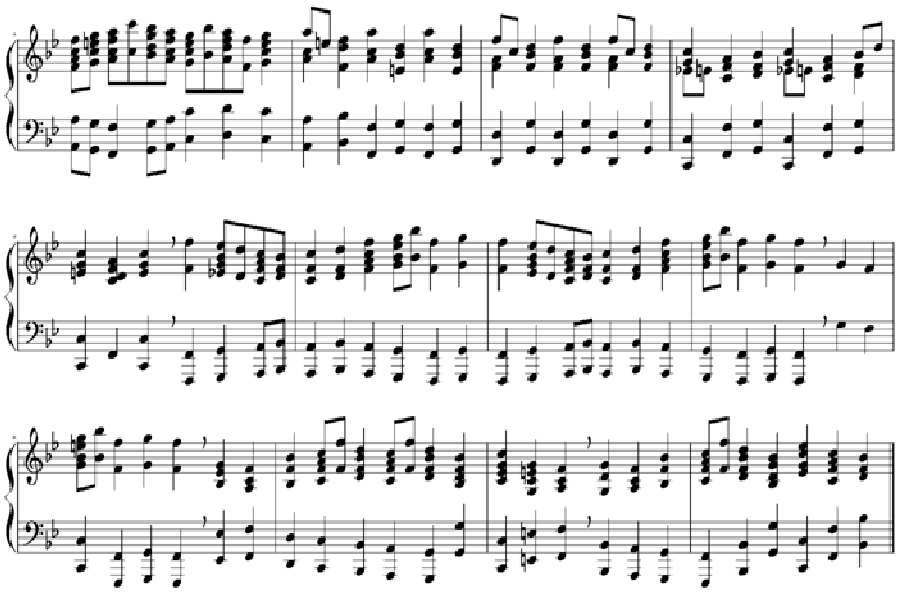


Figura 1 – Ejemplo de entrada de datos

Los requisitos necesarios para que esta partitura sea adecuada para nuestra aplicación es que los compases estén bien definidos, se disponga de la armadura de clave[2] y la obra esté convenientemente transcrita, ya que una partitura incompleta o de

[2] Todos los conceptos relacionados con la teoría musical en los que se basa este documento son explicados en el apartado *2.1 Teoría musical*

mala calidad (como la que puede extraerse de un archivo MIDI sin cuantizar ni preparar) puede dar lugar a un análisis incompleto o incluso incorrecto.

Con estos datos de entrada, la aplicación de análisis debe elaborar como salida una nueva partitura igual a la de entrada, pero con información añadida sobre la progresión armónica (o sucesión de acordes) en la que la obra se apoya, además de un análisis en formato de texto sobre dicha progresión armónica desde un punto de vista teórico musical. La información sobre la armonía que el motor de análisis deduzca, debe ser fácilmente accesible por parte de aplicaciones externas para que éste pueda ser utilizado en el desarrollo de proyectos futuros, por lo que deberá diseñarse una API que lo permita. Además, como se mencionó en el anterior punto, se ha desarrollado también un módulo integrado en la propia aplicación que toma los datos proporcionados ésta y los utiliza para realizar arreglos de forma automática. Esto se describirá detalladamente en los apartados 2.1 (en lo que respecta al aspecto teórico) y 3.4.7 (en lo que respecta a la implementación de dicho módulo).

De forma esquemática, podemos concluir que los objetivos que se marcaron al comienzo del desarrollo fueron los siguientes:

- Diseñar una aplicación que realice un análisis armónico de una pieza musical de forma que el resultado se parezca lo más posible al análisis que un ser humano con los conocimientos adecuados realizaría. Para ello hace falta:
    - Crear un sistema capaz de acceder y manipular archivos con partituras en un formato estandarizado para la representación de las mismas.
    - Identificar la sucesión armónica subyacente de la obra a analizar.
    - Actualizar la partitura con la nueva información.
- Realizar un análisis automático de la progresión armónica detectada aplicando conceptos de teoría musical.
- Ofrecer la posibilidad de realizar arreglos de forma automática sobre la progresión armónica obtenida por el motor de análisis.
    - Sería deseable que estos arreglos fueran lo más fieles posible a aquellos que haría un músico experimentado.
- El análisis armónico, así como los arreglos deben ser adecuados para una gran cantidad de obras musicales occidentales de música clásica, así como para obras populares (pop, rock, folk, funk…)[3].
- Facilitar el acceso a la información deducida por aplicaciones futuras, mediante una API amigable.
    - Además, se ha optado por desarrollar la aplicación enteramente en inglés, para hacerla accesible a un mayor número de desarrolladores en todo el mundo.

1.3. Estructura de la memoria

La memoria está estructurada de la siguiente manera:

- En el primer apartado tras la introducción, *2. Estado del arte*, se estudian los conceptos teóricos en los que se va a basar el análisis armónico

[3] No obstante, no se pretende conseguir un análisis concienzudo para el caso de obras de gran complejidad armónica, como sucede por ejemplo en algunas composiciones clásicas o de estilo jazz.

(subapartado *2.1 Teoría musical*). Después, se repasan algunos de los trabajos previos a este proyecto que han tenido una temática similar (subapartado *2.2. Trabajos previos*). A continuación, se muestran algunas de las tecnologías que pueden ser útiles como herramientas con las que desarrollar la aplicación dentro de *2.3. Recursos tecnológicos*, así como la justificación de las tecnologías escogidas.

- En el apartado *3. Descripción del sistema*, se describe el sistema que se ha desarrollado, primero mediante diagramas de su arquitectura para, a continuación, detallar el formato de la información de entrada (*3.2.* Datos de entrada) y describir la API desarrollada (*3.3. API de la aplicación*). Después, en el apartado *3.4. Descripción de los* procesos, se describe en detalle el funcionamiento interno de la aplicación.
- En el apartado *4.* Evaluación, se realiza la evaluación del funcionamiento de la aplicación mediante una serie de pruebas.
- En el apartado *5. Presupuesto*, se detalla la planificación de las tareas y el presupuesto total del proyecto.
- En el apartado *6. Conclusiones*, se repasan las principales conclusiones tras el desarrollo del proyecto, y se habla sobre posibles trabajos futuros.
- Al final del documento están la *Bibliografía* y los *Anexos*. Dentro de los *Anexos* se incluyen el *Manual de usuario* y el *Manual de instalación.*

2. Estado del arte

2.1. Teoría musical

En este apartado, se detallan los conceptos teóricos que serán necesarios para comprender el funcionamiento del motor de análisis armónico, así como para el módulo que genera los arreglos.

2.1.1. Conceptos básicos

Armonía: En música, la armonía estudia la percepción del sonido en forma vertical, es decir, simultánea en el tiempo. Por ello, en la música occidental, la armonía estudia el encadenamiento de notas simultaneas, las cuales se organizan como acordes. En ocasiones, estas notas son percibidas como simultaneas, aunque en realidad se trate de notas sucesivas, como ocurre en el caso de los arpegios. Generalmente también se engloba dentro de la armonía al estudio de las progresiones armónicas, esto es, las sucesiones de acordes que se presentan a lo largo de una obra musical, así como los principios estructurales que las gobiernan.

Compás: es la unidad de tiempo en la que se divide una frase u obra musical. En el pentagrama se indica mediante líneas divisorias que indican el final de un compás y el comienzo de otro. Mediante una doble barra se indica el final de la obra. También se utilizan dobles barras para separar distintas secciones o partes de la obra, en cuyo caso la segunda barra es igual de gruesa que la primera.

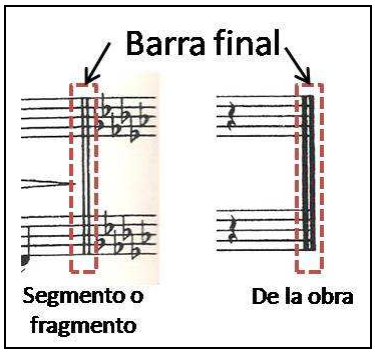


Figura 2 – Final de fragmento y final de obra

Armadura de clave: para lograr una escala distinta de la escala mayor de Do, se tiene que hacer uso de alteraciones adicionales. El conjunto de las alteraciones necesarias para construir una determinada escala mayor se denomina “armadura”, la cual se escribe inmediatamente después de la clave y su efecto es continuo hasta el fin de la obra o hasta que se produzca una modulación o cambio de tonalidad.

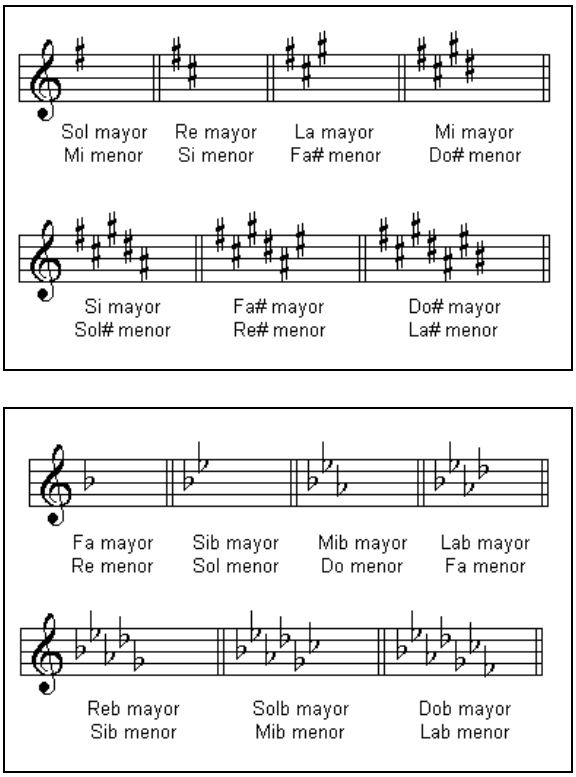


Figura 3 – Armaduras de clave en cada tono

Escala: Se llama escala musical al orden de notas de mayor a menor y por orden ascendente de la sucesión de sonidos consecutivos de un sistema (tonalidad) que se suceden regularmente en sentido ascendente o descendente, y todos ellos con relación a un sólo tono (tónica) que da nombre a toda la escala.

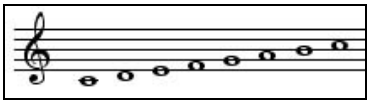


Figura 4 – Escala de C

Cadencia: Serie de acordes o fórmula de melodía que indica el fin de una sección en una obra. La cadencia perfecta correspondería al punto en la puntuación, la cadencia imperfecta al punto y coma y la cadencia rota a la coma [2].

2.1.2. Escalas

En este apartado se estudiará con más detalle qué es una escala. Para ello se utiliza como ejemplo la escala de Do mayor.

En música, cada nota de la escala se identifica con un grado. De este modo, la escala de Do mayor, que viene determinada de la siguiente manera (después de la séptima nota se repetiría nuevamente el mismo patrón):

DO RE MI FA SOL LA SI DO

y que en notación internacional[4], esta se escribiría de la siguiente manera:

C D E F G A B C

tiene unos grados correspondientes, identificados mediante números romanos como los siguientes:

I II III IV V VI VII I

Estos grados además tienen un nombre (lo que será fundamental para realizar el análisis automático):

I - Tónica
II - Supertónica
III - Mediante
IV - Subdominante
V - Dominante
VI - Superdominante
VII - Sensible

Esta escala mayor de Do (C en notación internacional) está dispuesta de manera que existe una distancia de un tono entre cada nota, excepto entre MI y FA, que están separadas por medio tono y entre SI y DO, que están también separadas por medio tono:

Tono[5] – tono – semitono – tono – tono – tono – semitono
C D E F G A B C

Fuera de la escala de DO mayor, existen 5 sonidos más que se identifican con 10 notas debido a la enarmonía (es decir, notas diferentes que se relacionan con una misma frecuencia sonora): C# y Db; D# y Eb; F# y Gb; G# y Ab; A# y Bb.

El instrumento en el que más claramente se observa esto es el piano, por lo que lo tomaremos como ejemplo[6]. En el piano, las teclas blancas pertenecen a la escala diatónica de C, mientras que las teclas negras serían sus alteraciones (las notas sostenidas y bemoles):

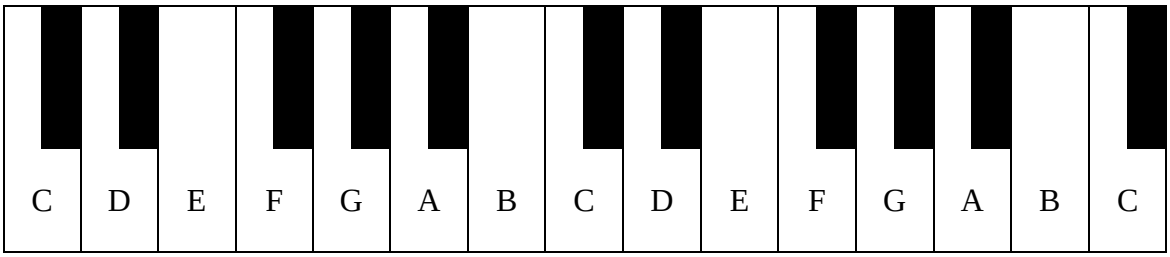


Tabla 1 – Representación de las teclas de un piano

[4] Esta notación será utilizada más adelante y será la que el programa utilice preferentemente para representar los acordes.
[5] A partir de ahora se utilizará la notación T para un tono completo y ST para un semitono.
[6] No obstante, las conclusiones que se extraen son generales para la armonía, y no sólo para el piano.

De manera que existe medio tono entre el sonido producido por cada tecla y la tecla inmediatamente más cercana (lo cual, tomando todas las notas, formaría la escala cromática).

Las teclas negras (que se identifican con las notas sostenidas o bemoles, es decir, notas con alteraciones) servirán para formar una escala igual a la de DO, es decir, con una distancia entre notas de [T – T – ST – T – T – T – ST], pero comenzando en notas diferentes de la de DO. Esto daría lugar a la escala de RE mayor, si se comienza en RE, de MI mayor, si se comienza en MI, y así sucesivamente. En una partitura musical, esto viene expresado mediante la armadura de clave.

2.1.1. Otras escalas

Como acabamos de ver, es posible formar escalas similares a la de C pero partiendo de tónicas distintas de C. Para ello es necesario utilizar alteraciones. Mostramos a continuación todas las escalas que pueden formarse de esta manera (siendo la primera nota de cada una la tónica, y la que le da nombre a la escala):

- C D E F G A B
- D E F# G A B C#
- E F# G# A B C# D#
- F G A Bb C D E
- G A B C D E F
- A B C# D E F# G#
- B C# D# E F# G# A#
- Db Eb F Gb Ab Bb C
- Eb F G Ab Bb C D
- Gb Ab Bb Cb Db Eb F
- Ab Bb C Db Eb F G
- Bb C D Eb F G A

2.1.2. Intervalos

Se conoce con el nombre de intervalo a la distancia entre dos notas, ya sean éstas interpretadas de forma consecutiva o a un mismo tiempo. Los intervalos se clasifican según la siguiente tabla:

	disminuida	menor	justa	mayor	aumentada
2ª	0	0,5		1	1,5
3ª	1	1,5		2	2,5
4ª	2		2,5		3
5ª	3		3,5		4
6ª	3,5	4		4,5	5
7ª	4,5	5		5,5	6
8ª	5,5		6		6,5

Tabla 2 – Tabla de los intervalos

Los intervalos de la tabla con valor en fondo gris existen, pero no serán considerados importantes para el análisis armónico que pretendemos realizar con la aplicación. Esto es así porque se utilizan sólo para análisis más avanzados que el que se busca con la aplicación, de modo que no aportan nada al análisis que aquí se pretende realizar (para los acordes que van a reconocerse, así como para las escalas consideradas, son innecesarios).

En la tabla, la columna de la izquierda determina los posibles tipos de intervalos que hay (según su separación en grados, por ejemplo, en la escala de C mayor, entre C y G hay cinco grados de separación, por lo que estaríamos ante un intervalo de quinta), los cuales se representan en columnas según su clase. La clase viene dada por la separación en tonos entre las notas. Así, según la tabla, de nuevo para la escala de C mayor, entre F y B había una cuarta aumentada, ya que hay una separación de tres tonos entre las dos notas.

Existen también intervalos que presentan una separación superior a 6 tonos. No obstante, 6 tonos representan una distancia de una octava, y a efectos prácticos, para el análisis armónico consideraremos que para los objetivos marcados no es relevante a qué octava pertenece una determinada nota, sino sólo el intervalo equivalente que formaría con respecto a otra nota dentro de los intervalos descritos en esta tabla. Así, una novena mayor sería equivalente a una segunda mayor, del mismo modo que una oncena justa sería equivalente a una cuarta justa.

2.1.3. Formación de acordes

Una vez hemos descrito las nociones básicas sobre escalas mayores e intervalos, es necesario describir cómo se forman los acordes. Los acordes más sencillos que existen son las tríadas, que son acordes formados por tres notas. La primera nota se conoce como la tónica, formando las otras dos un intervalo de tercera y quinta respectivamente con respecto a la tónica. Existen cuatro tipos de tríadas:

- Tríada mayor: formada por la tónica, la tercera mayor y la quinta justa.
- Tríada menor: formada por la tónica, la tercera menor y la quinta justa.

- Tríada aumentada: formada por la tónica, la tercera mayor y la quinta aumentada.
- Tríada disminuida: formada por la tónica, la tercera menor y la quinta disminuida.

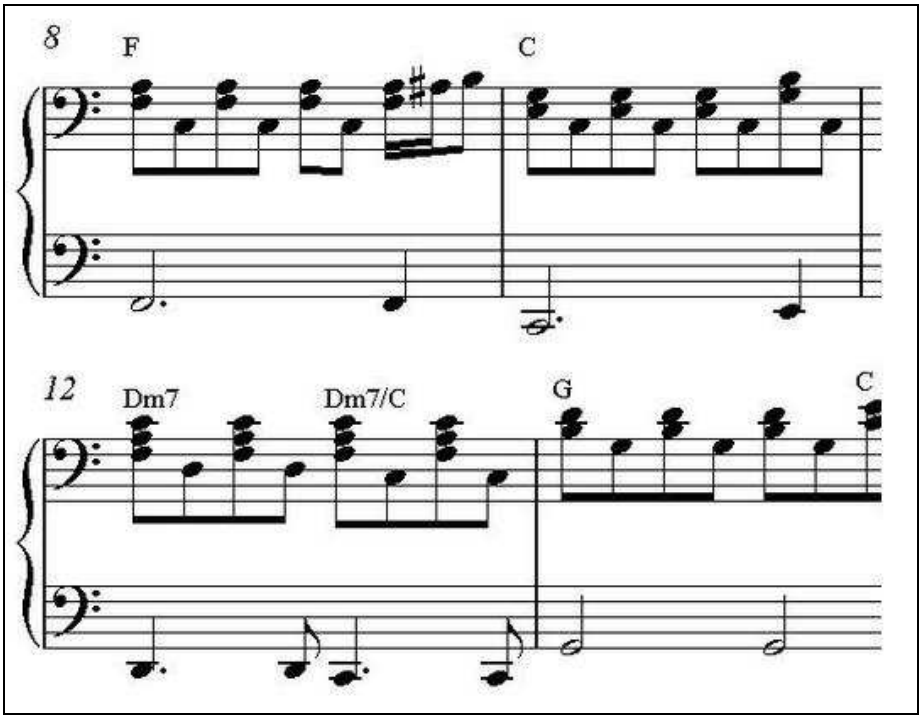


Figura 5 – Ejemplo de partitura musical con notación de acordes

De este modo, las notas C, E, G pueden ser identificadas como una tríada mayor de C, ya que E forma un intervalo de tercera mayor con respecto a C, y G forma un intervalo de quinta justa también con respecto a C. Otros ejemplos serían:

G, B, D - > Tríada mayor de G (o acorde de G).
D, F, A -> Tríada menor de D (o acorde de Dm).
F, Ab, C - > Tríada menor de F (o acorde de Fm).
B, D, F -> Tríada disminuida de B (o acorde de Bdim).

Existen varias nomenclaturas para estos acordes, pero la escogida por nuestra aplicación es la siguiente (tomamos el ejemplo con la tónica en C):

- Tríada mayor: C
- Tríada menor: Cm
- Tríada aumentada: C+
- Tríada disminuida: Cdim

Añadiendo más notas a las tríadas se pueden formar acordes más complejos. Solamente se reconocerán por la aplicación, además de las tríadas básicas, los acordes con séptima, los cuales son descritos más abajo.

Existe la posibilidad de que una tríada aparezca en una partitura con un orden distinto de tónica, tercera, quinta en lo que a altura tonal respecta. Por ejemplo, un

acorde de C mayor podría aparecer como E, G, C, ordenadas en altura tonal ascendente. Esto se corresponde con las inversiones, pero no se considerará importante en la aplicación de análisis, la cual ordenará las notas para hacerlas coincidir con su orden fundamental. Esto es así ya que un acorde y sus correspondientes inversiones son armónicamente equivalentes.

Volviendo a la escala de C mayor, si formamos todas las tríadas posibles, podemos asociar una tríada a cada grado de la escala de la siguiente forma:

I	II	III	IV	V	VI	VII
C	Dm	Em	F	G	Am	Bdim

Tabla 3 – Tríadas de cada grado en la escala diatónica de C

A continuación de las tríadas, los siguientes acordes en complejidad son los acordes de séptima, que son aquellas tríadas a las que se añade además la tensión de la séptima:

- Acorde de séptima mayor: tónica, tercera mayor, quinta justa y séptima mayor.
- Acorde mayor con séptima o acorde dominante: tónica, tercera mayor, quinta justa y séptima menor.
- Acorde menor con séptima: tónica, tercera menor, quinta justa y séptima menor.
- Acorde disminuido séptima (o semidisminuido): tónica, tercera menor, quinta disminuida y séptima menor.
- Acorde aumentado con séptima: tónica, tercera mayor, quinta justa y séptima menor.

Por su parte, las nomenclaturas escogidas para estos acordes son las siguientes (de nuevo tomando la tónica en C):

- Acorde de séptima mayor: CMaj7
- Acorde mayor con séptima o acorde dominante: C7
- Acorde menor con séptima: Cm7
- Acorde disminuido séptima (o semidisminuido): Cdim7
- Acorde aumentado con séptima: C+7

Por ejemplo, si estamos ante las siguientes notas:

D, F#, A, C

Tomando D como tónica, F# sería una tercera mayor, A una quinta justa y C una séptima menor, por lo que estaríamos ante el acorde de D7.

De acuerdo a la escala de C, podemos obtener los siguientes acordes con séptima:

- CMaj7
- Dm7
- Em7
- FMaj7

- G7
- Am7
- Bdim7 o BØ (semidisminuido)

Existen muchos acordes además de las tríadas y los acordes con séptima, pero sólo se reconocerán éstos por el motor de análisis, así como por el módulo de arreglos. Esto es así porque con las tríadas y los acordes con séptima se pueden analizar una gran variedad de obras musicales[7], de todos los estilos contemplados en 1.2, dejando el resto de acordes para futuras extensiones de la aplicación, que pretendan analizar obras de gran complejidad armónica.

2.1.4. Funciones tonales y áreas

Los siete grados de los que se compone la escala mayor, excepto la tónica (I), base de la escala, tienden en mayor o menor medida a resolver[8] sobre otro grado de dicha escala.

De esta manera, el segundo grado (II) tiende al primero (I), el cuarto (IV) al tercero (III), el sexto (VI) al quinto (V) y el séptimo (VII) al octavo, que es la tónica [2].

De este modo, los grados IV y VII son los más inestables, y los que tienen una mayor necesidad de resolver, ya que se encuentran más próximos a su resolución (un semitono de distancia).

Las funciones tonales de los siete acordes diatónicos en un tono mayor, están definidas por si éstos incluyen o no los grados IV y VII de la tonalidad. Los acordes que no incluyen el grado IV son considerados de tónica. De esta manera vemos que los acordes IMaj7, IIIm7 y VIm7 son de tónica. Los acordes que contienen el IV grado pero no el VII son considerados de subdominante. Son el IIm7 y el IVMaj7. Finalmente, los acordes que contienen los gados inestables, el IV y el VII, son considerados de dominante, y son el V7 y el VIIdim7 (también conocido como VIIm7(b5) o como BØ).

Los movimientos armónicos

Según sea su función tonal, los acordes tienden en mayor o menor grado a moverse hacia otros acordes de distinta función tonal:

- Acordes de tónica (IMaj7, IIIm7 y VIm7): Sobre todo el IMaj7, son los más estables y no tienen una tendencia a moverse hacia ningún otro acorde; su movimiento armónico es completamente libre y no viene marcado por una tendencia armónica determinada.
- Acordes de subdominante (IVMaj7 y IIm7): Son semiestables y tienden por igual a resolver sobre tónica o hacia un acorde de dominante que es más inestable.
- Acordes de dominante (V7 y VIIdim7): Finalmente los acordes de dominante tienen una marcada tendencia a resolver sobre tónica.

[7] Nótese que en todo momento, las obras musicales que se tienen en cuenta para el análisis son obras de la música occidental.

[8] Resolver en música es la tendencia a moverse hacia una nota o acorde, que tiene una determinada nota o acorde.

De acuerdo con esto, los grados que pertenecen a cada área se muestran en la tabla siguiente:

Área	Grados
Tónica	I, III, VI
Subdominante	II, IV
Dominante	V, VII

Tabla 4 – Áreas

2.1.5. Arreglos

En música, hablar de arreglos equivale a tomar una obra habitualmente compuesta por otro autor e introducirle cambios. En nuestro caso esos cambios se centrarán en la progresión armónica, es decir, la secuencia de acordes.

Para realizar arreglos sobre la armonía podremos realizar 3 acciones: añadir acordes, eliminar acordes y/o sustituir acordes. Se describe esto con más detalle a continuación:

Sustitución de acordes

En una progresión armónica determinada, es posible cambiar unos acordes por otros distintos. Esto se podrá hacer siempre y cuando la función de los acordes nuevos en la composición que se esté arreglando, sea equivalente a la de aquellos que había originalmente. Para ello, se prestará atención a las áreas que se describieron en el anterior punto. Según se describe en la Tabla 4, existen tres áreas diferentes: área de tónica, de subdominante y de dominante. Por este motivo, si el programa decide realizar una sustitución de un acorde determinado, lo cambiará por otro de la misma área al que está sustituyendo. Por ejemplo, de acuerdo con la tabla de las áreas, si estamos en tonalidad de C mayor, ante un acorde de Em tenemos 2 posibilidades, ya que se trata de un acorde del área de tónica: cambiarlo por un acorde de C o por uno de Am. En caso de que se tratara de un acorde de Dm, dado que se trata del segundo grado, estaríamos ante un acorde de subdominante, y la opción en este caso sería únicamente el acorde de F[9].

En relación a los tipos de acordes vistos, es posible realizar sustituciones atendiendo al tipo de acorde del que se trate. En nuestro caso, sólo habrá tríadas y acordes con séptima, que son iguales que las tríadas, pero con un intervalo de séptima añadido. Es posible siempre realizar una sustitución de un acorde de séptima por una tríada que contenga todas las notas del acorde de séptima menos la séptima. No obstante, en ese caso se está eliminando información que suele ser muy importante para la composición. La séptima menor añade una tensión sonora muy importante, lo que hace que los acordes que contengan este intervalo serán muy inestables, y tendrán una gran tendencia a resolver sobre otro acorde más estable (típicamente un acorde del área de tónica). En caso de sustituir un acorde mayor de séptima menor por una tríada mayor, estaremos disminuyendo la tensión en ese momento de la pieza musical, lo que no siempre es deseable. Del mismo modo, también es posible añadir la séptima a las tríadas que aparezcan a lo largo de la pieza musical. Esto puede conllevar los mismos problemas que en el caso anterior, pero en sentido inverso. No obstante, realizar

[9] Existen muchas otras opciones disponibles para realizar cambios, pero implican el uso de notas extrañas a la tonalidad, lo que se escapa a los objetivos de la aplicación de análisis armónico.

arreglos consiste fundamentalmente en eso, en cambiar de algún modo la obra original, y añadir o eliminar tensión es una de las estrategias disponibles para hacerlo, por lo que se podrá tener en cuenta a la hora de diseñar la aplicación.

Eliminación de acordes

Cuando estamos ante una progresión armónica en la que hay varios acordes seguidos que tengan la misma función, es posible eliminarlos sin que por ello se prescinda de una información necesariamente importante. Es común encontrar seguidos varios acordes que pertenezcan a la misma área, lo que se hace en muchas ocasiones para evitar monotonía en la composición. Al reducir el número de acordes, se estará simplificando la obra, lo que en ocasiones puede significar una mayor insistencia en un determinado grado, pudiendo conferirle más fuerza en la composición, pero en otras ocasiones es posible que la obra se vuelva más monótona.

Introducción de acordes nuevos

Es posible seguir varias políticas a la hora de introducir acordes nuevos en una determinada obra. En primer lugar, al igual que en los casos anteriores, es posible tener en cuenta el área al que pertenece un determinado acorde y dividir el tiempo que éste ocupa en dos o más partes para introducir otros acordes que tengan la misma función tonal. Por ejemplo, si en una composición en tono de C mayor aparece un compás que sólo tiene un acorde de C mayor, es posible dejar el C mayor en la primera mitad de dicho compás y añadir un E menor en la segunda mitad.

Por otro lado, cuando aparezca un acorde cualquiera, en muchas ocasiones podrá ser precedido de un acorde de séptima dominante con la tónica en el II grado relativo al acorde en cuestión. Por ejemplo, si estamos ante un acorde de G7, es posible en muchas ocasiones dividir el tiempo que ocupa en dos partes para introducir un D7 en la primera, ya que este acorde tiene una tendencia natural a resolver sobre el acorde de G. Esto es lo que se conoce como un dominante secundario, y no sólo puede ser utilizado sobre acordes de séptima, sino sobre cualquier acorde gracias a la tendencia a resolver de los acordes mayores de séptima menor, lo que permite introducir momentáneamente notas que están fuera de la tonalidad. No obstante hay que tener cuidado al utilizar este recurso, ya que dependiendo de la melodía sobre la que se utilice, es posible que dé lugar a una sonoridad desagradable. Por otro lado, cuando estamos ante un acorde mayor con la séptima menor, es posible dividir el tiempo que éste ocupa en dos mitades, introduciendo en la primera un acorde menor de II grado relativo. Por ejemplo, en tonalidad de C, si estamos ante un compás que sólo contiene un acorde de G7, es posible dejar ese acorde para la segunda mitad del compás e introducir un Dm7 o simplemente un Dm en la primera mitad.

2.1.6. Cadencias

Los enlaces entre los acordes de una progresión armónica pueden catalogarse en diferentes cadencias. Una cadencia es una sucesión armónica que lleva a un cierto punto de reposo. Viene a ser como la puntuación en la lengua escrita; los tipos de cadencia determinan el tipo de reposo en la frase musical, como los puntos y comas lo hacen en el lenguaje escrito.

Existen cuatro tipos de cadencias (se indican también los términos en lengua inglesa ya que es el inglés el idioma utilizado por la aplicación de análisis):

- Auténtica (Authentic)
- Plagal (Plagal)
- Rota (Deceptive)
- Semicadencia (Open)

La cadencia auténtica y la plagal se conocen como conclusivas, siendo las otras dos (semicadencia y cadencia rota) conocidas como suspensivas.

- La cadencia auténtica viene determinada por un reposo sobre el acorde de tónica al que se llega desde el acorde de dominante (V7).
- La cadencia plagal viene determinada por un reposo sobre el acorde de tónica al que se llega desde el acorde de subdominante.
- La semicadencia es un momentáneo reposo sobre un acorde que no es de tónica. La fórmula básica es sobre un acorde de dominante.
- La cadencia rota se da cuando el acorde dominante no va al del primer grado de tónica. Se trata de una cadencia auténtica truncada en el último acorde. Las más frecuentes son cuando el acorde dominante va a parar a un acorde de tónica diferente al de grado I.

2.1.7. Modos

Hasta ahora se ha explicado todo con referencia a la escala de C mayor, tal y como se describía al principio de este apartado. No obstante, existen muchas otras escalas sobre las que pueden estar escritas las obras musicales. Tampoco tenemos interés en profundizar mucho en este aspecto, por lo que sólo se explicarán algunas escalas fundamentales que contienen los 7 grados, y que pueden explicarse a partir de la que ya conocemos. Por ejemplo, si tomamos la escala de C, pero en vez de comenzar en C, se comienza con el primer grado en A, se obtiene la siguiente escala:

A B C D E F G A

Cuya separación en tonos ya no es T – T – ST – T – T – T – ST

Sino que pasa a ser: T – ST – T – T – ST – T – T

Pese a que las notas de composiciones escritas en esta escala y la de C son iguales, al convertirse A en la tónica de la escala, obtenemos una nueva escala completamente distinta de la original, y la función de cada nota varía. Esto da lugar a que el oyente perciba un estado de ánimo muy distinto para composiciones escritas en cada una de estas escalas. Mientras que la primera (escala de C mayor o escala jónica de C) transmite optimismo y alegría, la otra (escala de A menor o escala eólica de A) se asocia con un estado de ánimo triste y melancólico. Se dice que la primera es mayor porque el intervalo formado entre el primer y tercer grado es mayor, mientras que la segunda escala es menor ya que el intervalo formado por los grados I y III de la escala es menor.

Cualquier nota de la escala puede ser tomada como tónica para formar nuevas escalas, lo que dará lugar a un total de 7 escalas distintas, que se asocian con 7 modos

griegos. Cada modo tiene una sonoridad característica. La escala de C mayor se correspondería con el primer modo (el jónico), mientras que la de A menor se correspondería con el sexto modo (el eólico). Mostramos a continuación la lista de los 7 modos griegos:

I - Jónico
II - Dórico
III - Frigio
IV - Lidio
V - Mixolidio
VI - Eólico
VII – Locrio

2.2. Trabajos previos

Existen gran variedad de proyectos desarrollados hasta el momento que, de un modo u otro, relacionan informática y música. Como se dijo en la introducción, sólo resultan interesantes para este proyecto aquellos trabajos que consigan que la computadora presente algún resultado *inteligente*, es decir, que muestre resultados propios de una persona (o lo más cercanos posible) con unos conocimientos o entrenamiento específicos sobre alguna rama de la música.

Uno de los trabajos más interesantes que hemos tenido la oportunidad de encontrar trataba el problema de la identificación de la melodía principal en una partitura polifónica [11]. En él, Madsen y Widmer proponen un método que muestra una efectividad bastante sorprendente pese a lo relativamente sencillo de su planteamiento. El método consiste en analizar partituras polifónicas (extraídas de archivos MIDI), mediante ventanas solapables de duración fija, donde se analiza la entropía (entendida con relación a la teoría de la información de Shannon) de las notas atendiendo a la altura tonal, a los intervalos que éstas forman y a la duración de las mismas, partiendo del supuesto de que la melodía será percibida por el oyente humano como aquella sucesión monofónica de notas que presenta una mayor carga de información, esto es, que es menos predecible y, por tanto, en términos matemáticos que presenta una mayor entropía. Tras realizar varios experimentos con piezas de Mozart y Haydn, para las cuales contaban con la extracción manual de la melodía estimada por una musicóloga, llegaron a la conclusión de que el método presentaba unas tasas de acierto bastante altas dada la complejidad del problema y la dificultad inherente de su formalización para ser traducida al lenguaje computacional.

Por otro lado, existen varios proyectos que se han enfocado a la reproducción *expresiva* de partituras mediante la computadora. Uno de los primeros intentos de lograr expresividad en interpretaciones musicales por ordenador fue el que desarrolló Johnson en [29], donde presenta un sistema experto que determina el tempo y la articulación[10] que se ha de aplicar cuando se interpretan las fugas de Bach del "Clave bien temperado". El trabajo de Bresin, Friberg, Fryden, y Sundberg en la KTH [30],[31],[32],[33] es uno de los más conocidos sistemas de interpretación expresiva. Su actual sistema de "Director Musices" incorpora reglas para el tempo, valores dinámicos

[10] La articulación en música se refiere a la forma en que se produce la transición de un sonido a otro, o sobre la misma nota.

y transformaciones de la articulación en el estándar MIDI. Estas reglas son inferidas tanto del conocimiento teórico musical como de datos experimentales mediante sistemas de entrenamiento.

También hay que destacar las aplicaciones que generan acompañamientos inteligentes para músicos solistas y las de síntesis musical [17],[3]. En [17], se investiga cómo un músico artificial inteligente puede seguir a un intérprete humano mediante la interpretación de una pieza (es decir, siguiendo la partitura) gracias a un modelo oculto de Markov sobre la estructura de la pieza musical. El sistema interactúa con el músico y proporciona un acompañamiento musical adecuado.

Un grupo muy innovador de aplicaciones informáticas a destacar son las de composición musical, es decir, aquellas que tienen por objeto que el ordenador genere composiciones musicales. A este respecto, Hiller e Isaacson [15], barajan un método de "generación y prueba", en el que una serie de notas generadas de forma pseudo aleatoria mediante cadenas de Markov son posteriormente probadas por una heurística específica para comprobar si cumplen una serie de reglas de la armonía clásica y el contrapunto, eliminando aquellas notas que no cumplan las reglas. También hay que destacar otros trabajos que, para la composición, no utilizan una generación puramente aleatoria, sino que mediante sistemas de generación basados en heurística tratan de imitar al comportamiento humano. Por otro lado, existen además sistemas expertos para la armonización, como el CHORAL y redes neuronales que aprenden a modelar armonías musicales, como el proyecto MUSACT [34] y como HARMONET, que además introducía restricciones y que fue posteriormente ampliado al proyecto MELONET.

Relacionados con estos estudios estarían aquellos que tienen por objeto generar improvisaciones de forma automática. La improvisación musical es un proceso creativo muy complejo habitualmente denominado como "composición al vuelo". Debido a las fuertes restricciones temporales, la improvisación es, desde un punto de vista creativo, más complicada que la composición, en la que los músicos tienen tiempo para revisar y mejorar su trabajo. Uno de los primeros trabajos sobre improvisación musical mediante ordenador es el sistema "Flavors Band" de Fry [35], el cual es un lenguaje procedimental sobre LISP para especificar estilos musicales populares y jazz.

Por último, un trabajo especialmente interesante es el que encontramos en [16], donde se describe un sistema capaz de clasificar partituras de obras clásicas dentro del período temporal en el que fueron compuestas (en intervalos de un siglo entre los años 1500 y 2000).

No obstante, no hemos podido encontrar aplicaciones que, mediante la aplicación de conceptos teóricos musicales, analicen la estructura de una determinada partitura o estimen determinados parámetros relacionados con ella. Es por esto que la aplicación que aquí se propone tiene un carácter iniciático, tanto en lo que respecta a aplicaciones que muestren un análisis de la música desde un punto de vista teórico, como funcionando de motor de apoyo para aplicaciones relacionadas con las que comentábamos anteriormente, ya que la progresión armónica puede aportar información que seguramente genere patrones interesantes para el estudio de otro tipo de

características de la obra, no necesariamente ligadas de forma directa con la teoría musical[11].

2.3. Recursos tecnológicos

Uno de los primeros pasos a la hora de desarrollar el proyecto fue la elección de las tecnologías que iban a servir de base para la aplicación. En primer lugar, era evidente que se necesitaba escoger un formato en el que codificar las partituras que la aplicación tomaría como entrada de datos, así como escoger el lenguaje de programación más adecuado para el desarrollo de la misma. Además, había que determinar si eran o no necesarias otras herramientas, como pueden ser los entornos de análisis relacionados con la minería de datos o alguna otra arquitectura específica de la inteligencia artificial, tales como las redes neuronales o aproximaciones estadísticas por citar algunos ejemplos.

Dado que la teoría musical en la que se basa este proyecto establece reglas bastante claras sobre el análisis armónico, hemos decidido desarrollar un sistema basado en reglas, dejando el uso de paquetes externos para posibles extensiones de las funcionalidades básicas de la aplicación, es decir, se deja abierta la puerta para el uso de paquetes de análisis basados en cualesquiera enfoques que utilicen los resultados de esta aplicación para estudiar otros parámetros o profundizar en el análisis teórico (como se detalla en el apartado 6.2). Esto se facilita mediante el desarrollo de una API amigable que permite utilizar la funcionalidad del motor de análisis armónico y acceder con facilidad a los resultados que éste obtiene.

2.3.1. Formato de archivo

Uno de los aspectos que hay que tener en cuenta a la hora de realizar la aplicación, es el formato de archivo que ésta empleará para la entrada y salida de información sobre las partituras. Se necesita un tipo de archivo capaz de representar partituras de forma simbólica, es decir, que no presente una mera imagen o un mapa de bits de la partitura en cuestión, sino que contenga información sobre los objetos que han de representarse en la partitura (notas, silencios, claves, armaduras de clave, anotaciones…) para que ésta pueda representarse correctamente mediante una aplicación de representación gráfica de las mismas. Además es necesario buscar un formato que contenga partituras de la calidad suficiente como para poder realizar el análisis disponiendo de toda la información necesaria.

Figura 6 – Logo de Finale

En la actualidad existen gran variedad de formatos de representación electrónica de partituras. En este apartado se repasan algunos de ellos y se justifica cuál es el más adecuado para ser utilizado por la aplicación de análisis armónico.

Entre los formatos abiertos, destacamos:

- MusicXML (.xml o .mml), desarrollado por Recordare LLC [5]
- LilyPond (.ly) [8]

[11] Ver apartado 6.2

- LenMus (.lms) [9]

Entre los formatos cerrados de aplicaciones propietarias, destacamos:

- Finale (.mus) [10]
- Sibelius (.sib) [6]
- Encore (.enc) [7]

Se va a primar la elección de formatos libres, por lo que no se examinarán en detalle estos formatos propietarios, sino que simplemente se mencionan debido a su gran importancia y difusión. En caso de que fuera necesario analizar partituras disponibles únicamente en uno de estos formatos, existe la posibilidad de conversión. Por ejemplo, en lo que respecta a Sibelius, está soportada la posibilidad de exportación de partituras al formato MusicXML.

Figura 7 – Logo de Recordare

Con respecto a los formatos abiertos, todos ellos pueden representar partituras con alta calidad, siendo quizás LilyPond el que mayor calidad tiene [8]. GNU LilyPond es un programa de software libre para edición de partituras. Está escrito en C++ y construido mediante una biblioteca de Scheme (GNU Guile) que también permite la personalización y extensión por parte del usuario. Utiliza una sencilla notación de texto como entrada, y produce una salida en el formato predeterminado PDF (a través de PostScript) y también en SVG, PNG y MIDI. A pesar de que tiene el punto fuerte de que es capaz de representar partituras de una calidad muy alta, en su contra está la poca difusión de este formato, en parte debido a que el programa LilyPond carece de interfaz gráfica y, por este motivo, se hace muy inaccesible a personas que no tengan conocimientos sobre informática. Creemos que en el futuro es posible que haya una mayor cantidad de aplicaciones que hagan uso de este potente formato, pero por el momento preferimos descartarlo para la aplicación de análisis. Algo parecido ocurre con LenMus. LenMus es un proyecto de código abierto orientado a enseñar al usuario a aprender el lenguaje musical, y dispone de un formato propio de partituras, pero es difícil encontrar partituras escritas en este formato fuera de la aplicación LenMus. En cambio, existen una gran cantidad de recursos disponibles para MusicXML, ya que es el estándar abierto más aceptado actualmente por la industria, por lo que es fácil encontrar aplicaciones que lo soporten, e incluso traductores de otros formatos a MusicXML. El formato MusicXML fue desarrollado por Recordare LLC, asimilando varios conceptos clave de formatos académicos existentes (como el Musedata de Walter Hewlett, o el Humdrum de David Huron). Fue diseñado para el intercambio de partituras, particularmente entre diferentes editores de partituras.

La versión 1.0 fue publicada en enero de 2004. La versión 1.1 fue publicada en mayo de 2005 con un soporte de formato mejorado. La versión 2.0 fue publicada en junio de 2007 e incluyó un formato comprimido estandarizado. Todas estas versiones fueron definidas por una serie de definiciones de tipos de documento (DTD). Una implementación del esquema XML (XSD) de la versión 2.0 fue publicada en septiembre de 2008. Si además tenemos en cuenta que, al estar basado en XML, su lectura y escritura de forma secuencial es relativamente sencilla, y que también hay bibliotecas disponibles para varios lenguajes que facilitan esta tarea, podemos llegar a la conclusión de que, entre los formatos que estamos barajando, MusicXML es el más adecuado para

la aplicación de análisis. A continuación mostramos una figura que muestra algunas de las muchas aplicaciones disponibles que soportan este formato:

Figura 8 – Aplicaciones con soporte para MusicXML [5]

Sobre MIDI

Pese a que no es un formato específico para partituras, es interesante (debido a su tremenda difusión) mencionar la posibilidad de extraer una partitura de un archivo MIDI. Pasamos a describir esto brevemente:

MIDI son las siglas de (Musical Instrument Digital Interface) es decir, una interfaz digital para instrumentos musicales. Se trata de un protocolo estándar, definido en 1982, y diseñado para que los instrumentos musicales electrónicos pudieran comunicarse, controlarse y sincronizarse entre ellos. Este estándar define un tipo de archivo conocido por su extensión .MID o .SMF (de Standard MIDI File).

No obstante, dado que las partituras extraídas de este tipo de archivos en la mayoría de los casos suelen ser de muy baja calidad (insuficiente para el análisis), se descartará el uso de este tipo de archivos en las primeras versiones de la aplicación, dejando la puerta abierta para futuras versiones o extensiones.

2.3.2. Lenguaje de programación

Como toda aplicación software, la aplicación de análisis de partituras debe ser programada utilizando un lenguaje que permita, facilitando la tarea en la medida de lo posible, implementar las características necesarias para el desarrollo de su funcionalidad. Para ello, deberá encargarse de la entrada/salida, es decir, de tomar los datos de entrada desde un archivo con una partitura musical, procesarlos y enviarlos convenientemente al sistema o subsistemas de análisis. Estos sistemas pueden formar parte del programa o bien estar dentro de una aplicación externa específica.

Dado que el nivel de abstracción es alto, sólo se consideraron lenguajes de alto nivel. Entre ellos, destacamos Java, C, lenguaje M (de MATLAB), BASIC, Perl, Fortran, Ada y Python. Dentro de estos lenguajes, si prestamos atención a cuáles disponen de herramientas ya desarrolladas para gestionar la lectura de los archivos MusicXML, la lista se reduce drásticamente a tan sólo Java y C++, aparte de algunas aplicaciones aisladas en Perl que tienen por objeto la conversión entre formatos de partituras. De modo que para desarrollar la aplicación tenemos sólo estos dos lenguajes de programación como candidatos potenciales (C++ y Java). En principio ambas opciones serían válidas, pudiendo C++, por su parte, proporcionar una mayor velocidad y/o rendimiento en el uso del procesador, así como una gestión óptima de la memoria. No obstante, Java ha sido la opción escogida, ya que la velocidad no es un factor en absoluto crítico del sistema, y este lenguaje automatiza los procesos de gestión de memoria, permitiendo centrarnos en aspectos puramente funcionales de la aplicación. Además, el código programado en Java es mucho más fácilmente transportable a distintos sistemas siempre que tengan instalada la máquina virtual de Java (la JVM).

2.3.3. ProxyMusic

Dado que escogimos MusicXML como el formato utilizado para la entrada/salida de partituras y Java como el lenguaje de programación, es necesario resolver el manejo de la entrada/salida de datos MusicXML dentro de una aplicación Java. Para ello se ha hecho uso de la biblioteca que proporciona el proyecto ProxyMusic [20].

ProxyMusic proporciona un enlace entre objetos Java en memoria y los archivos XML basado en el formato MusicXML 2.0, y está desarrollado bajo la licencia GNU Lesser Public License. Se presenta como un archivo .jar dedicado, permitiendo la inclusión de ProxyMusic en la lista de paquetes a importar dentro de nuestra aplicación. ProxyMusic hace uso de JAXB2, el cual está incluido dentro de Java 6, [20]. Puesto que se utiliza ampliamente el concepto de *Generics*, para el desarrollo de la aplicación ha sido necesario utilizar una versión de java igual o superior a la 1.5 (en nuestro caso, se ha utilizado la versión 1.6).

Generics proporciona una forma de comunicar el tipo de objetos contenidos dentro de una colección de forma que el compilador pueda comprobar que se está utilizando el objeto *Collection* de una forma consistente. Así, por ejemplo, para definir un nuevo vector que albergue objetos de la clase String en Java se tendrá que utilizar la siguiente línea:

```
Vector <String> myStrings = new Vector<String>();
```

Para más información sobre *Generics*, Sun Microsystems pone a disposición del internauta un completo tutorial [25].

JAXB es un acrónimo derivado de Java Architecture for XML Binding, y constituye un entorno para el procesado de documentos XML, proporcionando ventajas significativas con respecto a métodos anteriores como Document Object Model (DOM). Existen dos métodos fundamentales para acceder a los archivos XML mediante JAXB: “unmarshall” y “marshall”. El primero se encarga de crear un árbol de objetos Java en

base a la información contenida en el archivo. El segundo toma dicho árbol y genera un archivo XML nuevo.

De esta manera, nuestra aplicación trata la información contenida en las partituras mediante un árbol de objetos cuya raíz es un objeto de la clase *ScorePartwise*, definida dentro del paquete *ProxyMusic*, a partir del cual puede obtenerse toda la información incluida en la partitura y, por consiguiente, toda la información necesaria para el análisis.

2.3.4. Sistema basado en reglas - DROOLS

Una vez desarrollado el núcleo de análisis armónico, estaremos en disposición de comenzar con el módulo de arreglos, el cual se apoya en los resultados ofrecidos por el motor de análisis (accediendo a ellos a través de la API) y los utiliza para alterar la progresión armónica. De esta forma, este módulo representa, aunque venga integrado en el resto de la aplicación, un ejemplo de posible aplicación para el motor de análisis armónico que, en este caso, tiene por objeto realizar arreglos[12] de manera automática.

Para el módulo de arreglos automáticos, se ha hecho uso de un sistema conocido en la jerga de la inteligencia artificial como sistema basado en reglas, el cual se clasifica dentro del grupo de sistemas expertos. En los sistemas basados en reglas hay dos tipos de elementos: los datos (hechos o evidencia) y el conocimiento (el conjunto de reglas almacenado en la base de conocimiento). El motor de inferencia usa ambos para obtener nuevas conclusiones o hechos. Por ejemplo, si la premisa de una regla es cierta, entonces la conclusión de la regla debe ser también cierta. Los datos iniciales se incrementan incorporando las nuevas conclusiones. Por ello, tanto los hechos iniciales o datos de partida como las conclusiones derivadas de ellos forman parte de los hechos o datos de que se dispone en un instante dado [26].

Para poder integrar un sistema de este tipo en la aplicación Java, se ha hecho uso de Drools, el cual permite definir las reglas a aplicar en un archivo específico, de forma que éste pueda ser modificado alterando el comportamiento del programa sin necesidad de recompilarlo, lo que puede ser de gran utilidad en casos en que estas reglas varíen con frecuencia o estén aún siendo probadas o completadas. Esto permite no sólo una modificación “al vuelo” del comportamiento del programa, sino que da lugar a una separación formal entre la lógica que se encarga de la tarea (en este caso la realización de los arreglos) y el resto del programa. En el apartado 3.4.7, se describe en detalle cómo funciona esta aplicación, pero aquí se avanzará sobre el funcionamiento y el cometido de Drools, el motor de reglas que se ha utilizado.

Drools es un sistema software de reglas de gestión de negocio (BRMS) con una inferencia de encadenamiento hacia adelante basado en un motor de reglas, más correctamente conocido como sistema de reglas de producción, mediante una implementación mejorada del algoritmo Rete [19].

Drools soporta el estándar JSR-94 para su motor de reglas de negocio y el entorno de empresa para la construcción, mantenimiento y ejecución de las políticas de negocio en una organización, una aplicación o un servicio.

[12] Ver apartado 2.1.5 para más información sobre los arreglos en música.

Historia

El proyecto Drools fue iniciado por Bob McWhirter en 2001, y registrado en SourceForge. La versión 1.0 de Drools nunca fue publicada ya que las limitaciones del enfoque de fuerza bruta de búsqueda lineal fueron pronto superadas, por lo que se puso en marcha Drools 2.0, que se inspira en el algoritmo Rete, siendo trasladado el proyecto a Codehaus. Durante el ciclo de desarrollo 2.0 en el Codehaus Nobi Y, se convirtió en el proyecto principal, llevándolo a una versión final 2.0. En este momento, el proyecto se había convertido en el principal motor de reglas abierto de Java, respaldado por una fuerte comunidad y con una demanda real de servicios comerciales. En octubre de 2005 Drools se federó en JBoss como parte de su oferta de JEMS, cambiado su nombre a JBoss Rules. En 2006 JBoss fue adquirido por Red Hat. Con el apoyo financiero de las JBoss, la reescritura de JBoss Rules fue posible con una implementación Rete completa y con herramientas GUI mejoradas. A mediados de 2007, el nombre Drools fue reclamado ya que después de dos años, se seguía llamándolo así predominantemente, y tener que referirse a ella como "Drools A.K.A. JBoss Rules" o "Drools (JBoss Rules)" daba lugar a confusión.

La versión 5.0 de Drools fue liberada el 19 de mayo de 2009. Los objetivos principales de esta versión son para hacer frente a CEP o Complex Event Processing (en un módulo llamado Fusion) y a las capacidades de flujo de trabajo (en un módulo llamado Flow)

Como ya se ha dicho, Drools es una implementación de motor de reglas basada en el algoritmo *Rete* de Charles Forgy adaptado para el lenguaje Java. Adaptar Rete a una interfaz orientada a objetos permite una expresión más natural de las reglas de negocio con respecto a los objetos de negocio. Está escrito en Java, pero capaz de funcionar con Java y. NET.

Actualmente, las reglas pueden ser escritas en Java, MVEL, Python, y Groovy. Drools también dispone de programación declarativa, y es lo suficientemente flexible para que coincida la semántica del dominio del problema con el dominio de lenguajes específicos (DSL) a través de XML mediante un esquema definido para el dominio del problema. Los DSLs consisten en elementos y atributos XML que representan el dominio del problema.

Integración con Java

Para introducir reglas interpretables por el motor de inferencias de Drools en una aplicación Java, es necesario seguir una serie de pasos.

En primer lugar hay que escribir una clase que sirva de interfaz con el Motor de Reglas (Rule Engine). Esta clase hará uso de las bibliotecas .jar de Drools, por lo que lo primero que habrá que hacer será ir a la dirección de descarga de Drools [27], y descargar el archivo *drools-5.0-bin.zip* del enlace de descarga en la línea donde indica *Drools 5.0 Binaries*. A partir de este archivo, Drools se proporciona al usuario como bibliotecas jar (listas para usar simplemente agregándolas al classpath de la aplicación).

Ejemplo

Un ejemplo sencillo de la clase que funciona como interfaz con el Motor de Reglas sería el siguiente:

-RuleRunner.java-

```
package com.javarepository.rules;
import java.io.InputStreamReader;
import org.drools.RuleBase;
import org.drools.RuleBaseFactory;
import org.drools.WorkingMemory;
import org.drools.compiler.PackageBuilder;
import org.drools.rule.Package;

public class RuleRunner
{
    public RuleRunner(){}

    public void runRules(String[] rules, Object... facts)
    throws Exception
    {
        RuleBase ruleBase = RuleBaseFactory.newRuleBase();
        PackageBuilder builder = new PackageBuilder();

        for(String ruleFile : rules)
        {
            System.out.println("Loading file: "+ruleFile);
            builder.addPackageFromDrl(
                    new InputStreamReader(
                        this.getClass().getResourceAsStream(ruleFile)));
        }

        Package pkg = builder.getPackage();
        ruleBase.addPackage(pkg);

        WorkingMemory workingMemory
            = ruleBase.newStatefulSession();

        for(Object fact : facts)
        {
            System.out.println("Inserting fact: "+fact);
            workingMemory.insert(fact);
        }

        workingMemory.fireAllRules();
    }

    public static void rule1()
    throws Exception
    {
        Number n1=3, n2=1, n3=4, n4=1, n5=5;
        new RuleRunner().runRules(
                new String[]{"/simple/rule01.drl"},
                n1,n2,n3,n4,n5);
    }
    public static void main(String[] args)
    throws Exception
    {
        rule1();
    }
}
```

Esta clase cargaría un archivo de reglas llamado *rule01.drl*, el cual se encuentra dentro del directorio *simple*.

En cuanto al archivo de reglas, un archivo correspondiente al ejemplo de arriba sería el siguiente:

rule01.drl

```
package simple
rule "Rule 01 - Number Printer"
    when
        Number( $intValue : intValue )
    then
        System.out.println("Number found with value: "+$intValue);
```

```
end
```

Con este sencillo ejemplo de aplicación para Drools, al ejecutar obtendríamos la siguiente salida:

Loading file: /simple/rule02.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 5
Number found with value: 4
Number found with value: 1
Number found with value: 3

Con esto vemos dos características clave del funcionamiento de este tipo de motor de inferencias: en primer lugar, comprobamos que se basa en dos acciones fundamentales: la primera consiste en introducir los hechos (en este caso los hechos son números, pero podemos introducir como hecho cualquier objeto Java), y la segunda acción consiste en la aplicación de las reglas, las cuales están codificadas dentro de archivos de extensión .drl. La otra característica que observamos de la ejecución de este ejemplo es que no pueden introducirse hechos repetidos, ya que al intentarlo, éstos sencillamente no quedarán registrados después de la primera repetición.

Para aprovechar el potencial de este paquete, es necesario hacer uso de la introducción y eliminación de hechos dentro de las reglas, de forma que tras introducir los hechos iniciales dentro de la clase Java que hemos visto arriba, estos hechos disparen unas determinadas reglas, las cuales a su vez introduzcan o eliminen nuevos hechos dando lugar a que nuevas reglas sean disparadas, hasta que se cumplan los objetivos que persiga el desarrollador. Para ello es necesario tener en cuenta las posibilidades que los archivos .drl brindan.

A continuación se hará una breve explicación[13] sobre qué puede hacerse con un archivo .drl:

Como ya ha podido observarse, la sintaxis básica para construir reglas dentro de un archivo .drl es la siguiente:

rule "nombre de la regla"
    when
        <condiciones>
    then
        <acciones>
end

Dentro de un archivo .drl pueden introducirse tantas reglas como se deseen.

[13] Para una información más detallada, ver [21].

En lo que respecta a las <*condiciones*> para que una regla se aplique, éstas se refieren a los hechos que hayan sido introducidos. Los hechos son objetos Java que poseen ciertos atributos (a elección del desarrollador). Para comprobar si un determinado objeto se ha añadido a *facts* (hechos) se utiliza la siguiente sintaxis:

$referenciaalobjeto : ClaseJava ($referenciaalatributo : atributo)
<comprobación booleana sobre el atributo en cuestión>

Sobre la comprobación booleana, se puede hacer uso de la función *eval()* de Drools, que hace una evaluación de un booleano Java similar al que pueda utilizarse por ejemplo dentro de un if(). Siguiendo el ejemplo anterior, podría implementarse la siguiente línea en caso de que *atributo* fuera un entero:

eval($referenciaalatributo > 0)

En cuanto a las <*acciones*> que pueden realizarse, es posible introducir código Java, pero se recomienda que sean lo más simples posible. Un ejemplo sería mostrar una determinada salida por la consola (*System.out.println("Texto a mostrar")*). Otro ejemplo sería eliminar un hecho concreto por medio de una línea como la siguiente:

retract($referenciaalobjeto)

O añadir un nuevo hecho, utilizando una llamada a insert(), como por ejemplo:

insert(new ClaseJava(atributoX, atributoY));

Para ver ejemplos más complejos y más posibilidades de las reglas de Drools, se recomienda visitar la guía de la web de jboss [21].

3. Descripción del sistema

La aplicación ha sido enteramente desarrollada en Java[14]. Por ello, para dar una idea global sobre cómo se ha desarrollado el programa, una vez repasada una descripción general de la arquitectura del sistema (apartado 3.1), se detallará el formato con los datos de entrada (apartado 3.2) y se describirá su API en líneas generales (apartado 3.3) para, posteriormente, pasar a describir en detalle su funcionamiento interno. Por último, como referencia para futuros desarrollos, se describirá brevemente cómo fue desarrollado dentro de un proyecto de Eclipse y cómo es puesto a disposición del usuario mediante un .jar ejecutable.

3.1. Arquitectura del sistema

Antes de hacer una descripción detallada de los procesos que son invocados a lo largo del análisis armónico, vamos a proceder a dar visión sobre cómo ha sido diseñada la arquitectura de la aplicación, para después detallar cómo esta arquitectura ha sido traducida a código Java.

Podemos describir la arquitectura de la aplicación desarrollada mediante el siguiente diagrama de bloques:

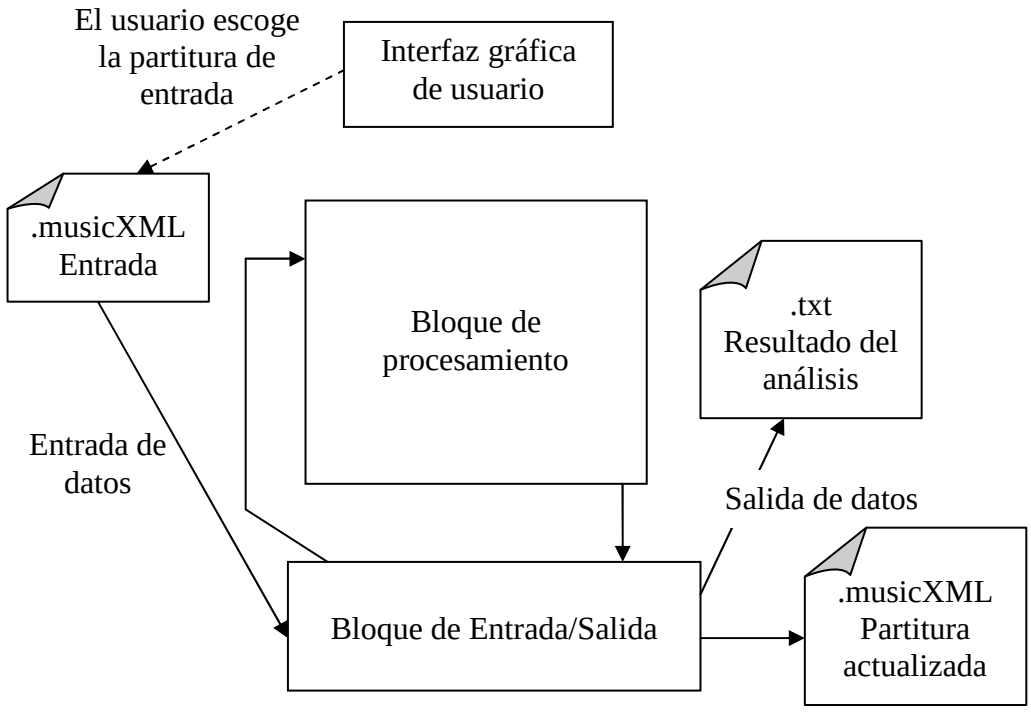


Figura 9 – Arquitectura de diseño

Las partes fundamentales de esta arquitectura son:

[14] Donde se incluye también al módulo de arreglos, puesto que Drools también está desarrollado en Java.

- Por un lado, lo que respecta a la información propiamente dicha (los archivos, es decir, las figuras con la esquina doblada), tanto lo que se refiere a la entrada de datos (en formato MusicXML), como a la salida (también en formato MusicXML y además, un archivo de texto con la información del análisis realizado).

- En un punto intermedio, se encontraría el bloque de entrada/salida, que es aquél que durante el proceso de entrada de datos traduce esta información (codificada en un formato XML estándar) en estructuras adecuadas para su manejo por parte del motor de análisis. Del mismo modo, realiza la operación inversa cuando se ha terminado el análisis, para obtener los archivos de salida con los resultados.

- Por último, está el bloque referente al procesado de esta información. Esta parte es aquella que maneja los datos de entrada para extraer la información relevante, valiéndose después de un conocimiento teórico (que ya fuera explicado dentro del apartado 2.1), para así realizar el análisis armónico.

Hay que destacar que la aplicación se ha desarrollado teniendo en mente que el motor de análisis pueda ser utilizado por una aplicación externa (para lo que se facilita la API que permite el uso del mismo), por lo que la interfaz gráfica mostrada en el diagrama se incluye únicamente como medio de demostración del funcionamiento del motor de análisis, de modo que es un bloque completamente accesorio y perfectamente podría sustituirse por una aplicación que hace una llamada al motor de análisis indicando el archivo de partitura que se ha de analizar.

Una vez vistos los bloques fundamentales, pasamos a describir con un grado más de detalle el bloque de entrada/salida:

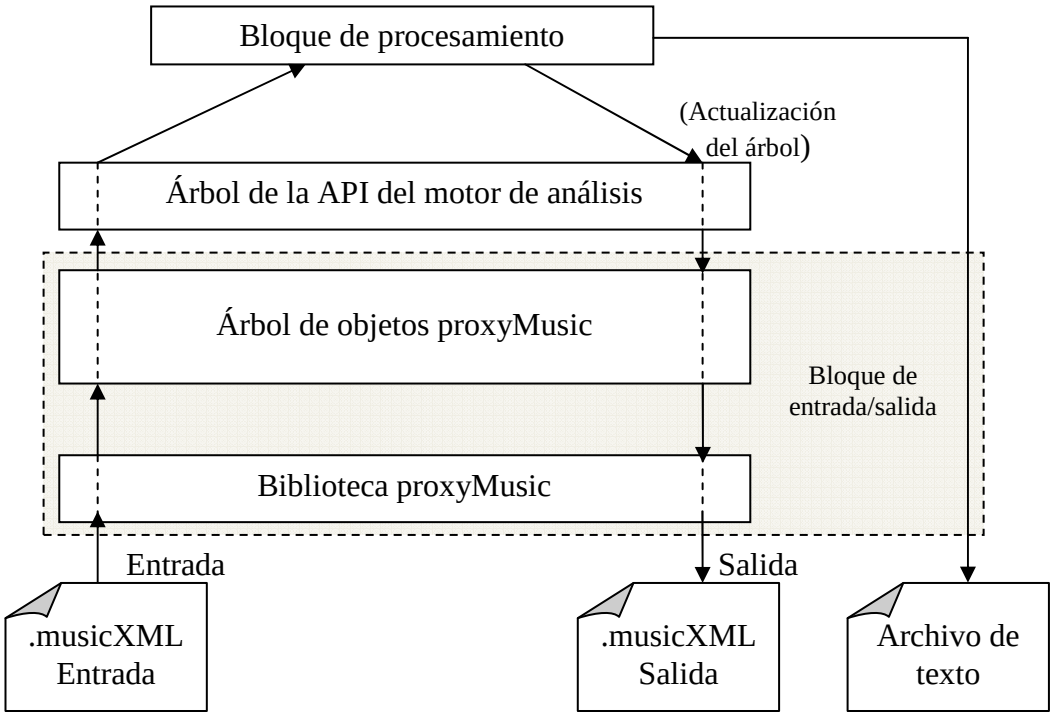


Figura 10 – Arquitectura de la entrada/salida

Para que el motor de análisis armónico pueda manejar la información contenida en los archivos de partituras, es necesario en primer lugar realizar una conversión de la información XML contenida en estos archivos en un árbol de objetos Java[15]. Para ello se ha hecho uso del paquete *proxyMusic* que ya fue mencionado en el apartado 2.3.3. Una vez se dispone de toda la información de la partitura en el árbol de objetos que proporciona proxyMusic, este árbol de objetos proxyMusic es traducido a un nuevo árbol definido en la API del motor de análisis[16], el cual es mucho más sencillo y sólo contiene la información relevante para el análisis armónico, estando ésta además perfectamente ordenada y preparada para el análisis. Cuando ha terminado el procesado de la información, se actualiza primero el árbol de objetos de la API del motor de análisis para, posteriormente, actualizar el árbol de proxyMusic y, de ahí se vuelve a hacer una llamada a proxyMusic para generar un nuevo archivo con la partitura actualizada. Por otro lado, el módulo de procesado genera un informe que el bloque de entrada/salida (esta vez sin tener que hacer uso de proxyMusic) convierte en un archivo de texto.

En lo que respecta al motor de análisis y arreglos, éste puede ser descrito con más detalle de la siguiente forma:

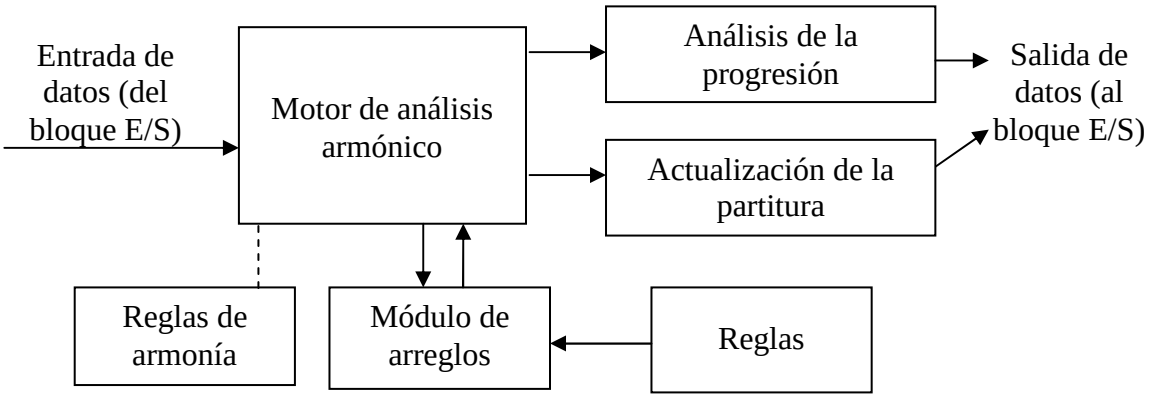


Figura 11 – Arquitectura del bloque de procesamiento

En la figura se ve cómo al motor de análisis llega la información preparada por el bloque de entrada/salida. Ésta es analizada por el motor de análisis armónico siguiendo las reglas de armonía. La información sobre armonía que se obtiene en el análisis es introducida en el árbol de la API del motor de análisis. Después, se pasa al módulo que analiza la progresión armónica y elabora un informe de texto con el análisis armónico realizado, así como al módulo que actualiza el árbol de objetos proxyMusic para poder generar la partitura de salida con información sobre la progresión armónica de la obra.

En caso de que se hayan seleccionado arreglos, después de esto se introducen como hechos en el módulo de arreglos toda la información relevante para la armonía extraída de la partitura, la cual está convenientemente organizada dentro de la API del

[15] Este proceso de conversión se describe detalladamente en el apartado 3.2
[16] Ver apartado 3.3 para más información sobre cómo está estructurado el árbol de objetos de la API del motor de análisis.

motor de análisis armónico. Con esta información, el módulo de arreglos modifica la progresión armónica y devuelve los resultados al motor de análisis para que éste llame de nuevo al módulo que analiza la progresión, así como al módulo de actualización de la partitura para que se generen dos nuevos archivos. Así, el módulo de arreglos tiene un doble objetivo en este proyecto: por un lado, ejemplifica cómo la API del motor de análisis puede ser utilizado por una aplicación externa y, por otro lado, cumple uno de los objetivos principales del proyecto como es el de realizar los arreglos[17] automáticos.

En cuanto a los procesos que llevan a cabo la funcionalidad descrita en estos diagramas de arquitectura, se realiza una descripción detallada en el apartado *3.4 Descripción de los procesos*.

Por último, la arquitectura del módulo de arreglos se describe mediante el siguiente diagrama:

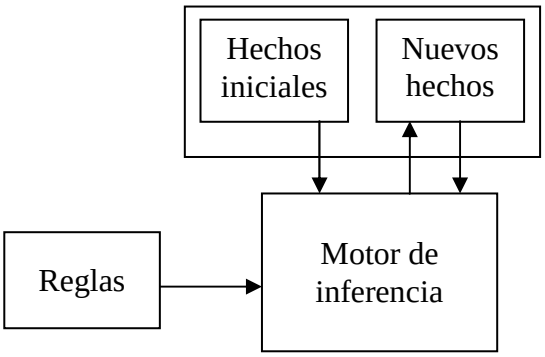


Figura 12 – Arquitectura del módulo de arreglos

En esencia, el módulo de arreglos recibe una serie de hechos iniciales (la progresión armónica), sobre los cuales, mediante el motor de inferencia, se aplica una serie de reglas (las reglas descritas en el apartado de teoría musical), de forma que se generan nuevos hechos (nuevos acordes) a los que se siguen aplicando las reglas. Una vez converge el algoritmo de aplicación de reglas, los hechos iniciales (que pueden haber sido modificados en el proceso), junto con los nuevos hechos, conforman el resultado del proceso.

[17] Ver apartado 2.1 sobre la teoría musical.

3.2. Datos de entrada

En este apartado se describe en detalle cómo son los datos que la aplicación de análisis toma como entrada y cómo el bloque de entrada/salida convierte esos datos en información útil y manejable por la aplicación.

Los archivos MusicXML siguen un formato basado en XML (eXtensible Markup Language), el cual es un metalenguaje extensible de etiquetas desarrollado por el W3C (World Wide Web Consortium), que permite definir la gramática de lenguajes específicos. Como todos los formatos basados en XML, MusicXML puede ser fácilmente manipulado y compilado por herramientas automáticas. Un ejemplo sencillo de un archivo en este formato sería el siguiente:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 2.0
Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="2.0">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
```

```
  </part>
</score-partwise>
```

Figura 13 – Ejemplo de código de una partitura básica en MusicXML

Este archivo representa una partitura en clave de sol con un compás de 4/4 y una redonda en el primer y único compás en la nota DO, lo que gráficamente sería:

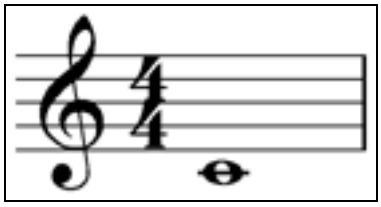


Figura 14 – Ejemplo gráfico de una partitura básica en MusicXML

Para convertir toda esta información en objetos Java que puedan ser utilizados por la aplicación se ha hecho uso de la biblioteca *proxyMusic*. Concretamente, mediante una llamada al método *unmarshal(InputStream inputStream)* de la clase *proxyMusic.Marshalling* se obtiene un árbol de objetos Java que contiene toda la información del archivo con la partitura en formato MusicXML que se pase como argumento. De esta forma, si se volviera a escribir una nueva partitura partiendo de dicho árbol (mediante una llamada al método *marshal(ScorePartwise, OutputStream)* de la misma clase), la nueva partitura sería exactamente igual a la original.

El árbol de objetos Java que proporciona la llamada a *unmarshal* de la clase *proxyMusic.Marshalling* puede representarse mediante el siguiente esquema:

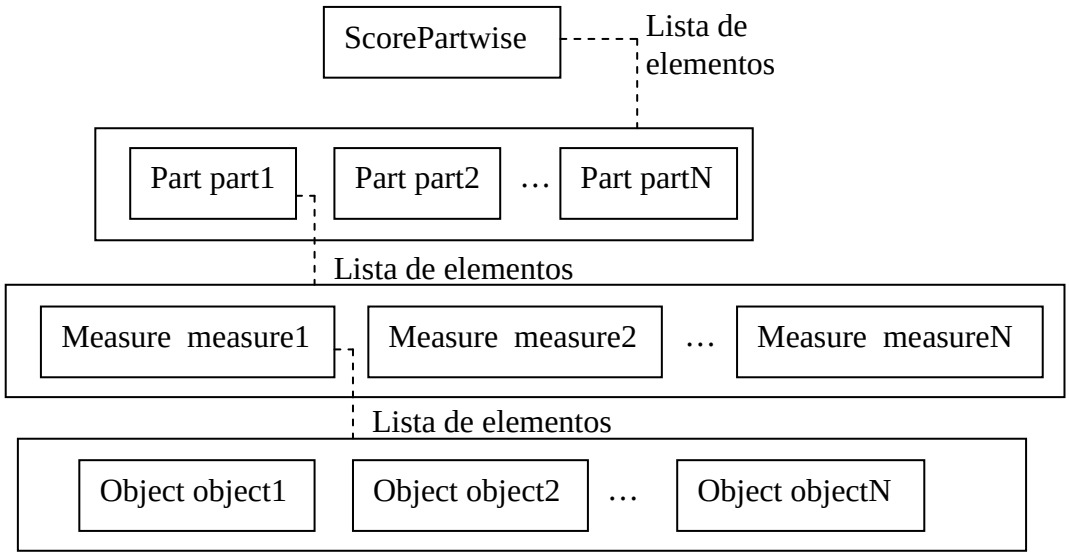


Figura 15 – Árbol de objetos Java extraído de una partitura MusicXML

Como podemos ver, la raíz es un objeto de la clase *proxyMusic.ScorePartwise*. Este objeto representa a la partitura en su conjunto. A su vez este objeto contiene una lista con elementos de la clase *proxyMusic.Part*. Cada elemento de esta lista representa a las distintas partes de la partitura, esto es, partes que están escritas para ser

interpretadas al mismo tiempo por distintos instrumentos o distintas voces. Cada parte a su vez contiene una lista con objetos de la clase *proxyMusic.Measure*. Éstos se corresponden con los compases. Como se vio en el apartado 2.1, un compás es la unidad de tiempo en la que se divide una frase u obra musical. En el pentagrama se indica mediante líneas divisorias verticales que indican el final de un compás y el comienzo de otro. Mediante una doble barra se indica el final de la obra. También se utilizan dobles barras para separar distintas secciones o partes de la obra, en cuyo caso la segunda barra es igual de gruesa que la primera. Dentro de cada compás también hay una lista, pero los elementos contenidos en ella pueden ser de diversas clases. Para el análisis hay dos tipos de objetos relevantes: los objetos de la clase *proxyMusic.Attribute* y los de la clase *proxyMusic.Note*, que representan atributos de la partitura (clave, armadura de clave, tipo de compás…) y notas respectivamente.

A continuación, se muestra una partitura, indicando los objetos que contienen cada uno de los elementos de interés:

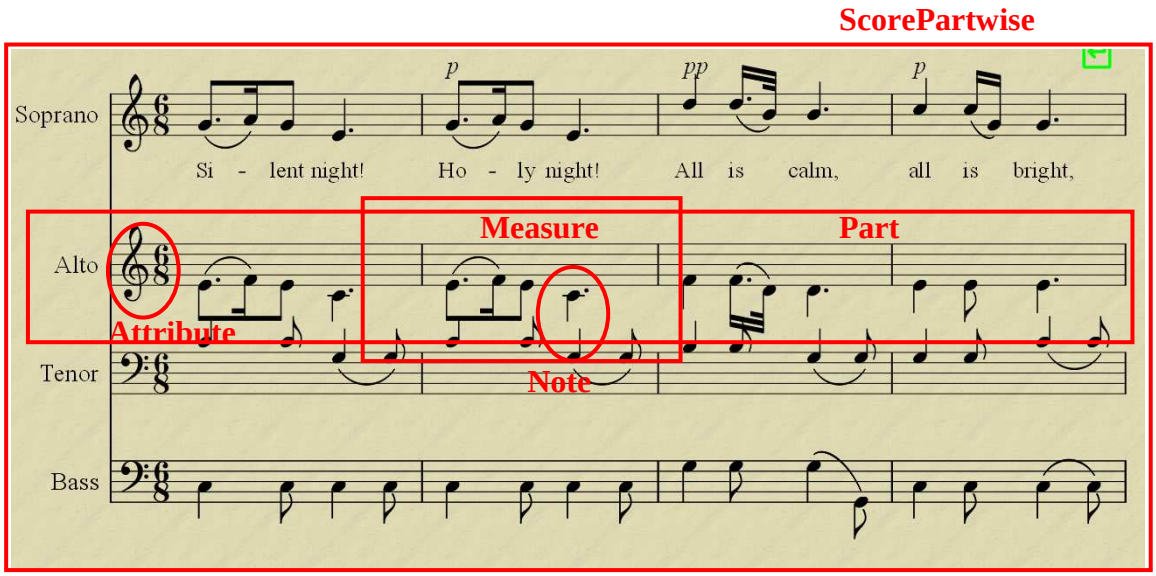


Figura 16 – Correspondencia entre algunos elementos de la partitura y los campos del formato MusicXML

Se pretende hacer hincapié en que no todos los objetos que contiene la partitura son de interés para el análisis. En este árbol sólo se han representado aquellos objetos que resultan de interés para realizar el análisis. Otros objetos serían la letra de las canciones, los indicadores de instrumentos, el título de la pieza, anotaciones diversas, etc.

Una vez vista la información que proporciona proxyMusic, podemos volver al ejemplo sencillo de archivo MusicXML que se mostró al inicio de este apartado con el objetivo de indicar la correspondencia entre los campos que el archivo contiene y los objetos que se acaban de describir:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 2.0 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="2.0">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

Aquí comienza la partitura

Aquí comienza la primera parte de la partitura

Aquí comienza el compás número 1

Del campo fifths se obtiene la armadura de clave

Este valor indica el tipo de compás usado por la partitura

En este campo se define una nota

Figura 17 – Archivo MusicXML donde se especifican algunos de los campos de interés y su presencia en el árbol de objetos Java

Por último, es importante destacar que mediante el campo *Doctype* se hace referencia al tipo de documento en concreto que se está representando (en este caso la definición concreta de MusicXML) y a la URL donde puede encontrarse en internet el archivo con su definición de tipo de documento[18] o DTD (Document Type Definition), la cual será descargada de Internet por *proxyMusic* para poder reconocer correctamente

[18] Para más información sobre este tipo de archivo, visitar la dirección: http://www.w3schools.com/DTD/default.asp

cada campo. Es por esto que se precisa de una conexión a Internet para que la aplicación de análisis armónico funcione. Si el archivo no cumple con la estructura y sintaxis allí definidas, se generará un error y el programa no comenzará análisis alguno, mostrando un mensaje de error, y volviendo al punto inicial tras la ejecución sin haber generado ningún archivo de salida.

3.3. API de la aplicación

En este apartado se explicarán aspectos relacionados con la API del motor de análisis, tanto en lo que se refiere a su diseño para el acceso por parte de aplicaciones externas, como en lo que respecta a la información que sus objetos albergan tras la realización del análisis.

Con el fin de facilitar la tarea de integración de la información obtenida por el motor de análisis armónico con aplicaciones futuras, se ha hecho uso intensivo de la documentación del código para así poder elaborar de forma automática mediante la herramienta javadoc un directorio con toda la documentación en HTML. Esta documentación, al igual que el resto de los componentes de la aplicación, ha sido enteramente desarrollada en inglés con el objetivo de hacerla accesible a la mayor cantidad posible de desarrolladores en todo el mundo.

En primer lugar, hay que destacar que la aplicación pertenece a un paquete llamado *scoreAnalysis*. Mostramos a continuación la información referente a las clases de este paquete:

Package scoreAnalysis

Class Summary	
ActualScale	ActualScale represents a scale and implements methods with some basic functions.
ChordLib	Auxiliary class used to detect intervals and chords within a given scale.
GUI	Graphical User Interface for the ScoreAnalysis application.
NotesInMeas	It symbolizes the ordered notes of a measure.
ProcMeasure	ProcMeasure is a representation of a measure that has processing information from the ScoreAnalysis application.
RuleRunner	Auxiliary class used to work with the drools inferences engine.
ScoreAnalysis	ScoreAnalysis is the class that contains the main functionality of the application.
SimpleChord	It symbolizes the basic information of a chord.

Tabla 5 – Clases de la API de la aplicación

Para permitir que el paquete pueda funcionar y ser probado sin necesidad de recurrir a una aplicación externa, se ha incluido una clase *GUI* que contiene una interfaz gráfica de usuario la cual permite probar las funciones básicas. No obstante, esta clase es completamente accesoria, y tiene como único fin la presentación gráfica, por lo que es totalmente prescindible si lo que se busca es utilizar el motor de análisis dentro de una aplicación externa.

La clase central de la aplicación, la cual implementa el motor de análisis desarrollado se llama *ScoreAnalysis*. De modo que si se desea analizar una partitura, es necesario crear un objeto de esta clase. A continuación se describe esta clase con más detalle, para lo que en primer lugar se muestra un extracto de la documentación de la misma:

scoreAnalysis

Class ScoreAnalysis

```
java.lang.Object
  └ scoreAnalysis.ScoreAnalysis
```

```
public class ScoreAnalysis
extends java.lang.Object
```

ScoreAnalysis is the class that contains the main functionality of the application. This class contains the analysis engine and its results can be obtained in a vector of ProcMeasure objects by means of the creation of a ScoreAnalysis object with a code like this:

```
Vector <ProcMeasure> results; // The vector with the results of
the analysis
File file = new File("inputfile.xml");
ScoreAnalysis sca = new ScoreAnalysis();
sca.unmarshalFile(file);
results = sca.startAnalysis(0); // No arrangements selected. The
results are now available
sca.marshalFile("outputfile.xml"); // The score with information
about harmony is saved.
```

Since:

1.0

Version:

1.0

Author:

F. Javier R. Donado

Como podemos observar, se muestra en la descripción de la clase un ejemplo con código sobre la forma de acceder a la información generada por el motor de análisis mediante la creación de un objeto de la clase *ScoreAnalysis*. Esta información se presenta dentro de un vector de objetos de la clase *ProcMeasure*, que es la que representa un compás ya procesado, con lo que el vector de resultados contiene el conjunto de la información obtenida por el motor de análisis en todos y cada uno de los compases de la aplicación. A continuación se muestra un esquema sobre la información que proporciona el motor de análisis con una llamada al método *startAnalysys(int arrangements)*:

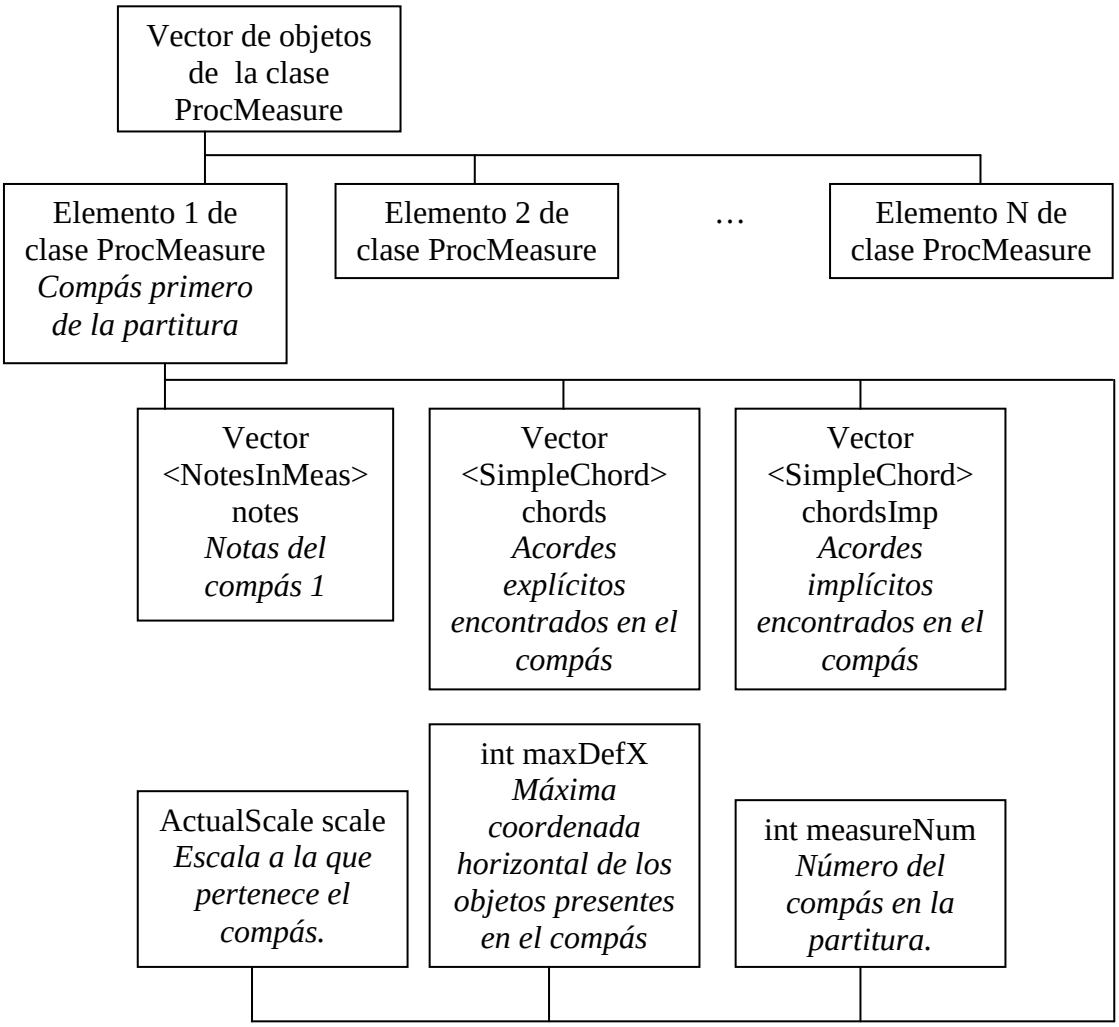


Figura 18 – Información proporcionada por el motor de análisis

Vemos que cada objeto del vector contiene información sobre las notas que hay en el compás, los acordes que aparecen de forma explícita, implícita[19] y aquellos que han sido seleccionados para la partitura de salida, un entero que contiene el número que representa el compás entre todos los compases de la partitura, un indicador sobre la máxima coordenada horizontal de los objetos que aparecen en el compás (notas y acordes), y un objeto que representa la escala utilizada para la composición de la obra que contiene el compás. Se describe a continuación con mayor detalle la información que contiene cada uno de estos objetos:

La clase *NotesInMeas* representa a cada nota o grupo de notas que aparecen en una misma posición horizontal (es decir, que deben sonar simultáneamente) en el compás. Contiene dos atributos:

- *defX*: Un entero con la coordenada en el eje horizontal de la nota o notas dentro del compás.

[19] Se denominan acordes explícitos a aquellos que aparecen como notas simultáneas en la partitura. Los acordes implícitos, por su parte, aparecen en la partitura como notas sucesivas. Esto se estudiará más detalladamente en el apartado siguiente.

- *notes*: Un vector de objetos de la clase *proxyMusic.Note*, con la nota o notas que han de interpretarse simultáneamente.

La clase SimpleChord representa a un acorde. Contiene tres atributos:

- *defX*: Un entero con la posición horizontal en el compás del acorde.
- *tonic*: Una cadena con la tónica del acorde, es decir, la nota que le da nombre.
- *type*: Una cadena que representa el tipo de acorde (mayor, menor, aumentado…)

Por último, la clase *ActualScale* representa una escala con siete grados, ya que son éstas las escalas que puede reconocer la aplicación de análisis (hay muchas más escalas en teoría musical que tienen diferente número de grados, pero no han sido incluidas en la aplicación por simplicidad). Contiene ocho cadenas: siete para las notas respectivas a cada grado y una para el modo. Aunque tampoco se tiene en cuenta el modo[20] en la aplicación de análisis, se ha considerado interesante tomar la información referente al modo que pueda haber en la partitura, ya que podría ser de utilidad en futuras versiones de la aplicación o para programas externos que hagan uso de la aplicación de análisis. Esta clase contiene una serie de métodos auxiliares para obtener información de la escala, los cuales son convenientemente descritos en la documentación de la aplicación.

Con respecto a la forma en la que se guarda la información sobre las notas, hay que destacar que es posible que haya varias partes diferenciadas en la partitura, cada una destinada a una voz distinta o a diferentes instrumentos, como ocurre, por ejemplo, en la siguiente partitura:

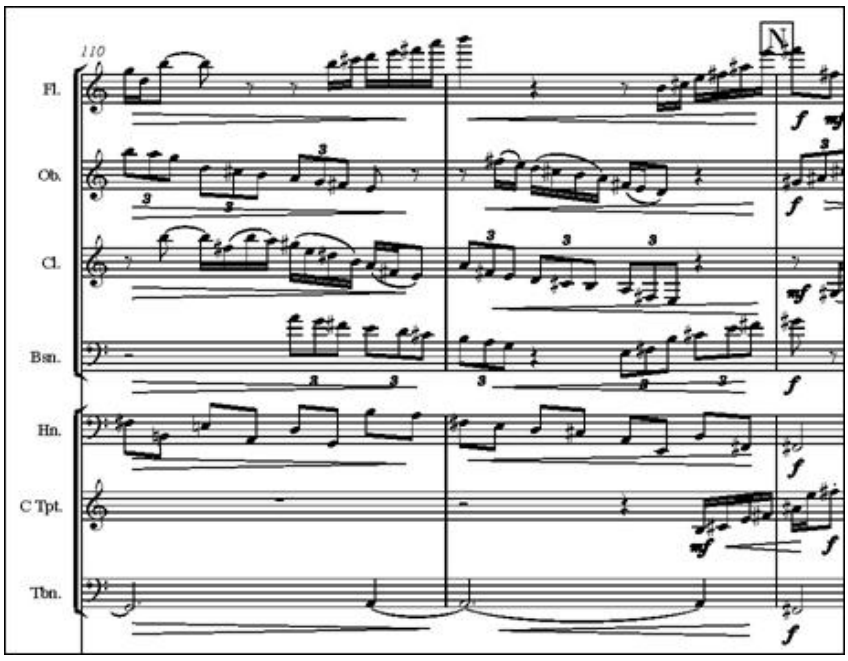


Figura 19 – Partitura con varias partes

Como podemos ver, esta partitura contiene varias voces simultáneas, correspondientes a distintos instrumentos. En lo referente a la información sobre las notas contenida en *ProcMeasure*, es importante destacar que, con el objetivo de

[20] Ver apartado 2.1.7 para más información sobre lo que es un modo.

encontrar fácilmente la información relativa a la armonía de la composición, estas partes son fusionadas en una sola para el análisis, y son tratadas como si se tratara en realidad de una sola línea para un instrumento polifónico como ocurre, por ejemplo, en las partituras para piano, por lo que al acceder a la información sobre las notas contenida en *ProcMeasure*, esto deberá ser tenido en cuenta. No obstante, la aplicación conserva la estructura original de la partitura de entrada a la hora de escribir la partitura de salida con la información sobre armonía, ya que el árbol de objetos que se obtiene de *ProxyMusic* y que se utiliza para escribir la partitura de salida se mantiene con su estructura original y solamente se añade información referente a la armonía (este cambio sólo afecta a la información albergada en objetos de la API de la aplicación de análisis).

Por otro lado, en lo que respecta al funcionamiento interno del motor de análisis armónico, contenido en la clase *ScoreAnalysis*, podemos destacar que las tareas relacionadas con éste se llevan a cabo en pasos que se corresponden con un orden de abstracción incremental con respecto a la información de entrada, trabajando en primer lugar directamente con los datos ofrecidos por el paquete proxyMusic para, posteriormente, filtrarlos y reordenarlos. Después se extrae la información relevante y, por último, se realiza un análisis de dicha información. En caso de que se hayan seleccionado arreglos automáticos, con la información de la última fase del análisis, se llama al módulo de arreglos, se aplican las reglas para dichos arreglos y se analiza la nueva información.

3.4. Descripción de los procesos

En este apartado se describen detalladamente los procesos llevados a cabo por la aplicación de análisis armónico, desde el comienzo de la ejecución hasta que se obtienen los resultados finales.

3.4.1. Inicio de la aplicación

Al iniciar la aplicación, se abre la interfaz gráfica de usuario (GUI), la cual es una sencilla ventana como la de la siguiente imagen:

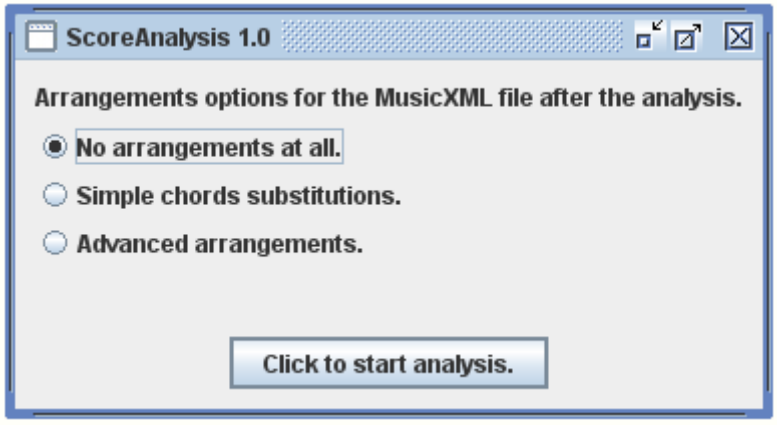


Figura 20 – GUI de la aplicación

Como podemos ver, esta ventana permite escoger entre tres clases de arreglos posibles. La primera (la que aparece seleccionada en la figura) consiste en prescindir de ellos; la segunda, por su parte, consiste en realizar sustituciones simples de acordes; en cuanto a la tercera, además de sustituir, también puede añadir y eliminar acordes, como se verá más adelante (en el apartado 3.4.7). Al hacer clic en el botón para comenzar el análisis, se abre un diálogo que permite buscar el archivo que el usuario desea analizar:

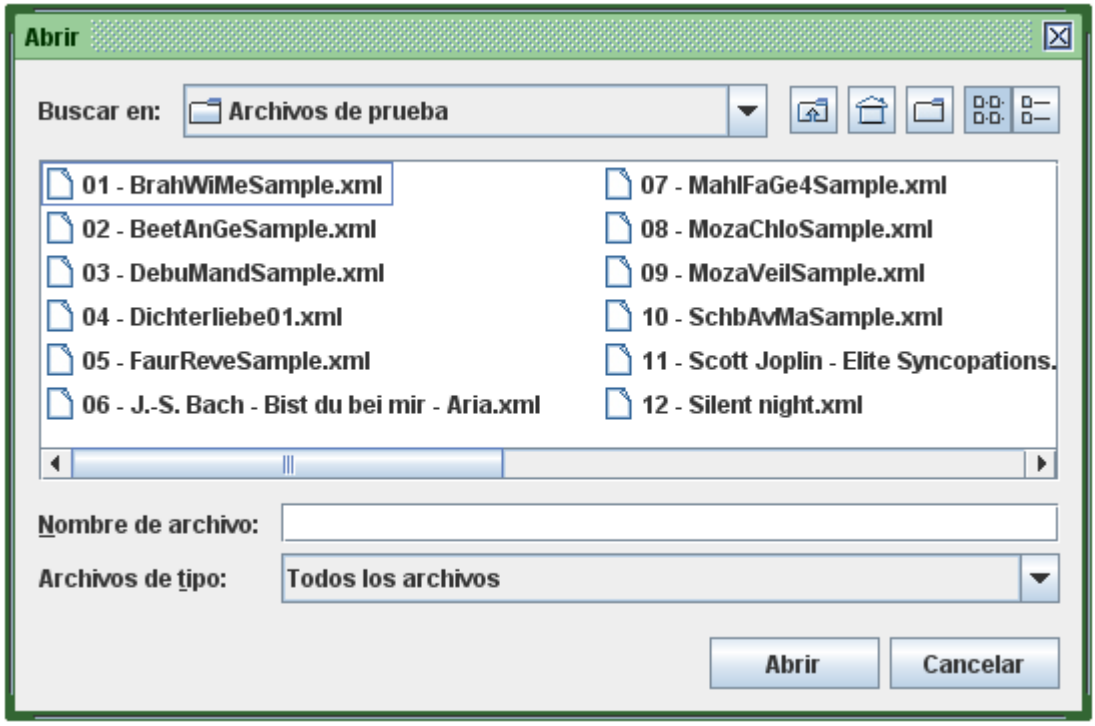


Figura 21 – Ventana de búsqueda de archivos

Una vez el usuario ha especificado a través de la interfaz gráfica cuál es el archivo con la partitura que desea analizar, ésta llama al método *unmarshalFile* de la clase *ScoreAnalysis* para generar el árbol de objetos Java de acuerdo a lo descrito en el apartado 3.2 y, acto seguido, llama al método *startAnalysis*, también de la clase *ScoreAnalysis* que ya se mencionó en el apartado 3.3, pasándole como argumento la elección del tipo de arreglos que haya solicitado el usuario. El método *startAnalysis* da comienzo al análisis armónico de la aplicación. Antes de detallar el funcionamiento interno de todos los procesos implicados en el análisis armónico, se muestra un diagrama de los mismos:

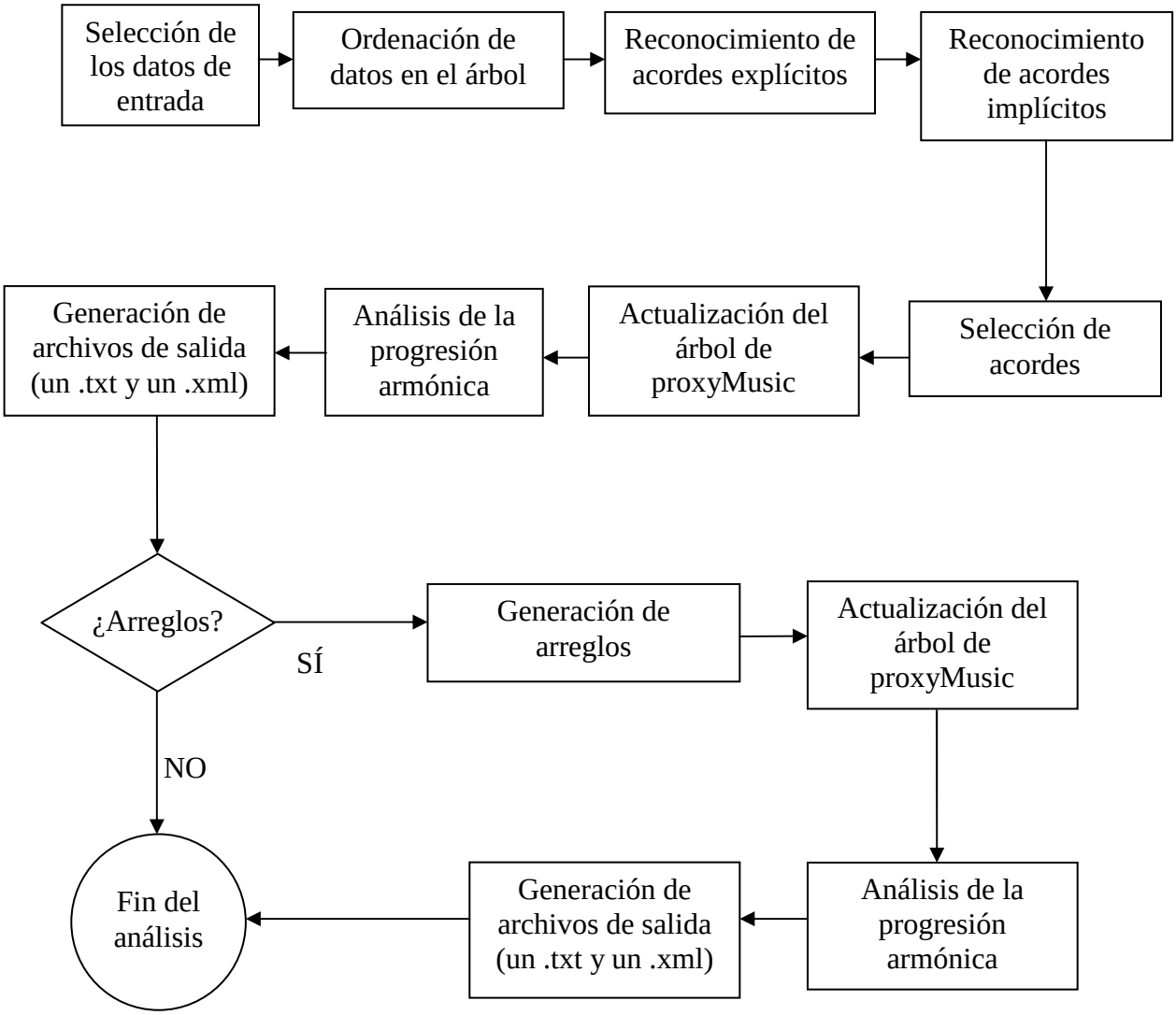


Figura 22 – Procesos del motor de análisis

A lo largo de los siguientes apartados se describirán detalladamente estos procesos.

3.4.2. Selección y ordenación de los datos

Dado que MusicXML es un formato enfocado fundamentalmente a la presentación, en ocasiones los datos de interés para el análisis armónico aparecen de forma desordenada. Es por esto que la primera operación necesaria consiste en recorrer la información de la partitura e ir ordenándola antes de proceder al análisis armónico propiamente.

Para la fase de ordenación de los datos se ha decidido mantener intacto el árbol de objetos procedente de proxyMusic (ya que modificarlo podría dar lugar también a modificaciones en el archivo de salida). Esto se ha hecho así porque lo que queremos es dejar intacto el archivo de entrada y posteriormente añadir información nueva, manteniendo igual la que ya estaba). De este modo, se almacena la información relevante en las clases descritas en el apartado 3.3 sobre la API de la aplicación.

En primer lugar, dado que lo que interesa es encontrar acordes (es decir, grupos de tres o más notas sonando simultáneamente), se recorren todos los objetos contenidos en cada compás, y se almacenan juntas aquellas notas que han de sonar al mismo tiempo (ya que en ocasiones aparecen separadas dentro del árbol). Para ello, como ya vimos en el apartado 3.3, se han creado varias clases específicas, como *scoreAnalysis.NotesInMeas*, que guarda un vector con notas simultáneas indicando además la posición en que éstas aparecen y *scoreAnalysis.ProcMeasures*, que guarda información de cada compás, entre la que podemos destacar al vector de objetos de la clase *scoreAnalysis.NotesInMeas*. Estos vectores se rellenan de forma ordenada, lo que luego facilitará el análisis. Recordamos también que cuando hay que analizar una partitura que tiene varias partes, los compases simultáneos de dichas partes se analizan como si fuera uno solo para, de esta forma, poder encontrar y analizar más tarde qué notas suenan de forma simultánea y encontrar así más fácilmente la sucesión armónica subyacente.

Por ejemplo, en la partitura siguiente, el vector *notes* de objetos de la clase *NotesInMeas* presente en cada objeto de la clase *ProcMeasure* se rellenaría de la siguiente forma:

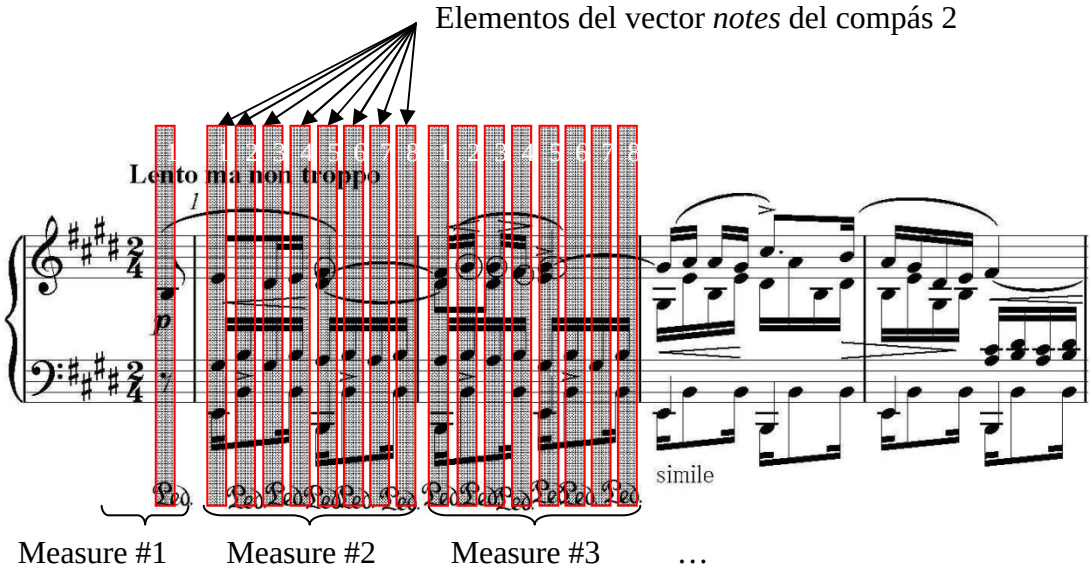


Figura 23 – Ordenación de las notas

Cada compás (measure) está representado por un objeto de la clase *ProcMeasure*, que a su vez contiene un vector (*notes*) de objetos de la clase *NotesInMeas* que contienen las notas que quedan dentro del recuadro respectivo. En el ejemplo, vemos que el primer compás sólo tiene un elemento en el vector notes, con una única nota. En cuanto al segundo compás, el vector notes tiene ocho elementos. De éstos, el primer elemento tiene tres notas, el segundo dos, el tercero dos, etcétera.

Esto se hace así porque en muchos de los archivos analizados, las notas aparecían en un orden que no se correspondía con el de ejecución como, por ejemplo (manteniendo la misma partitura):

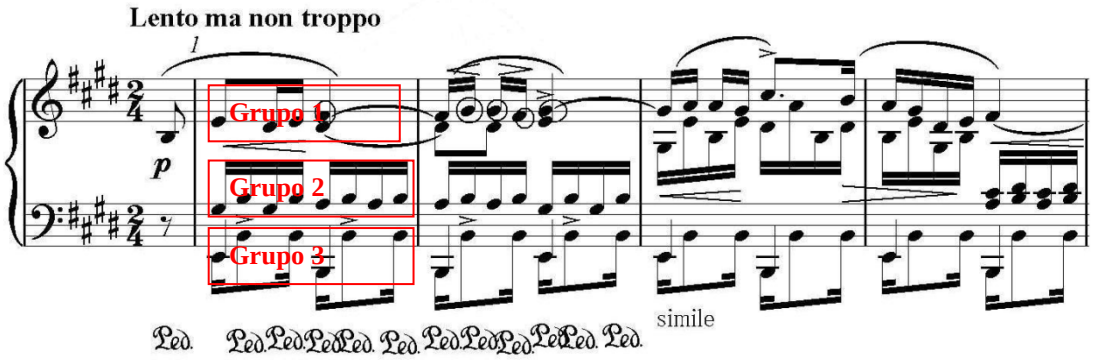


Figura 24 – Ejemplo de ordenación inadecuada de las notas para el análisis

Donde las notas aparecían en el archivo MusicXML mostrando primero aquellas del grupo 1, luego las del grupo 2 y, por último las del grupo 3 (aunque la representación en pantalla fuera adecuada, el orden con el que estos datos son escritos dentro del archivo es bastante caótico en la mayoría de los casos). Buscar notas simultáneas en el tiempo con esa clase de ordenación se hacía bastante pesado para cada nota, por lo que se decidió realizar una reordenación de los elementos de cada compás al inicio del análisis.

3.4.3. Reconocimiento de acordes explícitos

Cuando se ha terminado de ordenar los objetos de un compás, se pasa a la identificación de acordes que en él aparecen. Una vez explicados los conceptos básicos de armonía musical en los que se apoya nuestra aplicación para reconocer acordes[21], pasamos a detallar cómo estas reglas han sido implementadas en el sistema.

En primer lugar, es fundamental que el sistema de reconocimiento de acordes funcione correctamente dentro de cualquier tonalidad, aunque se presenten alteraciones a la escala de C, por lo que antes de comenzar a reconocer acordes en la partitura, el sistema toma los datos de la armadura de clave (concretamente del árbol obtenido del método *unmarshal(InputStream inputStream)* de la clase *proxyMusic.Marshalling*, a través del objeto de la clase *Fifths*, visto en la Figura 17) para averiguar qué escala se utiliza para la composición que presenta la partitura. Dicho sea de otra manera, si se le pasan como entrada a la aplicación dos partituras que contienen la misma obra pero en distintas tonalidades (lo cual se encuentra habitualmente, por ejemplo, para poder interpretarla en distintos instrumentos o por vocalistas con diferente registro), aunque todas las notas varían, el análisis debería ser fundamentalmente el mismo[22]. El análisis, posteriormente, se realizará con respecto a estos datos genéricos de todas las escalas (grados en vez de notas concretas) y no respecto a datos concretos de las notas utilizadas en una determinada escala, lo que permitirá al programa utilizar el mismo motor de análisis sea cual sea la tonalidad en la que esté escrita una determinada pieza.

[21] Ver apartado 2.1

[22] No se asegura que sea exactamente igual ya que, como veremos más adelante, hay un cierto componente aleatorio en cada análisis, puesto que, en ocasiones, hay que escoger entre varias alternativas igualmente válidas.

Por ejemplo, las siguientes notas consecutivas, que pertenecen a la escala de C:

C, G, A, B, B, D

Si en vez de estar en tono de C se necesitaran interpretar en un registro más agudo, por ejemplo, en tono de E, esta sucesión de notas, que seguiría sonando de idéntica forma (sólo que más aguda), debería tener un análisis también idéntico en su relación a los grados de la escala, pese a que ahora pasaría a estar compuesta por las siguientes notas :

E, B, C#, D#, D#, F#

Es decir, que el análisis es el mismo a pesar de que todas las notas varían, con la salvedad de que la aplicación especifica que la primera partitura está en tono de C y la segunda en E.

En este proceso, interesa encontrar acordes explícitos, es decir, grupos de notas simultáneas de la partitura que forman entre si alguno de los acordes que describimos en la teoría, como por ejemplo ocurre en la siguiente partitura:

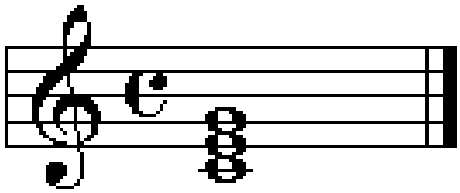


Figura 25 – Acorde de C en la partitura

Aprovechando que disponemos de la escala, a la hora de reconocer los acordes, en vez de tomar las notas directamente, se utiliza una conversión a números para identificar qué representa cada nota de la partitura. Esta conversión a números funciona de la siguiente manera: se utilizan números enteros decimales de tres dígitos, como por ejemplo:

453

El primer dígito representa el grado de la nota en la escala. Por ejemplo, si estamos en la escala de D mayor, la nota G sería el cuarto grado. El segundo número se identifica con la alteración de la nota con respecto a la escala. Si no hay ninguna alteración, este número es un 5. Si con respecto a la escala hay un semitono por encima, sería un 6, por ejemplo, un G# en la escala de D se representaría por un número cuyos dos primeros dígitos serían: 46. Por último, el tercer número representa la octava. En el ejemplo, estaríamos ante una nota perteneciente a la tercera octava.

Esto se ha hecho de esta manera para poder realizar una traducción sencilla entre cualquier escala en que pueda estar compuesta una obra y la escala de C mayor, encontrando así los intervalos por comparación dentro de una escala conocida (un tono entre notas consecutivas menos entre E y F y entre B y C que hay medio tono)

De modo que la aplicación, para detectar acordes explícitos, recorre el vector ordenado de notas simultáneas del compás, las pasa a números, y envía una agrupación

(o array) de números a un método específico (*chordCkeck(int [])*, de la clase *scoreAnalysis.ChordLib*) que finalmente identifica los posibles acordes que pueden interpretarse mediante las notas simultáneas del array.

Este método funciona de la siguiente manera: toma la primera nota como tónica y comprueba que se produzcan los intervalos necesarios entre ella (a través del método *intervalCheck(int [], String, int)* también de la clase *scoreAnalysis.ChordLib*) y alguna de las demás notas para formar alguno de los acordes reconocibles. Si es así, añade esa posibilidad a una cadena y sigue buscando con la siguiente nota como tónica. Cuando ya se han probado todas las notas del array como posibles tónicas se termina el proceso. El hecho de haber convertido las notas del acorde a números representativos de dichas notas en la escala, permite que este método sea el mismo para cualesquiera escalas y notas de entrada, ya que el método reconoce el tipo de acorde y del grado de la tónica, por lo que puede recuperarse qué nota es a partir de la escala (incluso aunque se trate de notas que no pertenezcan a la escala, ya que esta representación también permite procesarlas, indicando qué alteración con respecto a la escala incluyen).

Mostramos a continuación varios ejemplos para describir esta conversión más claramente:

Caso 1

- Notas F, A, C, todas en la tercera octava.
- Escala de C

El array de números resultante para estas notas sería:

F – 453
A – 653
C – 153

Y el acorde reconocido sería: F (fa mayor, que representa la tríada obtenida en la escala de C tomando como tónica al cuarto grado de la escala).

Caso 2

- Notas F, A, C, todas en la tercera octava.
- Escala de F

El array de números resultante para estas notas sería:

F – 153
A – 353
C – 553

Y el acorde reconocido sería: F (fa mayor, que representa la tríada obtenida en la escala de F tomando como tónica al primer grado de la escala).

Caso 3

- Notas F, A, C, todas en la tercera octava.
- Escala de Bb

El array de números resultante para estas notas sería:

F – 553
A – 753
C – 253

Y el acorde reconocido sería: F (fa mayor, que representa la tríada obtenida en la escala de Bb tomando como tónica al quinto grado de la escala).

Caso 4

- Notas F, A, C, todas en la tercera octava.
- Escala de Eb

El array de números resultante para estas notas sería:

F – 253
A – 463
C – 653

Y el acorde reconocido sería: F (fa mayor, un acorde que no pertenece a la escala de Eb, ya que la tercera del acorde con tónica en el segundo grado, según esa escala debería ser Ab en vez de A, por lo que en este caso, aunque el acorde es detectado, las reglas consideradas para analizar la progresión armónica no lo reconocerán, y será marcado como desconocido en la fase de análisis de la progresión).

Además de esto, también se ha implementado un método que compruebe que la elección de un acorde tiene sentido (método *makesSense(int [], int)* de la clase *scoreAnalysis.ChordLib*). Este método toma como entrada la tónica del acorde en cuestión y el array con los números identificativos de las notas para las que tendrá validez ese acorde. En caso de que se produzca entre la tónica del acorde y las notas sobre las que el acorde debe sonar, una cuarta aumentada (tritono) o una novena menor (medio tono), se considerará que el acorde tiene un efecto desagradable en esa parte de la pieza y se descartará como opción.

3.4.4. Reconocimiento de acordes implícitos

El siguiente paso tras la detección de los acordes explícitos de la partitura es encontrar acordes que, aunque no aparecen como notas simultáneas, pueden reconocerse a partir de una secuencia de notas que aparecen de forma consecutiva. Para ello, el método de reconocimiento de acordes es el mismo que en el caso explícito, pero varía la forma en que se agrupan los datos. En el caso de los acordes implícitos se envía el primer grupo de notas simultáneas al motor de reconocimiento. En caso de que se reconozcan uno o más acordes, éstos se apuntan dentro del apartado de acordes implícitos del vector de objetos de la clase *scoreAnalysis.ProcMeasures*. En caso de que no se detecte ningún acorde, el siguiente grupo de notas se añade al anterior y se busca un acorde entre las notas de ambos grupos, hasta que se detecte algún acorde. Cuando un acorde es detectado se comienza el proceso con el siguiente grupo de notas simultáneas hasta que se encuentra un nuevo acorde o hasta que se llega al final del compás. Así podrán encontrarse un gran número de acordes que, aunque no se

correspondan con notas que suenen de forma simultánea, sí pueden reconocerse agrupando varias notas no simultáneas del compás.

De este modo, el tamaño de ventana en lo que respecta a la cantidad de notas que pueden formar parte de un grupo sobre el que se van a detectar acordes implícitos es inicialmente ilimitado. Sólo se pasará a la siguiente ventana en el caso en que se haya detectado un acorde. Si se llega al final del compás y con las notas de la última ventana no se ha detectado ningún acorde, ningún acorde es añadido y se pasa al siguiente compás.

Se ejemplifica este proceso a continuación mediante los dos primeros compases de la siguiente partitura:

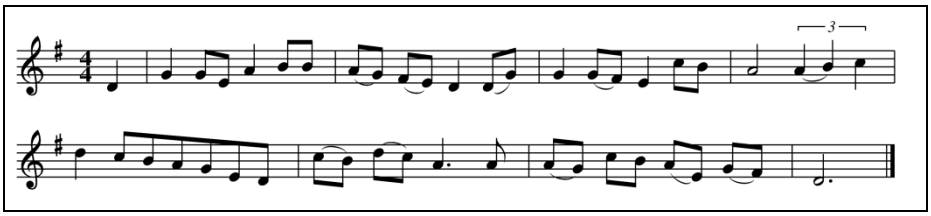


Figura 26 – Ejemplo de secuencia de notas formando acordes implícitos

El **primer compás** sólo tiene una nota, por lo que no se reconocerá ningún acorde en él.

Describimos el proceso seguido para el análisis de acordes implícitos del **segundo compás**:

- Lectura de nota: G ----> Una sola nota no forma un acorde
- Se guarda y se sigue buscando.
- Nuevo grupo de notas: G, G ----> Seguimos sin tener un acorde
- Se guarda y se sigue buscando.
- Notas: G, G, E ----> Aún no se reconoce ningún acorde
- Se guarda y se sigue buscando.
- Notas: G, G, E, A ----> No existen acordes según las reglas introducidas en el programa que identifiquen ese patrón de notas (no forman una tríada mayor ni un acorde con séptima[23])
- Se guarda y se sigue buscando.
- Notas: G, G, E, A, B ----> Se reconoce el acorde de E mayor (E, G, B)[24]
- Se comienza el proceso desde ese punto.
- Notas: B ----> No se reconoce ningún acorde
- Fin del análisis para el segundo compás

El proceso continuaría de forma equivalente para los siguientes compases de la partitura.

[23] Podría interpretarse que se forma A7, pero dado que no aparece la tercera de este acorde, el motor de análisis no lo reconocerá.
[24] En caso de reconocerse más acordes, se guardan todos los posibles acordes para una posterior selección.

3.4.5. Selección final de acordes y actualización del árbol

Una vez se ha finalizado la búsqueda de acordes implícitos y explícitos, es el momento de escoger cuáles de entre los posibles acordes se escogerán finalmente como los acordes del compás. Para la elección final de acordes se han tenido en cuenta los siguientes criterios:

- Si en un determinado instante hay acordes implícitos y explícitos, los implícitos se descartan en favor de los explícitos.
- Cuando se está escogiendo un acorde, se priman aquellos que no aparecen en la secuencia de acordes del siguiente evento temporal, ya que éstos parecen contener información distinta y posiblemente relevante para el análisis.
- Los acordes de séptima priman ante las tríadas.
- Cuando estos criterios se han aplicado y quedan dos o más acordes candidatos para un determinado instante, se escoge el resultado al azar entre ellos.

Además de esto, se tiene en cuenta que cuando el acorde de un determinado momento es igual al del momento anterior del mismo compás, éste no debe registrarse, ya que se presupone que el acorde seguirá siendo el mismo mientras no se especifique lo contrario.

Cuando se ha terminado el proceso de reconocimiento de acordes implícitos y explícitos, y se han seleccionado los candidatos para cada instante temporal del compás, el árbol con los datos originales de la partitura se actualiza con la información de la progresión armónica. Para esto, es necesario añadir un nuevo objeto de la clase *proxyMusic.Harmony* dentro del objeto de la clase *proxyMusic.Measure* que queremos actualizar, por lo que es necesario realizar una traducción entre la nomenclatura para los acordes empleada por nuestra aplicación y la empleada por *ProxyMusic*[25]. Una vez hecho esto, el acorde se añade en la posición de la lista correspondiente al instante temporal al que se refiere el acorde.

Cabe destacar que, pese a ser MusicXML un formato enfocado a la presentación, en el momento de generar archivos con la información obtenida acerca de la progresión armónica, hemos encontrado bastante divergencia en la forma que tienen de interpretarlo los distintos editores. Así, en algunas ocasiones (como ocurre con el programa MuseScore), pese a que hay un campo que indica la posición horizontal de los elementos y pese a que ese campo es utilizado por nuestra aplicación, el programa de presentación lo ignora, y los elementos se presentan en pantalla ordenados de acuerdo a la posición que ocupan dentro de la lista del árbol. Con el objetivo de maximizar la compatibilidad entre distintos editores, en nuestra aplicación se ha tenido en cuenta tanto el campo de la localización horizontal (DefX) como la colocación adecuada de los elementos dentro de la lista. No obstante, en ocasiones ha sido imposible conseguir una presentación perfecta en todos los editores probados y algunos objetos se mostraban solapados (ver apartado 4). A nuestro entender, este es un problema de representación de partituras puesto que se debe a que el programa de presentación ignora información

[25] No se empleó para la aplicación la misma nomenclatura que en ProxyMusic ya que consideramos esta última un tanto engorrosa especialmente al tener que añadir varias posibilidades para un mismo instante temporal.

relevante disponible en el archivo, y que está en muchos casos asociado al propio formato MusicXML ya que no sólo ocurre con las partituras tratadas por nuestra aplicación, sino que frecuentemente, partituras descargadas de internet o convertidas a MusicXML presentaban objetos solapados dependiendo del editor utilizado para representarlas.

3.4.6. Análisis de la progresión armónica

Una vez realizado el análisis compás por compás para extraer los movimientos armónicos en los que se apoya la composición, es momento de analizar esta información como un todo, y reconocer las funciones tonales de cada acorde, así como las cadencias.

Tras haber obtenido los acordes de cada compás en la fase anterior, nuestra aplicación dispone de la información suficiente para analizar la secuencia armónica subyacente en la partitura mediante las reglas vistas en el punto 2.1. Para hacer esto se han añadido algunos métodos en la clase *ActualScale* que facilitan el trabajo con los grados de la escala siguiendo el esquema numérico que se describió en el apartado 3.4.3, y se han introducido además algunos acordes que, aunque son externos a la escala, se utilizan habitualmente como sustitución de acordes de la escala. No obstante, existen muchos caminos para salirse de la escala fundamental cuyos fundamentos teóricos no han sido codificados en esta aplicación (ya que se buscaba realizar un análisis basado en las reglas básicas de armonía, y no en conceptos avanzados), los cuales se dejan para trabajos futuros. De esta forma, cuando la aplicación llega a un acorde cuya función tonal no es conocida, esto se representa simplemente mediante un símbolo de interrogación (*?*). Todos los resultados de este análisis son guardados en un archivo de texto.

El funcionamiento de esta última fase de análisis es el siguiente: se escribe al comienzo de cada nueva tonalidad qué nota se asocia con cada grado de la escala. Esto se hará tantas veces como se cambie de escala a lo largo de la partitura. Seguidamente se escriben los movimientos armónicos de acorde en acorde, registrando qué acorde es cada uno, a qué grado representa dentro de la escala y qué función diatónica cumple. En caso de que el movimiento de un acorde al siguiente esté relacionado con una cadencia determinada, se indica qué cadencia es. Se reconocen tres tipos de cadencia en la aplicación: cadencia auténtica, plagal, rota y semicadencia, según se describió en el apartado teórico del estado del arte (2.1).

A continuación mostramos un extracto de ejemplo de un archivo de texto generado como salida en esta fase del análisis:

```
**************************************************
Written analysis file: Output/Analysis - 11 - Scott Joplin -
Elite Syncopations.txt
**************************************************
Scale:
I - F
II - G
III - A
IV - Bb
V - C
VI - D
VII - E
Mode (according to the information of the file): major
```

```
Measure #1
First chord: F

Measure #2
Armonic movement: F to F
Degrees: I to I
Diatonic function: Tonic to Tonic

Measure #3
Armonic movement: F to C
Degrees: I to V
Diatonic function: Tonic to Dominant
Cadence: open

Measure #4
Armonic movement: C to C
Degrees: V to V
Diatonic function: Dominant to Dominant
Cadence: open

Measure #5
Armonic movement: C to Dm7
Degrees: V to VIm7
Diatonic function: Dominant to Tonic
Cadence: deceptive
Armonic movement: Dm7 to F
Degrees: VIm7 to I
Diatonic function: Tonic to Tonic
Armonic movement: F to F#dim
Degrees: I to #Idim
Diatonic function: Tonic to ?

Measure #6
Armonic movement: F#dim to C
Degrees: #Idim to V
Diatonic function: ? to Dominant
Armonic movement: C to C+7
Degrees: V to V+7
Diatonic function: Dominant to Dominant
Armonic movement: C+7 to C+
Degrees: V+7 to V+
Diatonic function: Dominant to Dominant
Cadence: open

Measure #7
Armonic movement: C+ to F
Degrees: V+ to I
Diatonic function: Dominant to Tonic
Cadence: authentic

Measure #8
Armonic movement: F to F
Degrees: I to I
Diatonic function: Tonic to Tonic
Armonic movement: F to Edim
Degrees: I to VIIdim
Diatonic function: Tonic to Dominant
Armonic movement: Edim to C7
Degrees: VIIdim to V7
Diatonic function: Dominant to Dominant
```

```
Cadence: open

Measure #9
Armonic movement: C7 to F
Degrees: V7 to I
Diatonic function: Dominant to Tonic
Cadence: authentic

Measure #10
Armonic movement: F to Gm7
Degrees: I to IIm7
Diatonic function: Tonic to Subdominant
Armonic movement: Gm7 to Bb
Degrees: IIm7 to IV
Diatonic function: Subdominant to Subdominant
Armonic movement: Bb to F
Degrees: IV to I
Diatonic function: Subdominant to Tonic
Cadence: plagal
```

El análisis continúa de manera similar hasta llegar al compás 55, donde detecta una modulación (es decir, un cambio de tono):

```
Measure #55
Modulation detected!!
New scale:
I - Bb
II - C
III - D
IV - Eb
V - F
VI - G
VII - A
Mode (according to the information of the file): major

Armonic movement: Edim to Gm
Degrees: #IVdim to VIm
Diatonic function: Tonic to Tonic
Armonic movement: Gm to Bb
Degrees: VIm to I
Diatonic function: Tonic to Tonic

Measure #56
Armonic movement: Bb to Cm7
Degrees: I to IIm7
Diatonic function: Tonic to Subdominant
Armonic movement: Cm7 to Eb
Degrees: IIm7 to IV
Diatonic function: Subdominant to Subdominant
Armonic movement: Eb to Gm7
Degrees: IV to VIm7
Diatonic function: Subdominant to Tonic
Cadence: plagal
Armonic movement: Gm7 to Bb
Degrees: VIm7 to I
Diatonic function: Tonic to Tonic
```

El análisis sigue hasta el último compás, pero sólo dejamos este fragmento como muestra de la salida de texto del análisis.

3.4.7. Módulo de arreglos

Con el doble objetivo de, por un lado mostrar una aplicación del motor de análisis armónico desarrollado y, por otro lado, cumplir con uno de los objetivos de análisis marcados al comienzo del desarrollo, integrado dentro de la aplicación se ha incluido un módulo que extrae los datos obtenidos durante el análisis armónico y realiza arreglos de acuerdo a las reglas teóricas que se expusieron en el apartado 2.1. Este módulo se basa en el motor de inferencias Drools para aplicar las reglas armónicas adecuadas. Esta tecnología se describió en el apartado 2.3.4. El objetivo de separar el módulo del resto del código Java es, por un lado, demostrar mediante un ejemplo práctico cómo puede accederse a la API desarrollada por parte de un programa externo y, por otro lado, al separar la lógica que aplica el módulo de arreglos del resto del programa dentro de archivos de reglas específicos, se brinda al usuario la posibilidad de editar estas reglas cambiando los archivos de extensión .drl. Esto es de una gran utilidad tanto para depurar las reglas ya existentes como para añadir reglas nuevas, ya que se permite al usuario alterar la funcionalidad de la aplicación mediante el acceso de la lógica destinada únicamente a la creación de arreglos.

Con respecto a Drools, el objetivo de este motor es separar la lógica (el conocimiento armónico) del procesamiento en sí. Las reglas armónicas son información en que se basa el programa, así que se aíslan del código, lo cual facilita la implementación de distintas modalidades de arreglos, simplemente modificando las reglas y sin tener que recompilar o modificar el código.

Como se comentó al final del apartado 3.3, y también en el subapartado 3.4.1, al comienzo de la aplicación se brinda al usuario la posibilidad de escoger alguna modalidad de arreglos automáticos. En caso de que estos arreglos sean escogidos, el módulo de arreglos es invocado después de realizar el análisis de la progresión armónica obtenida por el motor de análisis de la partitura de entrada. En este punto, en vez de finalizar, el programa comienza la aplicación de las reglas sobre arreglos para, una vez realizados éstos, analizar la nueva progresión armónica resultante de la realización de arreglos automáticos y escribir el informe en un nuevo archivo de texto. Por este motivo, cada vez que se analiza una partitura, en el directorio */Output* del programa aparecen dos o cuatro archivos nuevos. Dos en caso de que no se hayan realizado arreglos, uno con la partitura MusicXML que incluye la progresión armónica calculada, y otro con el archivo de texto con el informe del análisis. Dos más en caso de que se hayan realizado arreglos, que corresponderían a la partitura con los arreglos y al informe de la nueva progresión.

Se han desarrollado dos modalidades de arreglos:

- Sustitución simple de acordes.
- Arreglos avanzados.

En la primera, solamente se aplican reglas que tienen por objeto identificar acordes en la progresión con una determinada función tonal (ver apartado 2.1.4) y sustituirlos por otros que desempeñen una función equivalente.

En cuanto a la segunda modalidad, la de *arreglos avanzados*, las reglas, además de sustituir acordes, también añaden acordes nuevos y eliminan acordes redundantes.

Las reglas que se han de aplicar para cada modalidad están escritas en dos archivos de reglas de extensión .drl, uno para cada opción de arreglos.

Sustitución de acordes

El primer bloque de reglas, correspondiente a la modalidad de arreglos sencillos (que únicamente sustituyen acordes por otros con función tonal equivalente) tiene diez reglas, seis para sustituciones del área de tónica, dos para el área de subdominante y otras dos para el área de dominante. Se muestra a continuación a modo de ejemplo el código correspondiente a las dos primeras reglas:

```
import scoreAnalysis.*;

rule "Rule 1 - Tonic area rule Num.1"
    when
      $as : ActualScale( $dI :  dI, $dIII : dIII)
        $sc : SimpleChord( tonic == $dI, $tonic : tonic, type == "",
$name : name, $defX : defX)
        eval(Math.random()>0.4) # Sometimes this rule will not be
applied
        eval($defX > 0) # A tonic chord in the beginning of the
measure should remain the same
    then
        System.out.print(".");
        $sc.setName($dIII+"_m");
        retract($sc);
end

rule "Rule 2 - Tonic area rule Num.2"
    when
      $as : ActualScale( $dI :  dI, $dVI : dVI)
        $sc : SimpleChord( tonic == $dI, $tonic : tonic, type == "",
$name : name, $defX : defX)
        eval(Math.random()>0.7) # Sometimes this rule will not be
applied
        eval($defX > 0) # A tonic chord in the beginning of the
measure should remain the same
    then
        System.out.print(".");
        $sc.setName($dVI+"_m");
        retract($sc);
end
```

Se trata de reglas sencillas que son llamadas en caso de que se cumplan las condiciones descritas. En este caso (atendiendo a la primera regla), estas condiciones son:

- Acorde de tónica, mayor y de primer grado.
- Que no esté justo al comienzo del compás.
- Que un número aleatorio sea mayor que una cierta cantidad.

La primera condición identifica al acorde sobre el que se aplicará la regla que se codifica en las sentencias entre la palabra clave *then* y la palabra clave *end*. La segunda se ha incluido porque se ha considerado que un acorde que esté al comienzo del compás, que sea de tónica, y cuya tónica además sea el primer grado de la escala, no debería sustituirse, ya que este acorde representa los sonidos de mayor estabilidad en una pieza musical, y cambiarlos cuando están al principio del compás no será acertado en la

mayoría de los casos. Esta condición no se ha incluido nada más que en estos dos casos (acordes del área de tónica y del primer grado). En cuanto a la última condición, se ha puesto para que la sustitución no se haga en todos los casos (ya que eso cambiaría toda la progresión armónica), sino sólo algunas veces, de forma aleatoria, para que la progresión original se mantenga en esencia y sólo se cambien algunos acordes. Esta condición de aleatoriedad se ha incluido en todas las reglas, por lo que no volverá a explicarse. Cuando una regla es aplicada sobre un determinado hecho (en este caso los hechos son los acordes de la progresión armónica), este hecho es retirado del motor de inferencias para evitar que puedan aplicársele más reglas.

Inserción y eliminación de acordes

En cuanto al segundo bloque de reglas, se ha implementado una funcionalidad más complicada que hace mayor uso del potencial del motor de inferencias de Drools. En concreto, se han codificado once reglas. Pasamos a describirlas a continuación:

La primera regla es:

```
rule "Rule 1 - Chords addition"
    when
      $pm : ProcMeasure($selectedChords : selectedChords)
      # New chords will only be added if the measure contains one or
two chords.
      eval( $selectedChords.size() > 0 && $selectedChords.size() < 3 )
    then
      insert(new ArranMeasure($pm));
      retract($pm);
end
```

Esta primera regla revisa entre los hechos a los compases procesados de la API del motor de análisis (objetos de la clase *ProcMeasure*). Cuando detecta que hay uno o dos acordes en el compás, considera que puede ser adecuado añadir más acordes al compás en cuestión. Para ello, añade un nuevo hecho al motor. Este nuevo hecho es de una clase específica que se ha creado para identificar compases que van a recibir arreglos. El compás de la clase *ProcMeasure* es extraído de los hechos.

Pasamos a describir la segunda regla:

```
rule "Rule 2 - Chords disambiguation"
    when
      $pm : ProcMeasure($selectedChords : selectedChords)
      # Disambiguation will only be applied when no chord has been
found.
      eval( $selectedChords.size() == 0)
    then
      insert(new EmptyMeasure($pm));
      retract($pm);
end
```

Esta regla se encarga de lo que hemos llamado desambiguación en los compases procesados. Se aplica para compases en los que el motor de análisis no ha reconocido acorde alguno. Es posible que aparezca únicamente un intervalo en vez de acordes completos entre las notas del compás. En este caso, se procurará encontrar posibles acordes que encajen para las notas que presenta el compás y, en caso de encontrar una coincidencia, incluirla en el compás. El único cometido de esta regla es por tanto identificar compases sin ningún acorde, e introducir esto como un nuevo hecho a ser

tratado por otras reglas, las cuales se encargarán de identificar qué acorde puede encajar con las notas del compás. Mostramos a continuación un ejemplo (mediante la descripción de la regla número 3) de cómo funcionan estas reglas para la desambiguación (reglas de la 3 a la 8):

```
rule "Rule 3 - Major chord for major third with tonic in I"
    when
      $em : EmptyMeasure()
      eval( $em.checkInterv("3maj", "I")>0)
            $as : ActualScale()
    then
      $em.getPm().addSelChord(new
SimpleChord($as.getdI()+"_",$em.checkInterv("3maj", "I")));
      retract($em);
end
```

Como vemos, toma como hechos los compases vacíos introducidos por la regla analizada anteriormente. Después, comprueba que se produzca un intervalo (o más) de tercera mayor con respecto a la tónica de la escala. En caso afirmativo, la regla considera que puede ser adecuado un acorde mayor de primer grado en ese compás, por lo que lo añade y elimina el hecho para evitar que éste sea considerado por más reglas. Al ser ésta la primera regla de la serie, tendrá prevalencia sobre las demás, ya que se comprobará antes. Hay que tener en cuenta que si se consideran las reglas de desambiguación, es porque no se ha encontrado ningún acorde conocido que cuadre con las notas que aparecen en el compás, de modo que es de esperar que haya poca variedad disponible de notas. Es por esto que se considera condición suficiente para añadir el acorde que aparezca una tercera (aunque en la mayoría de los casos esto dé lugar a análisis correctos, podría añadirse una nueva condición que verificara además si alguna de las notas del acorde a añadir entrara en disonancia con las notas del compás). El resto de reglas de este grupo (de la 4 a la 8) son muy similares a ésta (teniendo también por objetivo buscar un acorde que cuadre para las notas de un compás para el que no se haya detectado acorde alguno por parte del motor de análisis), por lo que a continuación se describe la regla 9:

```
rule "Rule 9 - Arrangements for measures with 1 or 2 chords. A gap is
present before a chord."
      when
            $am : ArranMeasure($pm : pm)
            $as : ActualScale()
            eval( $am.prevGap() && !$am.finished())
      then
            if($am.nextChord().getName().equals("X_X")){
                  $am.addPrev("X_X");
            }else if (Math.random() > 0.3){
                  $am.addPrev(getChordFromArea($as, $am.nextChord()));
            }
            else if ($am.nextChord().getType().equals("7") &&
Math.random() > 0.3) {
                  // If there's a seventh-type chord, a II-7 relative
could be an option
                  $am.addPrev(II7($as, $am.nextChord()));
            }else{
                  // A secondary dominant chord could also be another
choice.
                  $am.addPrev(secondaryDominant($as, $am.nextChord()));
            }
```

```
end
```

Esta regla se aplica compases que tienen uno o dos acordes (detectados mediante la regla 1). En primer lugar, detecta que haya un hueco donde pueda colocarse un nuevo acorde antes de alguno de los acordes presentes, y en caso afirmativo genera un nuevo acorde teniendo en cuenta para la generación al acorde al cual precederá. La regla 10 realiza la operación equivalente para el caso en que un hueco aparezca después de un acorde. Por último, la regla 11 retira como hechos aquellos compases cuyos huecos rellenables están completados.

3.5. Desarrollo en Eclipse

Por último, para facilitar la integración con futuros proyectos, se comentará cómo se ha organizado la aplicación en un proyecto de Eclipse, y cómo ésta es puesta a disposición del usuario mediante una biblioteca .jar.

El proyecto de Eclipse, está en una ruta que consta del siguiente árbol de directorios:

\
\bin
\bin\rules
\bin\scoreAnalysis
\bin\scoreAnalysis\util
\doc
\doc\[…] (Varios subdirectorios generados por la aplicación javadoc)
\lib
\lib\Drools
\Output
\src
\src\scoreAnalysis
\src\scoreAnalysis\util

Dentro de \bin y sus subdirectorios se almacenan los archivos compilados, excepto en \bin\rules, donde se almacenan los archivos con las reglas usadas por el módulo de arreglos desarrollado para el motor de Drools.

El directorio \doc, así como sus subdirectorios, están dedicados a la documentación de la aplicación.

Dentro del directorio \lib y sus subdirectorios se almacenan las bibliotecas *.jar* que son necesarias para el funcionamiento del programa.

En el directorio \Output se guardan los archivos de salida del programa. Éstos pueden ser de dos tipos: archivos con partituras en formato MusicXML, de extensión .xml y archivos de texto con un informe escrito del análisis realizado, en formato .txt.

En el directorio \src, así como en sus subdirectorios, se encuentra el código fuente de la aplicación.

Posteriormente, una vez estaba terminada la fase de desarrollo, se ha elaborado un archivo .jar para que el programa pueda ser arrancado fuera del entorno de Eclipse. El archivo .jar, junto con las bibliotecas y archivos que éste utiliza se distribuye dentro de un archivo comprimido de extensión .rar. Este archivo contiene todo lo necesario para que, al descomprimirlo, la aplicación pueda ser arrancada en cualquier ordenador que disponga de máquina virtual de java. Detallamos a continuación el contenido de dicho archivo:

\ScoreAnalysis.jar
\run.bat
\rules\AddChords.drl
\rules\ChangeChords.drl
\ScoreAnalysis_lib\...
\Output\

El archivo ScoreAnalysis.jar deberá ser ejecutado mediante la siguiente llamada para arrancar la aplicación:

java -jar ScoreAnalysis.jar

El archivo run.bat es un archivo por lotes para DOS/Windows que contiene esta llamada para ejecutar la aplicación (un script similar podría haberse realizado para Linux, ya que la biblioteca .jar es independiente del sistema operativo siempre y cuando éste disponga de la máquina virtual).

Dentro de la carpeta *rules,* están los archivos con las reglas usadas por el módulo de arreglos, las cuales pueden ser editadas por el usuario. La carpeta *ScoreAnalysis_lib* contiene las bibliotecas necesarias para que la aplicación funcione (aquellas usadas por proxyMusic y Drools). La carpeta *Output* está vacía, pero es necesaria ya que en ella se escribirán los archivos de salida de la aplicación.

4. Evaluación

A continuación se mostrará una batería de pruebas realizadas sobre el motor de análisis, así como sobre el módulo de arreglos. En primer lugar, se comenzará con el análisis de partituras sencillas con propósitos específicos, las cuales se irán complicando de forma progresiva. Para la creación de estas partituras, así como para la lectura tanto de las partituras de entrada como de las generadas por la aplicación de análisis, se ha utilizado el editor de partituras de código abierto *MuseScore* [24].

4.1. Pruebas sobre el motor de análisis

4.1.1. Reconocimiento de acordes

Reconocimiento de acordes explícitos en tonalidad de C mayor

En primer lugar, se ha probado la detección de acordes explícitos en tono de C mayor, tomando como entrada la siguiente partitura:

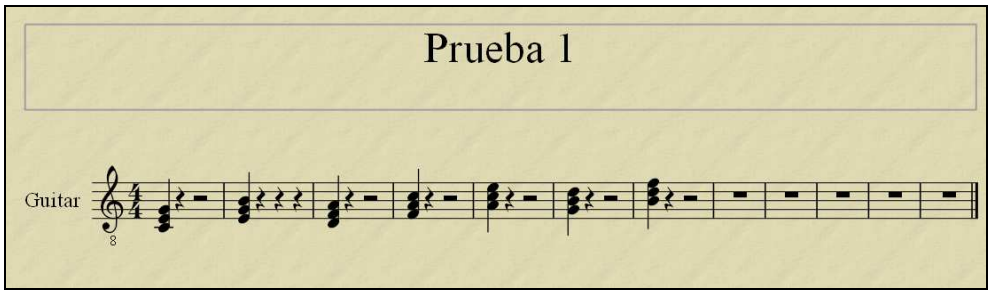


Figura 27 – Partitura de la primera prueba, antes de ser procesada por la aplicación

Como vemos, al principio de cada compás se ha escrito una tríada que se corresponde con un acorde perteneciente a la escala de C mayor. Tras aplicar el motor de análisis (sin utilizar el módulo de arreglos), obtuvimos el siguiente resultado como salida en la consola:

```
Opening score file named: 01 .xml
Please wait...
Score succesfully imported from E:\Mis Documentos\Archivos
de prueba\Pruebas\01 .xml
The analysis is in process...
Written analysis saved to the file: Output/Analysis - 01
.txt
End of analysis.
Score succesfully exported to Output\Analysis - 01 .xml
```

Al abrir el archivo Analysis – 01 .xml, pudimos leer la siguiente partitura:

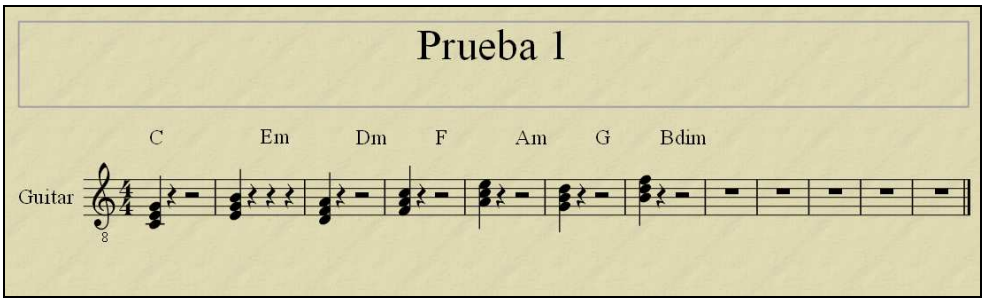


Figura 28 – Análisis de acordes explícitos en tono de C (primera partitura de prueba, tras el análisis)

Como podemos comprobar, el archivo de salida es exactamente igual al de entrada con la salvedad de que se ha añadido la información acerca de la armonía que aparece en la partitura. En este caso, el motor de análisis ha identificado correctamente todas las tríadas de la escala mayor de C, por lo que los resultados de esta prueba han sido satisfactorios.

Reconocimiento de acordes explícitos en una tonalidad distinta de C mayor

Para la segunda prueba hemos diseñado una partitura similar a la de la prueba 1, pero en otra escala. En este caso la escala es de A, que contiene las siguientes notas:

A B C# D E F# G#

Tras analizar el archivo de entrada (únicamente mostramos ya los archivos de salida ya que son idénticos a los de entrada, sólo que contienen información sobre la armonía), podemos ver como también en este caso el análisis es satisfactorio, ya que es capaz de identificar correctamente las tríadas en escalas diferentes de las de C mayor:

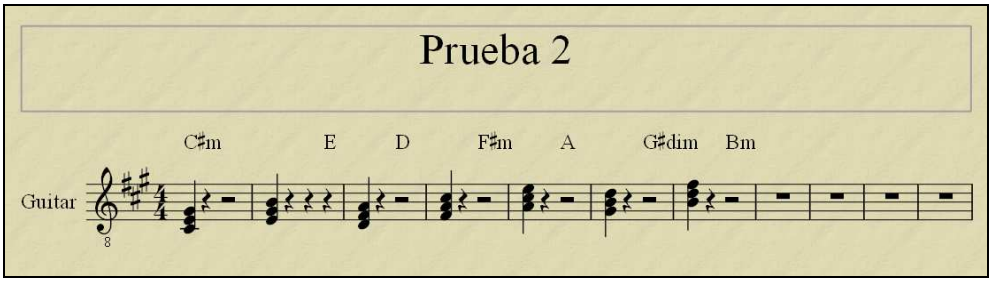


Figura 29 – Detección de acordes en otra escala (escala de A)

Compases con acordes incompletos

La siguiente prueba consiste en mostrar acordes *incompletos*, es decir, tomando la partitura de la segunda prueba, hemos eliminado notas para ver que, al no aparecer una tríada completa en un determinado compás, el motor de análisis no muestra ningún resultado:

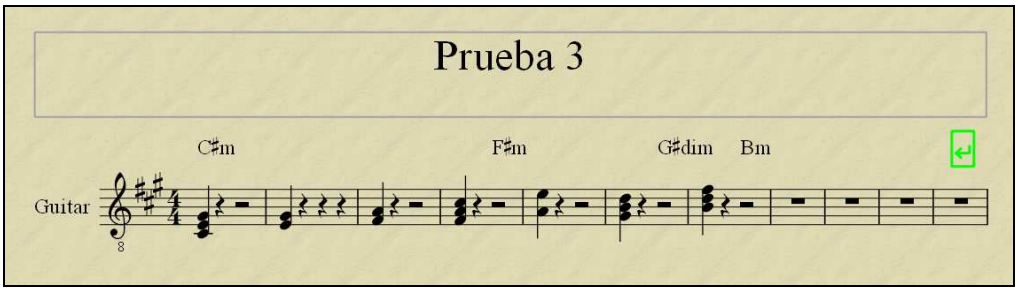


Figura 30 – Compases con acordes incompletos

Esto ocurre porque no se ha creado dentro del motor de análisis un módulo para generar acordes que cuadren con una determinada melodía, sino que solamente proporciona información que puede extraerse directamente de las notas de cada compás, sin necesidad de atender al contexto. Como la información del contexto es precisamente la que se obtiene tras la aplicación del motor de análisis, se ha considerado que la desambiguación de acordes debe hacerse por un módulo externo al motor de análisis armónico, que tenga en cuenta toda la información del análisis proporcionada por éste.

Dado que se considera importante que la aplicación sea capaz de reconocer acordes en estos compases que no presentan tríadas completas, en el módulo de arreglos, dentro del segundo grupo de reglas, se ha añadido una *lógica de desambiguación*, que identifica tríadas en los casos en los que en la partitura aparezcan incompletas, como en este ejemplo (ver apartado 4.2).

Reconocimiento de acordes de séptima

En la siguiente prueba se ha probado la identificación de acordes con séptima, también en la escala de C mayor:

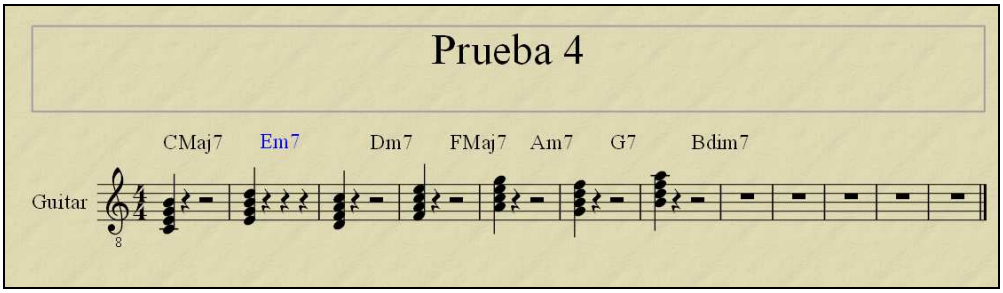


Figura 31 – Análisis de acordes con séptima

Como podemos comprobar, el motor de análisis reconoce correctamente los acordes con séptima que aparecen en la partitura de la cuarta prueba. Para poder realizar esto fue necesario que la aplicación diera prioridad a este tipo de acordes frente a las tríadas, ya que dentro de cada acorde de séptima pueden encontrarse además dos tríadas. Por ejemplo, el acorde de CMaj7 está formado por las siguientes notas:

C, E, G, B

Si tomamos las tres primeras, ignorando la última, estaríamos ante una tríada de C mayor (la cual es detectada por el motor de análisis, sólo que, al haber detectado también un acorde de séptima, ésta es descartada). Además, si se ignora la primera nota, con las tres últimas: E, G y B, también se forma una tríada (de Em), la cual también ha

sido descartada por el motor en favor del acorde CMaj7, el cual tiene un significado más importante en términos de armonía dentro de una composición que cualquiera de las otras dos tríadas.

Reconocimiento de acordes implícitos en tonalidad de C mayor

En la siguiente prueba, pasamos a demostrar cómo los acordes implícitos, es decir, aquellos que no aparecen como notas simultáneas, sino dentro de un arpegio, también son correctamente identificados por la aplicación:

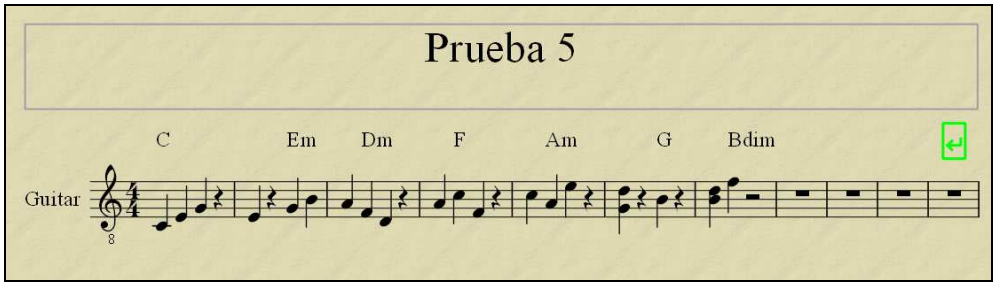


Figura 32 – Pruebas de detección de acordes implícitos

Una vez más, la aplicación ha identificado satisfactoriamente los acordes que aparecían en la partitura.

Reconocimiento de acordes implícitos en una partitura que incluye una modulación

La siguiente prueba consiste en identificar acordes implícitos en una partitura en la que además hay una modulación, es decir, un cambio de tono. En la partitura aparece una melodía arpegiada en la escala de C mayor, para después dar lugar a una nueva melodía, pero en escala de E mayor:

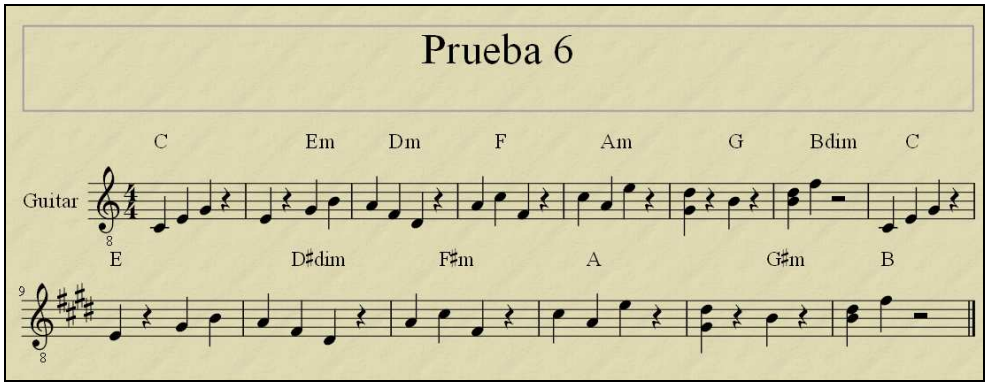


Figura 33 – Detección de acordes implícitos en partituras con modulaciones

Vemos cómo la aplicación ha sido capaz de detectar la modulación, identificando correctamente los acordes de la escala de E mayor, teniendo en cuenta que ésta presenta cuatro sostenidos (F#, C#, G# y D#).

Reconocimiento de acordes en partituras con varias partes

La siguiente partitura sirve para comprobar cómo el motor es capaz de tomar las distintas partes de una partitura (en este caso para cuatro voces) y analiza la armonía de la obra tomándolas todas de forma conjunta:

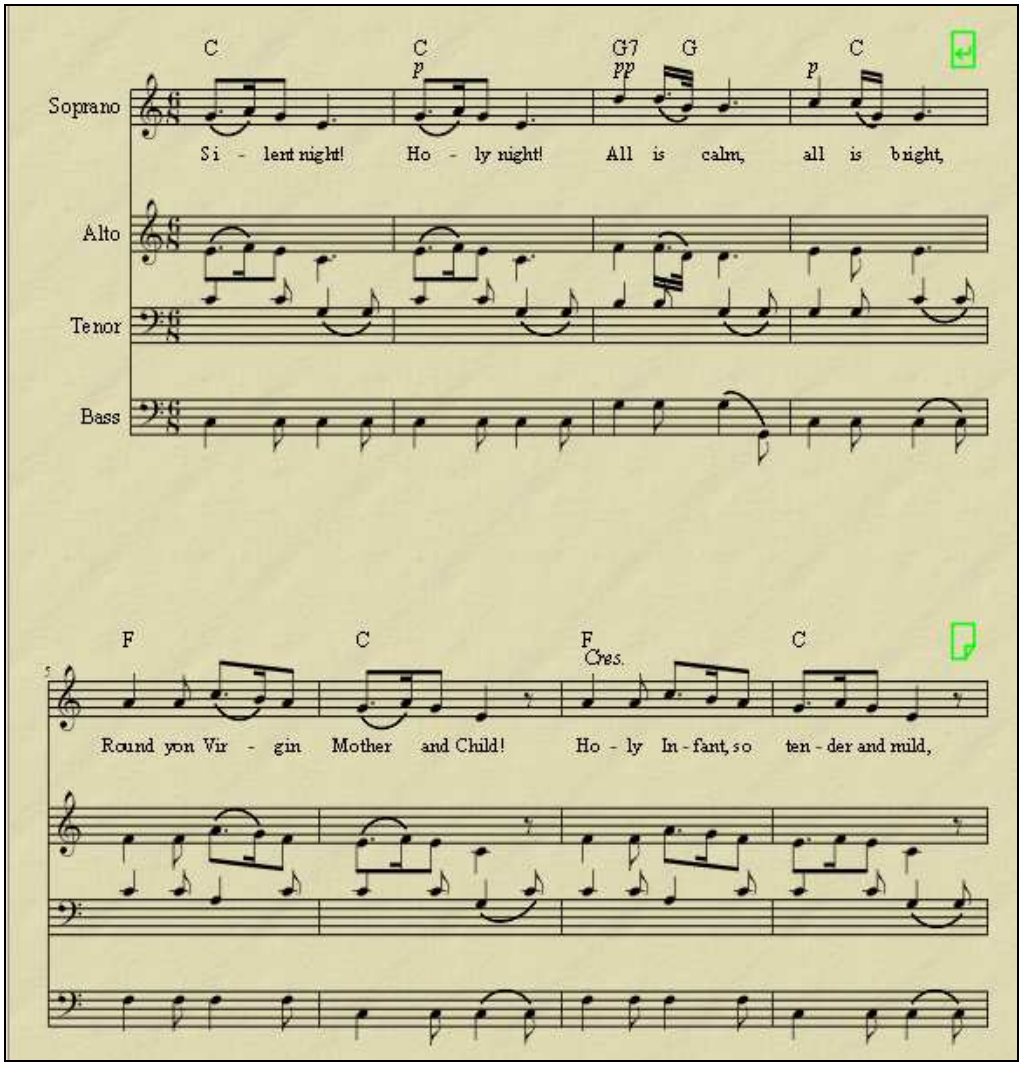


Figura 34 – Análisis de una partitura con varias partes

Al tratarse de partes monofónicas (ya que la voz humana no puede emitir dos notas distintas al mismo tiempo), si cada parte se analizara por separado, la aplicación no estaría teniendo en cuenta muchos de los acordes que se forman por la reproducción simultánea de las partes, por lo que el análisis estaría incompleto. No obstante, dado que se ha tenido en cuenta la posibilidad de que en una partitura aparezcan distintas partes, y éstas deben ser analizadas de forma conjunta, el análisis reconoce correctamente los acordes que aparecen en la obra.

4.1.2. Análisis de la progresión armónica

A continuación mostraremos los resultados obtenidos por la parte del motor de análisis que se encarga de realizar un análisis en texto de la partitura de entrada. El objetivo de este análisis es identificar qué cadencias aparecen entre acordes consecutivos, y analizar la función tonal de cada acorde dentro de la obra.

Uno de los casos más sencillos para comprobar buena parte de la funcionalidad del análisis realizado en texto es la partitura de la prueba 1 de los acordes explícitos (Figura 28), ya que aparecen todos los acordes de la escala, y se puede comprobar la función tonal de cada uno de ellos.

Tras la ejecución del análisis, obtenemos el siguiente resultado:

```
**************************************************
Written analysis file: Output/Analysis - 01 .txt
**************************************************
Scale:
I - C
II - D
III - E
IV - F
V - G
VI - A
VII - B
Mode (according to the information of the file): major

Measure #1
First chord: C

Measure #2
Armonic movement: C to Em
Degrees: I to IIIm
Diatonic function: Tonic to Tonic

Measure #3
Armonic movement: Em to Dm
Degrees: IIIm to IIm
Diatonic function: Tonic to Subdominant

Measure #4
Armonic movement: Dm to F
Degrees: IIm to IV
Diatonic function: Subdominant to Subdominant

Measure #5
Armonic movement: F to Am
Degrees: IV to VIm
Diatonic function: Subdominant to Tonic
Cadence: plagal

Measure #6
Armonic movement: Am to G
Degrees: VIm to V
Diatonic function: Tonic to Dominant
Cadence: open

Measure #7
Armonic movement: G to Bdim
Degrees: V to VIIdim
Diatonic function: Dominant to Dominant
Cadence: open
```

Vemos que la función de cada acorde es identificada correctamente, de acuerdo con la teoría que expusimos en los apartados 2.1.4 y 2.1.6.

A continuación se muestra una prueba en la que se incluye además una modulación, para lo que se toma de nuevo como entrada la partitura de la Figura 33. El resultado del análisis es el siguiente:

```
**************************************************
Written analysis file: Output/Analysis - 06.txt
**************************************************
Scale:
I - C
II - D
III - E
IV - F
V - G
VI - A
VII - B
Mode (according to the information of the file): major

Measure #1
First chord: C

Measure #2
Armonic movement: C to Em
Degrees: I to IIIm
Diatonic function: Tonic to Tonic

Measure #3
Armonic movement: Em to Dm
Degrees: IIIm to IIm
Diatonic function: Tonic to Subdominant

Measure #4
Armonic movement: Dm to F
Degrees: IIm to IV
Diatonic function: Subdominant to Subdominant

Measure #5
Armonic movement: F to Am
Degrees: IV to VIm
Diatonic function: Subdominant to Tonic
Cadence: plagal

Measure #6
Armonic movement: Am to G
Degrees: VIm to V
Diatonic function: Tonic to Dominant
Cadence: open

Measure #7
Armonic movement: G to Bdim
Degrees: V to VIIdim
Diatonic function: Dominant to Dominant
Cadence: open

Measure #8
Armonic movement: Bdim to C
Degrees: VIIdim to I
Diatonic function: Dominant to Tonic
```

```
Measure #9
Modulation detected!!
New scale:
I - E
II - F#
III - G#
IV - A
V - B
VI - C#
VII - D#
Mode (according to the information of the file): major

Armonic movement: C to E
Degrees: bVI to I
Diatonic function: ? to Tonic

Measure #10
Armonic movement: E to D#dim
Degrees: I to VIIdim
Diatonic function: Tonic to Dominant
Cadence: open

Measure #11
Armonic movement: D#dim to F#m
Degrees: VIIdim to IIm
Diatonic function: Dominant to Subdominant

Measure #12
Armonic movement: F#m to A
Degrees: IIm to IV
Diatonic function: Subdominant to Subdominant

Measure #13
Armonic movement: A to G#m
Degrees: IV to IIIm
Diatonic function: Subdominant to Tonic
Cadence: plagal

Measure #14
Armonic movement: G#m to B
Degrees: IIIm to V
Diatonic function: Tonic to Dominant
Cadence: open
**************************************************
End of analysis
**************************************************
```

Como podemos ver, en el compás 9, la aplicación detecta la modulación, muestra la nueva escala y realiza el análisis armónico de acuerdo con ella.

4.1.3. Pruebas con partituras complejas

A continuación se mostrarán los resultados de analizar una partitura compleja mediante el motor de análisis. La partitura escogida es la obra titulada “Ellite syncopations”, del compositor estadounidense Scott Joplin. Esta partitura ha sido descargada de la página web de *Recordare* [28]. Tras realizar el análisis, obtuvimos el siguiente resultado (mostramos sólo la primera página, ya que consideramos innecesario mostrar los resultados de las ocho páginas de la partitura):

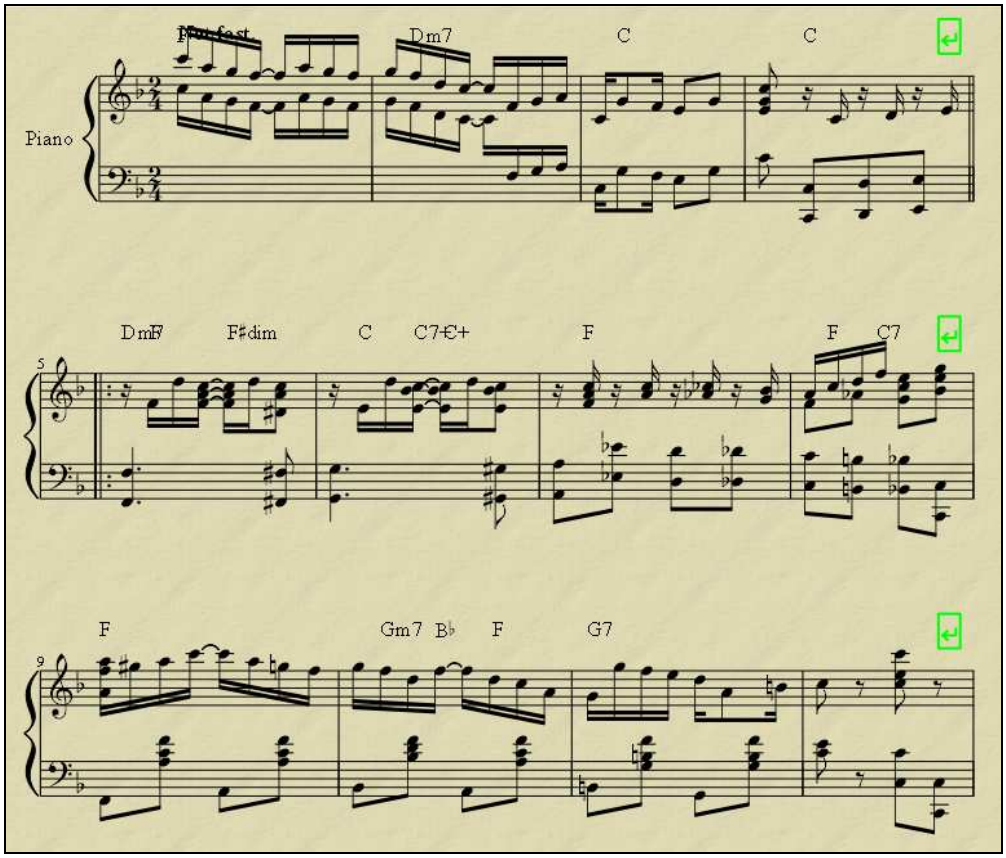


Figura 35 – Análisis de una partitura compleja

Como podemos comprobar una vez leída la partitura, los acordes escogidos en el análisis son correctos para las notas que en ella aparecen. En el último compás, al no haber una tríada completa ni, por ende, un acorde de séptima, el motor no presenta ningún resultado. Por ello, este archivo se volverá a utilizar para demostrar el funcionamiento de las reglas de desambiguación del módulo de arreglos, en el apartado 4.2. En cuanto al análisis en texto de la partitura, mostramos a continuación el resultado para los 12 primeros compases (los que se ven en la **Figura 35**):

```
**********************************************************
Written analysis file: Output/Analysis - 08 - Elite Syncopations.txt
**********************************************************
Scale:
I - F
II - G
III - A
IV - Bb
V - C
VI - D
VII - E
Mode (according to the information of the file): major

Measure #1
First chord: F

Measure #2
Armonic movement: F to Dm7
Degrees: I to VIm7
```

Diatonic function: Tonic to Tonic

Measure #3
Armonic movement: Dm7 to C
Degrees: VIm7 to V
Diatonic function: Tonic to Dominant
Cadence: open

Measure #4
Armonic movement: C to C
Degrees: V to V
Diatonic function: Dominant to Dominant
Cadence: open

Measure #5
Armonic movement: C to Dm7
Degrees: V to VIm7
Diatonic function: Dominant to Tonic
Cadence: deceptive
Armonic movement: Dm7 to F
Degrees: VIm7 to I
Diatonic function: Tonic to Tonic
Armonic movement: F to F#dim
Degrees: I to #Idim
Diatonic function: Tonic to ?

Measure #6
Armonic movement: F#dim to C
Degrees: #Idim to V
Diatonic function: ? to Dominant
Armonic movement: C to C+7
Degrees: V to V+7
Diatonic function: Dominant to Dominant
Armonic movement: C+7 to C+
Degrees: V+7 to V+
Diatonic function: Dominant to Dominant
Cadence: open

Measure #7
Armonic movement: C+ to F
Degrees: V+ to I
Diatonic function: Dominant to Tonic
Cadence: authentic

Measure #8
Armonic movement: F to F
Degrees: I to I
Diatonic function: Tonic to Tonic
Armonic movement: F to C7
Degrees: I to V7
Diatonic function: Tonic to Dominant
Cadence: open

Measure #9
Armonic movement: C7 to F
Degrees: V7 to I
Diatonic function: Dominant to Tonic
Cadence: authentic

Measure #10

Armonic movement: F to Gm7
Degrees: I to IIm7
Diatonic function: Tonic to Subdominant
Armonic movement: Gm7 to Bb
Degrees: IIm7 to IV
Diatonic function: Subdominant to Subdominant
Armonic movement: Bb to F
Degrees: IV to I
Diatonic function: Subdominant to Tonic
Cadence: plagal

Measure #11
Armonic movement: F to G7
Degrees: I to II7
Diatonic function: Tonic to ?

Measure #12

Vemos en este análisis que, al tratarse de una obra más compleja, se detectan algunos acordes que se escapan a lo que puede ser directamente extraído de la escala sin realizar alteraciones adicionales. Entre ellos, estaría el quinto grado séptima con la quinta aumentada (V+7), el cual es introducido en el motor de análisis como un caso habitual de distanciamiento de las notas de la escala, por lo que se trata como una excepción conocida, y es reconocido como un acorde de dominante. No obstante, también aparecen acordes que se escapan a la capacidad de análisis de la aplicación, como ocurre con el segundo grado séptima del compás 11. En este caso, la aplicación muestra una interrogación, ya que ignora qué función tonal tiene dicho acorde en la escala.

4.2. Pruebas sobre el módulo de arreglos

Por último, se han realizado algunas pruebas que han tenido por objeto comprobar el correcto funcionamiento del módulo de arreglos automáticos.

Cuando la aplicación se ejecuta habiendo seleccionado un módulo de arreglos, la salida en la consola para el caso de un análisis sin errores tiene el siguiente aspecto:

```
Opening score file named: 01 .xml
Please wait...
Score succesfully imported from E:\Mis Documentos\Archivos
de prueba\Pruebas\01 .xml
The analysis is in process...
Written analysis saved to the file: Output/Analysis - 01
.txt
Starting the arrangements phase...
.......
Finish of the arrangements phase.
Written analysis saved to the file: Output/Arrangements
Analysis - 01 .txt
End of analysis.
Score succesfully exported to Output\Analysis - 01 .xml
```

```
Score with arrangements succesfully exported to
Output\Arrangements Analysis - 01 .xml
```

Arreglos sencillos (sustitución de acordes)

La primera prueba que se ha realizado es sobre la modalidad de arreglos sencillos, basados en sustitución de acordes. Aplicándola a la partitura de la prueba 1, obtenemos el siguiente resultado:

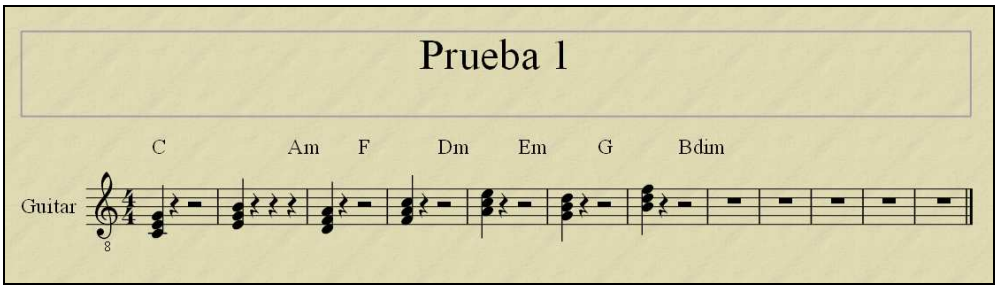


Figura 36 – Sustitución simple de acordes sobre la partitura de la prueba 1

Como puede verse, se realiza una sustitución sencilla basándose en la función tonal del acorde original por un acorde de la misma área. En este caso se han sustituido, en el segundo compás, Em por Am, ya que ambos son acordes del área de tónica; en el tercer compás, Dm por F, ya que se trata de acordes del área de subdominante; la sustitución inversa se ha realizado en el cuarto compás. Por último, en el compás quinto, se ha sustituido Am por Em. El resto de acordes se han dejado inalterados, ya que estas sustituciones se aplican sólo en algunos casos, basándose en una probabilidad fijada en cada regla (ver apartado 3.4.7 para más información).

Arreglos avanzados (adición, sustitución y eliminación de acordes)

Aplicando la modalidad de arreglos avanzados sobre la misma partitura obtuvimos el siguiente resultado:

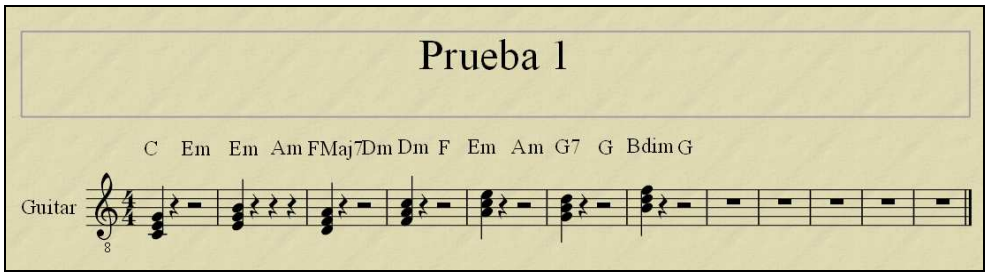


Figura 37 – Arreglos avanzados sobre la prueba 1

Como vemos, la modalidad de arreglos avanzados ha introducido nuevos acordes sobre los que ya habían sido detectados por el motor de análisis. En el primer compás, junto al C mayor, añade un Em, ya que se trata de acordes de tónica. Lo mismo ocurre en el compás segundo, tercero cuarto y quinto, con Em y Am, FMaj7 y Dm, Dm y F, y Em y Am respectivamente. En el compás 6 lo que hace es añadir un acorde de quinto grado con séptima (dominante) precediendo G. En el último compás añade otro acorde de dominante al Bdim que ya había.

Hay que destacar que uno de los problemas con los que nos encontramos a la hora de representar las partituras fue que en ocasiones los acordes introducidos se solapaban entre sí. Esto varía en función de la herramienta utilizada para la representación. Para evitar solapamientos, la aplicación ha hecho uso de dos estrategias: en primer lugar, existe un campo en cada objeto (notas, acordes, etc) llamado DefX, que indica la posición horizontal de dicho objeto en el compás. Por otro lado, también se tiene en cuenta la ordenación de los objetos dentro de los elementos del compás, de forma que si un objeto se representa más a la izquierda que otro, aparece normalmente (aunque no es requisito indispensable) antes en el archivo MusicXML que otro que debe representarse más a la derecha. Pese a que se ha hecho del campo DefX y se han ordenado correctamente los elementos introducidos dentro del árbol de elementos MusicXML, con determinados visualizadores hemos encontrado problemas de solapamientos. En concreto, con MuseScore, que es el utilizado para representar estas partituras, se hace uso de la ordenación interna de los elementos, ignorando en muchas ocasiones el valor del campo DefX. Esto da lugar a que, cuando ha sido necesario representar dos acordes en un compás pero sólo había una posición con notas (como en el caso de la **Figura 37** que ahora nos ocupa), con este programa los acordes introducidos aparecían solapados, y era necesario separarlos manualmente. El mismo archivo abierto con otros editores no ha presentado este problema. No obstante, el editor escogido finalmente fue MuseScore porque, además de ser libre y de código abierto, en la mayoría de los casos las partituras se representaban correctamente.

Sustitución de acordes en tonalidad distinta a la de C mayor

La siguiente prueba consiste en sustituir acordes de una escala distinta a la de C mayor:

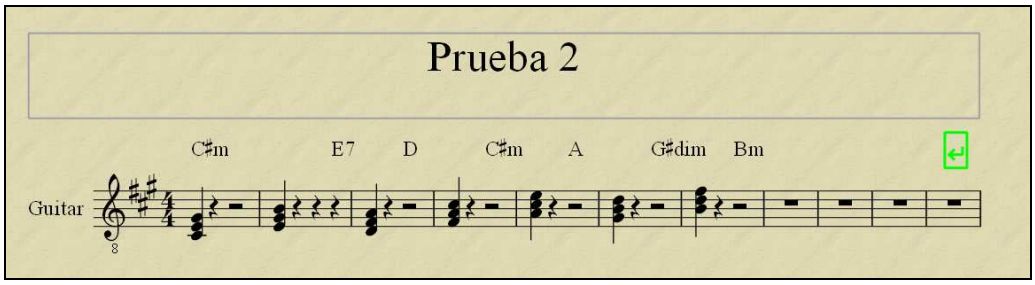


Figura 38 – Sustitución de acordes sobre la partitura de la prueba 2

Como podemos ver, la sustitución es equivalente a la de la primera prueba, pero en la escala que corresponde a la nueva partitura, por lo que el resultado es satisfactorio.

Arreglos avanzados en tonalidad distinta a la de C mayor

A continuación realizamos la misma prueba pero sobre el módulo de arreglos avanzados:

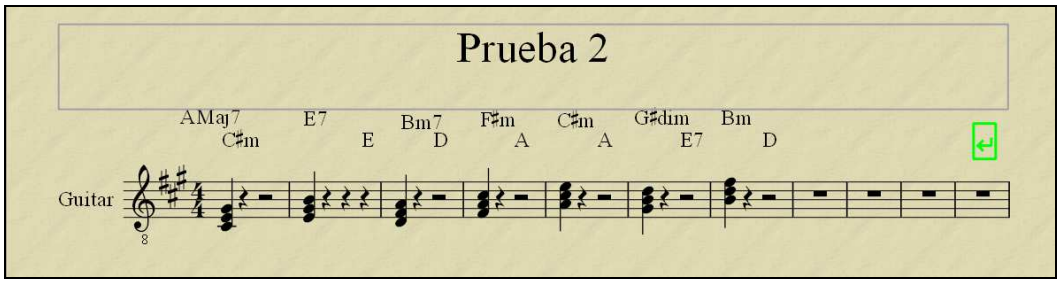


Figura 39 – Arreglos avanzados sobre la partitura de la prueba 2

Nuevamente el resultado coincide con lo esperado tras el análisis de la prueba 1, obteniendo unos arreglos similares.

Desambiguación de acordes

La prueba siguiente está enfocada a demostrar el funcionamiento de las reglas de desambiguación de la modalidad de arreglos avanzados, donde incluso acordes que aparecen incompletos en la partitura son detectados:

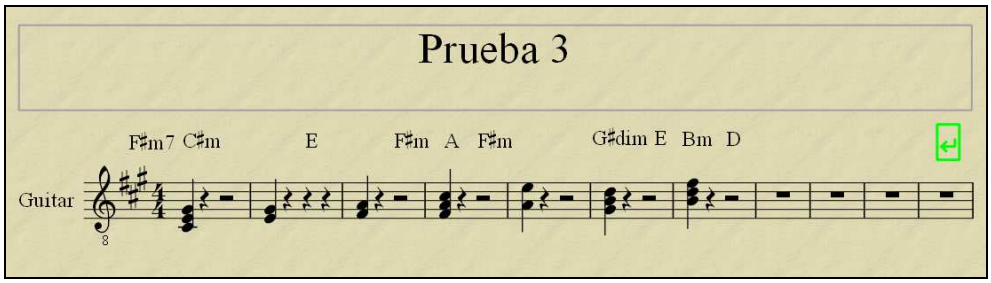


Figura 40 – Arreglos avanzados sobre la partitura de la prueba 3

En esta partitura hemos vuelto a tener los problemas de solapamiento que mencionábamos anteriormente. En cuanto a la desambiguación, podemos ver que en el segundo compás, las notas E y G# son interpretadas como pertenecientes a un acorde de E, por lo que éste es introducido en el compás, rellenando el blanco que aparecía tras la aplicación del motor de análisis. De la misma forma, en el compás 3 es detectado el acorde de F#m. No ocurre lo mismo con el quinto compás, ya que al no haber un intervalo de tercera, sino que simplemente aparece una quinta, no se cumplen las reglas escritas para la desambiguación, y ningún acorde es detectado. En el resto de casos, donde había acordes ya detectados por el motor de análisis, las reglas habituales de arreglos avanzados son aplicadas, añadiendo en este caso acordes nuevos con la misma función tonal.

Sustitución de acordes en una partitura compleja

A continuación, la partitura compleja que se utilizó en las últimas pruebas del motor de análisis es aplicada a la modalidad de sustituciones sencillas:

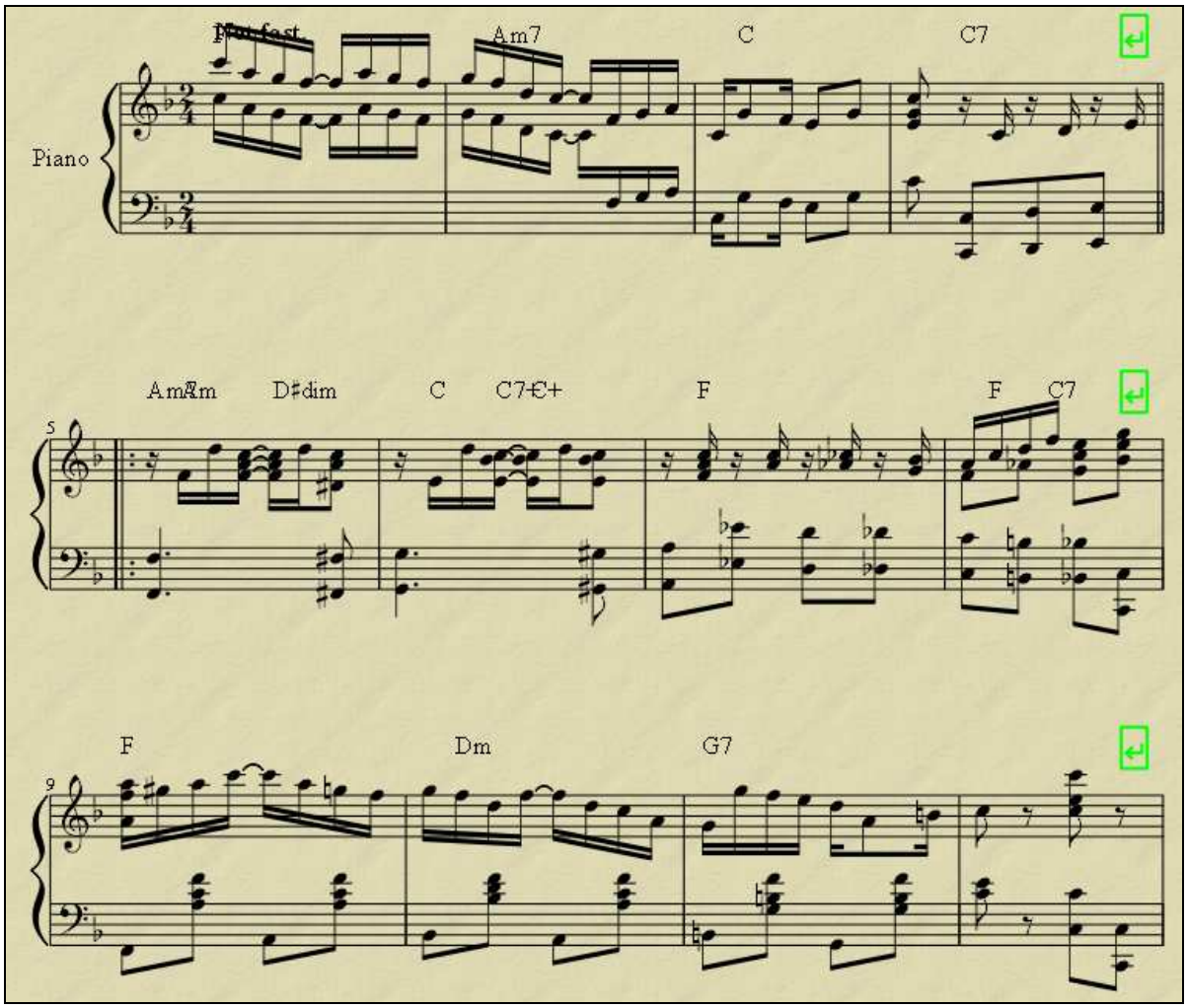


Figura 41 – Sustitución de acordes sobre una partitura compleja

De forma similar a las partituras de las pruebas anteriores, el módulo de sustituciones básicas de acordes actúa sobre una parte de los acordes cuya función tonal es reconocida, sustituyéndolos por acordes equivalentes.

Arreglos avanzados en una partitura compleja

Por último, la modalidad de arreglos avanzados es aplicada sobre esta misma partitura, para comprobar cómo el hueco dejado por el motor de análisis en el compás 14 es rellenado ahora por las reglas de desambiguación:

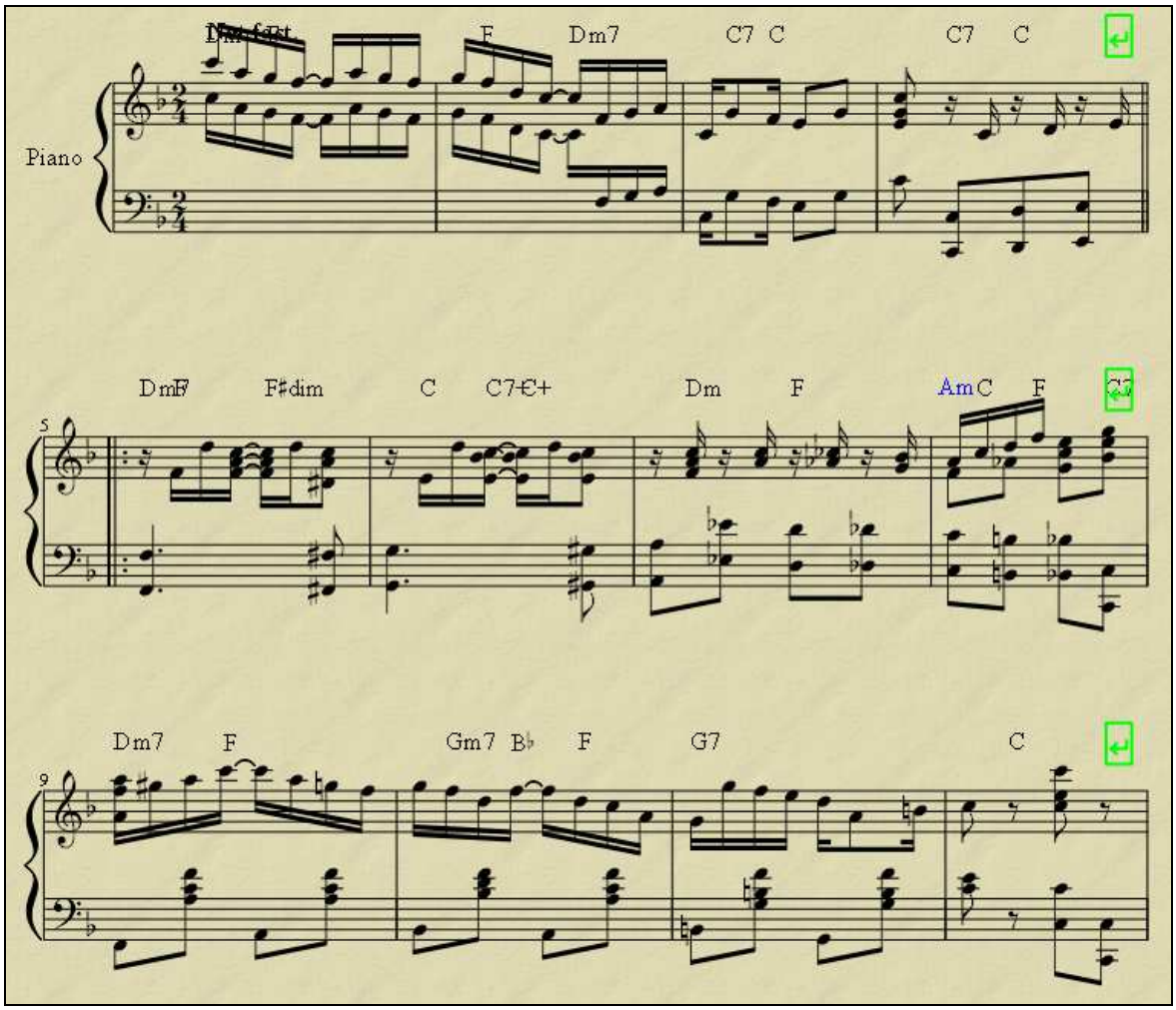


Figura 42 – Arreglos avanzados sobre una partitura compleja

5. Presupuesto

5.1. Planificación por etapas

Material y personal

Para la realización de este proyecto, al igual que ocurriría con cualquier otra empresa de magnitud semejante, ha sido necesaria una planificación previa. El objetivo de esta planificación ha sido realizar el reparto de recursos para las tareas que se han de realizar de acuerdo al diseño elegido, así como estimar el tiempo que dichas tareas necesitarán para su realización con los recursos asignados. Esto es muy importante a la hora de calcular y distribuir los recursos necesarios para su realización.

Gracias a la planificación, se ha tenido un control suficiente sobre los plazos temporales y materiales en el proyecto, lo que ha permitido su realización dentro de un margen de tiempo adecuado y sin necesidad de hacer acopio de recursos materiales o personales de última hora.

A continuación se muestra la planificación temporal que se ha seguido para la realización del motor de análisis armónico.

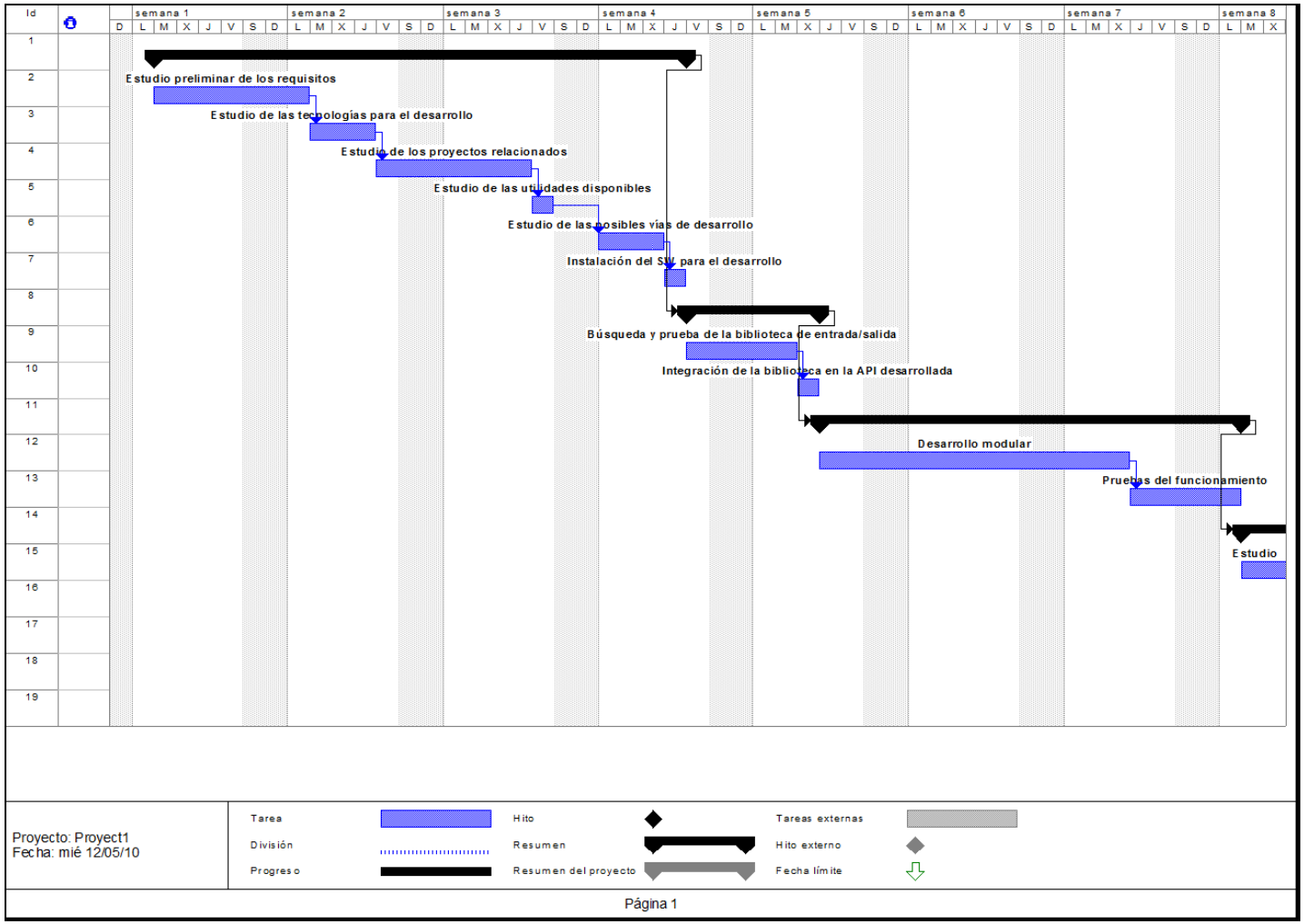
Preparación del proyecto	**18 días**	**Fase de análisis**
Estudio preliminar de los requisitos	5 días	
Estudio de las tecnologías para el desarrollo	3 días	
Estudio de los proyectos relacionados	5 días	
Estudio de las utilidades disponibles	1 día	
Estudio de las posibles vías de desarrollo	3 días	**Fase de diseño**
Instalación del SW para el desarrollo	1 día	
Integración de MusicXML	**4 días**	**Fase de desarrollo**
Búsqueda y prueba de la biblioteca de entrada/salida	3 días	
Integración de la biblioteca en la API a desarrollar	1 día	
Desarrollo del motor de análisis	**17 días**	
Desarrollo modular	2 semanas	
Pruebas del funcionamiento	3 días	
Desarrollo del módulo de arreglos	**18 días**	
Estudio del entorno DROOLS	5 días	
Integración con el programa	2 días	
Desarrollo y depuración de las reglas	11 días	
Evaluación del funcionamiento	**1 semana**	**Fase de evaluación**
Desarrollo de la memoria del proyecto y la documentación relacionada (Fase de documentación)	**4 semanas**	**Fase de documentación**

Tabla 6 – Descomposición de las tareas

Esta duración se refiere únicamente al tiempo dedicado a cada tarea, y no al tiempo total desde que una tarea comienza hasta que es finalizada, ya que este tiempo ha sido mayor puesto que la disponibilidad para la realización del proyecto no ha sido exclusiva, sino que ha debido ser compaginada con el trabajo en la empresa.

Por otro lado, debido a la escasez de recursos humanos disponibles, estas tareas han debido competir por el uso de los mismos, dando lugar a un tiempo total de desarrollo que podría haberse reducido sustancialmente en caso de haber dispuesto de una cantidad de recursos mayor. Si el número de recursos materiales y humanos hubiera sido mayor, algunas de las tareas habrían podido realizarse en paralelo, reduciendo de esta forma el tiempo de desarrollo global del proyecto. No obstante, a la vista de los resultados, dado que no fue necesario cumplir con unos requisitos temporales estrictos para la entrega del proyecto, los recursos de los que hemos dispuesto para este proyecto han demostrado ser suficientes.

La siguiente figura representa gráficamente la planificación del proyecto:


Id
semana 1
semana 2
semana 3
semana 4
semana 5
semana 6
semana 7
semana 8
D L M X J V S D
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Estudio preliminar de los requisitos
Estudio de las tecnologías para el desarrollo
Estudio de los proyectos relacionados
Estudio de las utilidades disponibles
Estudio de las posibles vías de desarrollo
Instalación del SW para el desarrollo
Búsqueda y prueba de la biblioteca de entrada/salida
Integración de la biblioteca en la API desarrollada
Desarrollo modular
Pruebas del funcionamiento
Estudio
Proyecto: Proyect1
Fecha: mié 12/05/10
Tarea
División
Progreso
Hito
Resumen
Resumen del proyecto
Tareas externas
Hito externo
Fecha límite
Página 1

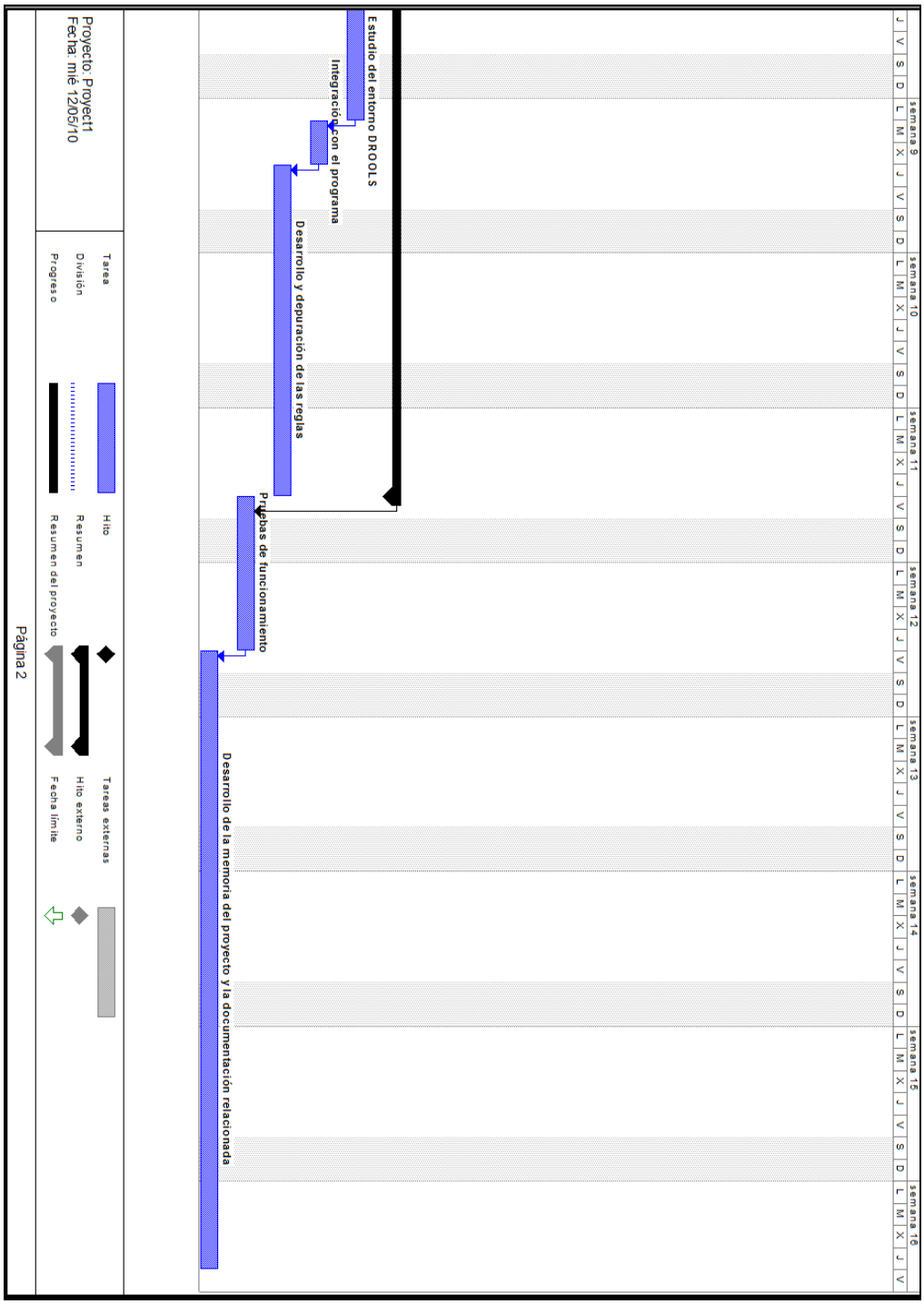


Figura 43 – Diagrama de Gantt

5.1. Presupuesto

A la hora de estudiar la viabilidad de cualquier proyecto, el presupuesto es una herramienta fundamental a tener en cuenta. Este proyecto no ha sido una excepción. Pese a que se trata de una herramienta inherentemente imprecisa, ya que para realizar presupuestos con precisión sería necesario disponer de información sobre el desarrollo y sobre posibles eventualidades que, en la inmensa mayoría de los casos, al comienzo de un proyecto sencillamente no está disponible, el presupuesto es clave para realizar una estimación bajo supuesto de que las fechas y condiciones que se estiman en la planificación son las correctas. Lógicamente, cualquier variación de los supuestos iniciales a lo largo del desarrollo hace necesaria una revisión del presupuesto.

En cuanto a los perfiles de personal necesarios para el desarrollo del proyecto, éstos han sido los siguientes:

- Técnico informático
- Analista de software
- Personal encargado de la documentación
- Programador
- Jefe de proyecto

No obstante, sólo ha habido una persona física para el desarrollo de todo el trabajo, la cual ha desempeñado los papeles que corresponden a cada uno de los perfiles anteriores, con la salvedad de que se ha considerado además necesaria la presencia de un jefe de proyecto que coordine y supervise las tareas que han de llevarse a cabo, papel que ha sido desempeñado por la tutora del proyecto, quien supervisó el desarrollo, sugirió líneas de actuación para la resolución de problemas y trabajó activamente en la revisión y corrección tanto de aspectos técnicos como de documentación.

En cuanto al ordenador personal utilizado para el desarrollo del software, se trata de un ordenador portátil marca Asus basado en un Intel Pentium Core 2 Duo, con 2 Gb de memoria y 120 Gb de disco duro. Las prestaciones de este equipo cumplieron con las expectativas para el desarrollo.

En lo que respecta al software utilizado para el desarrollo, al haber utilizado herramientas libres como Eclipse, Drools o MuseScore, no es necesario incluirlas en el presupuesto ya que éstas pueden ser utilizadas sin cargo alguno por parte del usuario. Sólo aquellas herramientas propietarias utilizadas para la documentación (Microsoft Office) y la planificación (Microsoft Project), así como el sistema operativo (Windows XP Professional SP2), deben ser tenidas en cuenta.

Vistos los recursos necesarios a incluir en el presupuesto, podemos pasar a realizar las mediciones. Con respecto a las mediciones, cabe decir que se trata de una estimación de los recursos necesarios para el desarrollo del proyecto, ya sean éstos materiales o humanos. En el caso de los recursos materiales, es necesario indicar el número de unidades necesarias de cada elemento. Por otro lado, con respecto a los recursos humanos, es necesario indicar el número de horas de trabajo que cada persona implicada en el proyecto debe dedicar. Las mediciones para el presente proyecto se muestran a continuación:

Recurso humano	Horas de trabajo	Precio (€/h)
Técnico informático	16	8,16
Analista de software	200	10,61
Programador	256	9,80
Jefe de proyecto	30	12,79
Persona encargada de tareas de documentación	224	9,80
Recurso material	**Unidades**	**Precio/unidad**
Ordenador portátil	1	800€
Microsoft Windows XP profesional SP2	1	60€
Microsoft Office 2007 (hogar y estudiantes)	1	129€
Microsoft Project	1	300€

Tabla 7 – Recursos asociados al proyecto y coste

Los costes asociados a estas horas de trabajo se han obtenido de acuerdo al convenio colectivo nacional de empresas de ingeniería y oficinas de estudios técnicos, donde se establece, en la revisión salarial del 28 de Enero de 2010, que las tareas desarrolladas por profesionales perteneciente a los grupos[26]: I (jefe de proyecto), IIA (analista de software), IIB (programador y personal encargado de la documentación) y IIIB (técnico informático), tienen los siguientes niveles retributivos:

Grupo	Nivel retributivo	Sueldo base mensual (€)
I	1	2.046,96
IIA	2	1.698,54
IIB	3	1.567,89
IIIB	5	1.306,58

Tabla 8 – Niveles retributivos de acuerdo a cada grupo profesional

En la Tabla 7, como podemos ver, se muestran además los precios unitarios, los cuales muestran el coste por hora de trabajo de cada empleado en el caso de recursos humanos. Para los recursos materiales, muestran el precio de todos los elementos utilizados en el proyecto.

Teniendo en cuenta todos los elementos vistos hasta ahora, estamos en condiciones de elaborar un presupuesto general del proyecto para realizar una valoración definitiva de su coste:

[26] Según lo establecido en el artículo 17 del BOE número 267 del jueves 5 de noviembre de 2009, sección 3, página 92526, [22]

Recurso humano	**Precio (€/h)**	**Horas de trabajo**	**Coste del recurso (€)**
Técnico informático	8,16	16	130,56
Analista de software	10,61	200	2122
Programador	9,80	256	2508,8
Jefe de proyecto	12,79	30	383,7
Persona encargada de tareas de documentación	9,80	224	2195,2
Recurso material	**Precio/unidad (€)**	**Unidades**	**Coste del recurso[27] (€)**
Ordenador portátil	800	1	40
Microsoft Windows XP profesional SP2	60	1	3
Microsoft Office 2007 (hogar y estudiantes)	129	1	6,45
Microsoft Project	300	1	15
Coste total (€)			**7404,71**

Tabla 9 – Presupuesto total

El presupuesto total de este proyecto asciende a la cantidad de 7404,71€

Leganés a 10 de Julio de 2003

El ingeniero proyectista

Fdo. Francisco Javier Rodríguez Donado

[27] Se ha aplicado la siguiente fórmula para el cálculo de la amortización de estos recursos:

$\frac{A}{B} C \cdot D = Amortización$, donde *A* es el número de meses desde la fecha de facturación en que el equipo es utilizado; B es el período de depreciación (60 meses), C es el coste del equipo (sin IVA), y D es el porcentaje de uso que se dedica al proyecto del recurso en cuestión.

6. Conclusiones

6.1. Conclusiones

A lo largo del presente Proyecto Fin de Carrera, se ha desarrollado una aplicación capaz de analizar partituras musicales basándose en teoría sobre la armonía musical, la cual obtiene la progresión armónica subyacente y realiza además un informe en formato de texto basado también en la armonía acerca de la progresión obtenida. Esta aplicación incluye además un módulo de arreglos que, haciendo uso de los resultados del motor de análisis armónico, logra generar arreglos musicales de forma automática, alterando la progresión armónica detectada y proponiendo otra que, pese a ser distinta, es equivalente a la original.

Esta aplicación ha logrado de forma satisfactoria gestionar la entrada y la salida de datos utilizando el formato abierto de partituras electrónicas más extendido que hay en la actualidad, MusicXML.

Se han realizado pruebas, tanto de características básicas de la aplicación, como pruebas más avanzadas utilizando partituras con obras musicales de gran complejidad, y los resultados obtenidos cumplen completamente con las expectativas planteadas al comienzo del desarrollo. Es destacable además la capacidad de la aplicación para obtener análisis coherentes sobre una gran cantidad de obras occidentales de estilos que varían desde la música clásica hasta el pop, el rock, música latinoamericana e incluso algunas composiciones jazz cuya base armónica no es excesivamente compleja.

Este proyecto ha sido completamente desarrollado en Java y, para ello, se ha diseñado una API que hace que sea fácilmente integrable en aplicaciones externas, tanto en lo que respecta a las llamadas necesarias para desarrollar su funcionalidad como en lo que respecta al fácil acceso a los resultados generados, mediante estructuras específicamente desarrolladas que proporcionan toda la información relevante. Por otro lado, también con el afán de facilitar la integración en proyectos futuros, tanto los comentarios en el código como la documentación de la API han sido enteramente realizados en Inglés.

Por último, en el aspecto personal, el desarrollo del Proyecto Fin de Carrera ha sido una experiencia muy útil para afianzar muchos de los conocimientos adquiridos durante la carrera así como para adquirir conocimientos nuevos, teniendo además que desempeñar tareas relacionadas con distintos roles, elaborar una importante planificación, y tener también que afrontar la responsabilidad de tomar importantes decisiones de diseño.

6.2. Trabajos futuros

En lo que respecta a trabajos futuros que partan de la aplicación desarrollada en el presente Proyecto Fin de Carrera, podríamos comenzar por ampliar las características y la funcionalidad ofrecidas por la aplicación. Dentro de este enfoque, y en lo que respecta a la profundización en el análisis teórico musical, pueden introducirse nuevas reglas sobre armonía, que permitan a la aplicación reconocer un mayor número de acordes (acordes de novena y de trecena por ejemplo) para, de esta manera, tratar de generar análisis correctos incluso en obras de gran complejidad armónica. Por otro lado,

también podría incluirse un módulo capaz de detectar el modo[28] en el que la obra está compuesta (o los modos si es que varían a lo largo de la obra).

Algo similar puede hacerse con el módulo de arreglos, ya que la teoría musical ofrece una infinidad de posibilidades en lo que respecta a la realización de arreglos sobre piezas musicales, y en esta aplicación ha sido materialmente imposible incluirlas todas.

Una forma interesante de complementar esta aplicación sería mediante un análisis basado en los aspectos rítmicos de la composición además de la armonía. Combinando ambos tipos de análisis podrían, entre otras cosas, refinarse los criterios de selección de acordes incluyendo, además de las reglas ya utilizadas, otras que por ejemplo den preferencia a acordes cuya tónica aparezca en pulsos fuertes del compás frente a las que aparezcan en pulsos débiles.

En lo que respecta a la funcionalidad, se puede completar la lista de formatos de partituras compatibles con la aplicación, añadiendo por ejemplo algunos de los que fueron comentados en el apartado 2.3.1, tales como LilyPond, o incluso considerar la posibilidad de utilizar archivos MIDI.

En un nivel más alto de abstracción, el siguiente paso podría consistir en elaborar o adquirir una colección de partituras previamente clasificadas de forma manual por un musicólogo en torno a los más diversos parámetros (desde aspectos teóricos como puede ser el modo al que pertenece la obra hasta otros aspectos tan dispares como el compositor, la época en que fue compuesta, los estados de ánimo que suscita…). Con esta información, mediante un sistema basado en minería de datos se podrían relacionar todos estos parámetros con la información sobre armonía que obtiene la aplicación aquí desarrollada. De esta manera, podría conseguirse que, frente a una partitura cualquiera, la computadora pudiera proporcionar no sólo información sobre la armonía, sino además información acerca del estilo de música en que puede clasificarse, los posibles compositores, con qué estados de ánimo se puede asociar, y un largo etcétera. Esto puede ser además de tremenda utilidad para, por ejemplo, mejorar los sistemas de recomendación musical a la carta, como la radio por internet *Pandora* [23], así como encontrar similitudes aparentemente ocultas entre obras dispares o servir de ayuda en sistemas de pedagogía musical.

Este tipo de enfoque podría además relacionarse con algunos de los trabajos previos que ya se comentaron en el apartado 2.2 para así profundizar en aspectos como la composición musical generada o asistida por ordenador o la generación de acompañamientos automáticos para músicos solistas por citar algunas posibilidades. Así, disponiendo de un sistema de análisis lo suficientemente depurado, podrían incluso dársele órdenes al programa para que, por ejemplo, generara partituras nuevas siguiendo el estilo de un determinado compositor, el de una época temporal concreta, o siendo afín a uno o más estados de ánimo.

[28] Ver apartado 2.1.7

Bibliografía

[1] Ramón López de Mantaras, *Making Music with AI: Some examples*, IIIA-Artificial Intelligence Research Institute, CSIC-Spanish Scientific Research Council, Campus UAB 08193 Bellaterra

[2] Enric Herrera, *Teoría musical y armonía moderna (Vol. 1)*. Antoni Bosch editor, 1995.

[3] Roger B. Dannenberg, *Artificial Intelligence, Machine Learning, and Music Understanding*, School of Computer Science and College of Art, Carnegie Mellon University, Pittsburgh, PA 15213 (USA)

[4] Stephen Travis Pope, *Fifteen Years of Computer-Assisted Composition*, Computer Music Journal, And Cnmat, Dept. Of Music, U. C. Berkeley. P. O. Box 9496, Berkeley, California, 94709 Usa

[5] Recordare, *MusicXML Definition* - *http://www.recordare.com/xml.html*

[6] Sibelius - *http://www.sibelius.com*

[7] Wikipedia, Encore (software) - *http://en.wikipedia.org/wiki/Encore_(software)*

[8] LilyPond, notación musical para todos - *http://lilypond.org/*

[9] LenMus project in sourceforge - *http://sourceforge.net/projects/lenmus/*

[10] Finale music composing and notation software - *http://www.finalemusic.com/*

[11] Søren Tjagvad Madsen, Gerhard Widmer, *Towards a Computational Model of Melody Identification in Polyphonic Music*, IJCAI conference on artificial intelligence, 2007

[12] Rafael Ramirez and Amaury Hazan, *A Learning Scheme for Generating Expressive Music Performances of Jazz Standards*, Music Technology Group, Pompeu Fabra University

[13] George Tzanetakis, Student Member, IEEE, and Perry Cook, Member, IEEE, *Musical Genre Classification of Audio Signals*, IEEE transactions on speech and audio processing, vol. 10, no. 5, july 2002

[14] Paolo Annesi, Roberto Basili Raffaele Gitto, Alessandro Moschitti, *Audio Feature Engineering for Automatic Music Genre Classification*, Department of Computer Science, Systems and Production, University of Roma, Tor Vergata, ITALY

[15] L. Hiller, and L. Isaacson. *Musical composition with a high-speed digital computer (1958).* Reprinted in Schwanauer, S.M., and Levitt, D.A., ed. Machine Models of Music. 9-21. Cambridge, Mass.: The MIT Press, 1993.

[16] Alberto Simões, Anália Lourenço, and José João Almeida, *Using Text Mining Techniques for Classical Music Scores Analysis*, Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal

[17] Anna Jordanous, Alan Smaill, *Artificially Intelligent Accompaniment using Hidden Markov Models to Model Musical Structure*, CIM08 Abstract Proceedings, Thessaloniki, 3-6 July 2008

[18] B. Thom. *BoB: An improvisational music companion.* PhD dissertation, School of Computer Science, Carnegie-Mellon University. 2001.

[19] JBoss Community – *Documentation*, *http://www.jboss.org/drools/documentation.html*

[20] ProxyMusic project home page – *https://proxymusic.dev.java.net/*

[21] Drools Expert User Guide - *http://downloads.jboss.com/drools/docs/5.0.1.26597.FINAL/drools-expert/html/index.html*

[22] BOE Núm. 267, Jueves, 5 de Noviembre de 2009 - *www.boe.es/boe/dias/2009/11/05/pdfs/BOE-S-2009-267.pdf*

[23] Pandora Internet radio - *http://www.pandora.com*

[24] MuseScore, programa libre de notación musical - *http://musescore.org/*

[25] Gilad Bracha, *Generics in the Java Programming Language* - *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*, July 5, 2004

[26] José Manuel Gutiérrez, *Sistemas Expertos Basados en Reglas*, Departamento de Matemática Aplicada. Universidad de Cantabria

[27] Drools downloads - *http://www.jboss.org/drools/downloads.html*

[28] A Complete MusicXML 2.0 Example - *http://www.recordare.com/xml/example20.html*

[29] M.L. Johnson, *An expert system for the articulation of Bach fugue melodies.* Readings in Computer Generated Music, ed. D.L. Baggi, 41-51. Los Alamitos, Calif.: IEEE Press. 1992.

[30] R. Bresin. *Articulation rules for automatic music performance*. Proceedings of the 2001 International Computer Music Conference 2001. San Francisco, Calif.: International Computer Music Association. 2001.

[31] A. Friberg. *A quantitative rule system for musical performance*. PhD dissertation, KTH, Stockholm. 1995.

[32] A. Friberg, R. Bresin, L. Fryden, and J. Sunberg. *Musical punctuation on the microlevel: automatic identification and performance of small melodic units*. Journal of New Music Research 27:3 (1998), 271-292.

[33] A. Friberg, J. Sunberg, and L. Fryden. *Music From Motion: Sound Level Envelopes of Tones Expressing Human Locomotion*. Journal on New Music Research, 29:3 (2000), 199-210.

[34] J. Bharucha. *MUSACT: A connectionist model of musical harmony. In Machine Models of Music*, ed. Schwanauer, S.M., and Levitt, D.A., 497-509. Cambridge, Mass.: The MIT Press. 1993.

[35] C. Fry. *Flavors Band: A language for specifying musical style (1984). Reprinted in Schwanauer*, S.M., and Levitt, D.A., ed. 1993. Machine Models of Music. 427-451. Cambridge, Mass.: The MIT Press.

Anexos

Manual de usuario

La aplicación *ScoreAnalysis* permite realizar análisis armónicos de partituras codificadas en archivos que sigan el estándar MusicXML, así como arreglos en la armonía de manera automatizada. Una vez realizada la instalación de la aplicación (y de la JVM si ésta no estuviera ya instalada en el sistema), ésta podrá ser ejecutada escribiendo el siguiente comando desde la consola del sistema:

java -jar ScoreAnalysis.jar

Esto cargará la interfaz gráfica de usuario, la cual tiene el aspecto de la siguiente imagen:

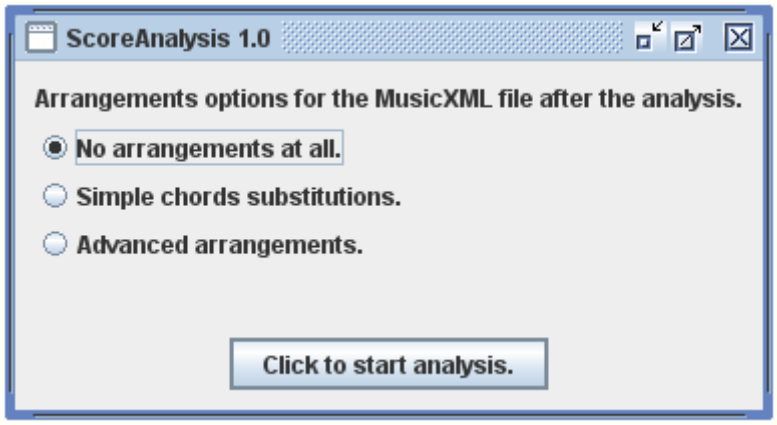


Figura 44 – Aplicación ScoreAnalysis

Para realizar un análisis de una partitura habrá que hacer click en el botón "*Click to start analysis*". Esto abrirá un rastreador de archivos que le permitirá seleccionar el archivo con la partitura que desee analizar:

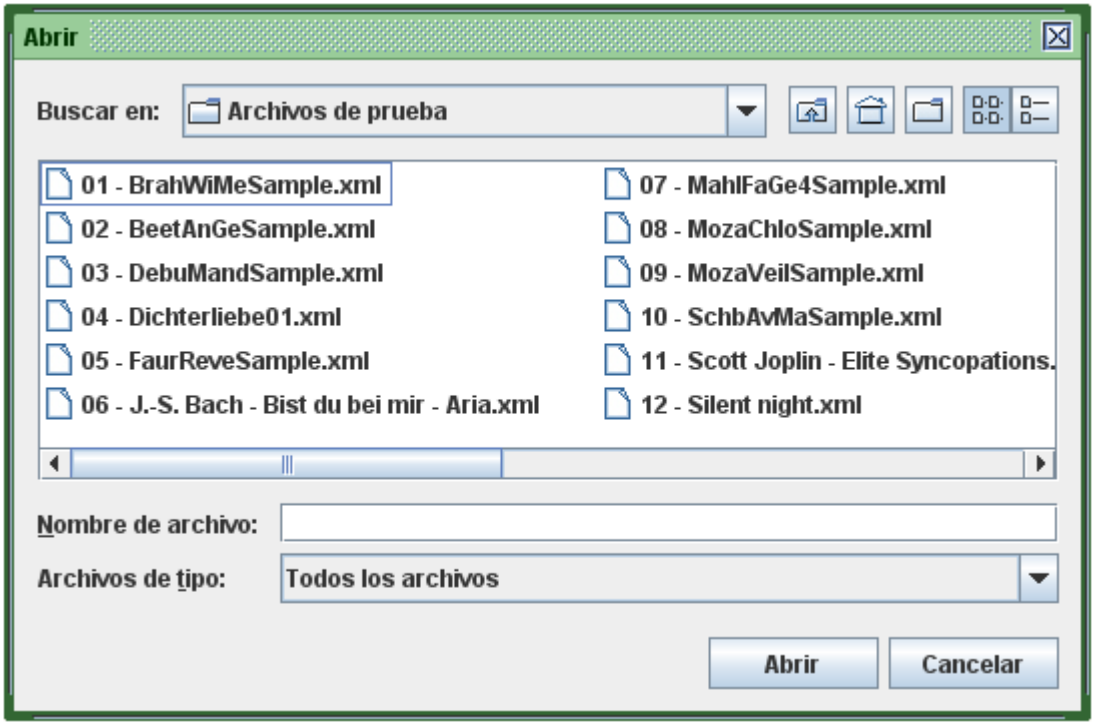


Figura 45 – Buscador de archivos de la aplicación

Tras seleccionar el archivo, la aplicación comenzará el proceso del mismo, cambiando el cursor del ratón al modo de espera mientras éste pase sobre ella, e indicando mediante el texto del botón (ahora no presionable) que se está realizando el análisis:

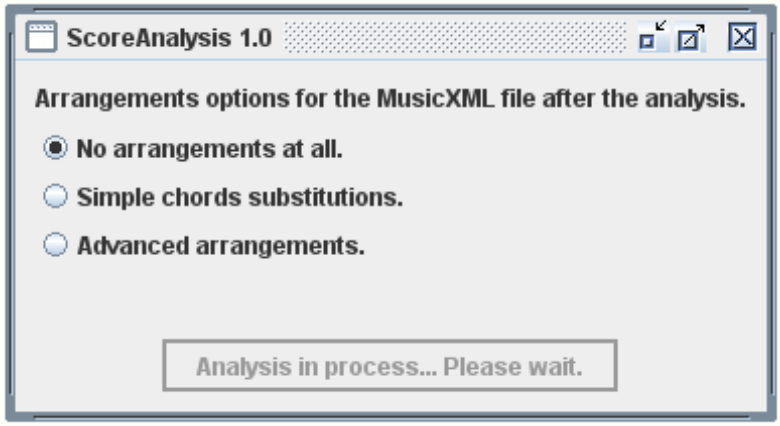


Figura 46 – Ventana de la aplicación: análisis en proceso

Durante el tiempo en el que la aplicación está realizando el análisis, se muestra información acerca del mismo en la consola desde la que ésta se lanzara.

Cuando el análisis ha finalizado, el botón volverá a ser accesible para el usuario e indicará que el análisis ha terminado y que es posible hacer click de nuevo para realizar otro análisis:

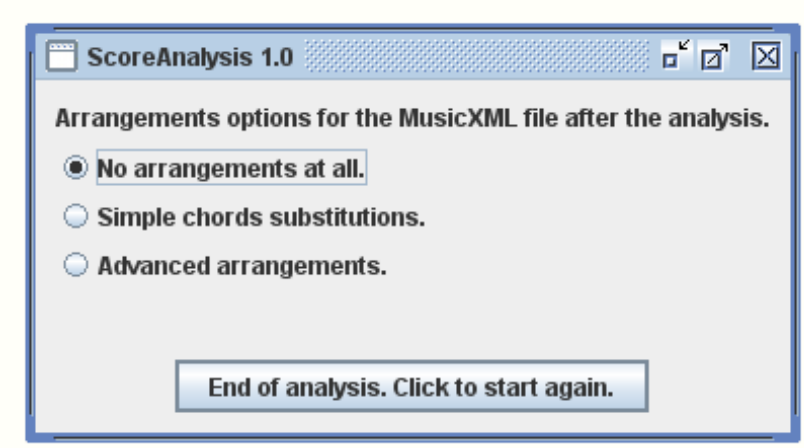


Figura 47 – Ventana de la aplicación: fin del análisis

Posibles errores:

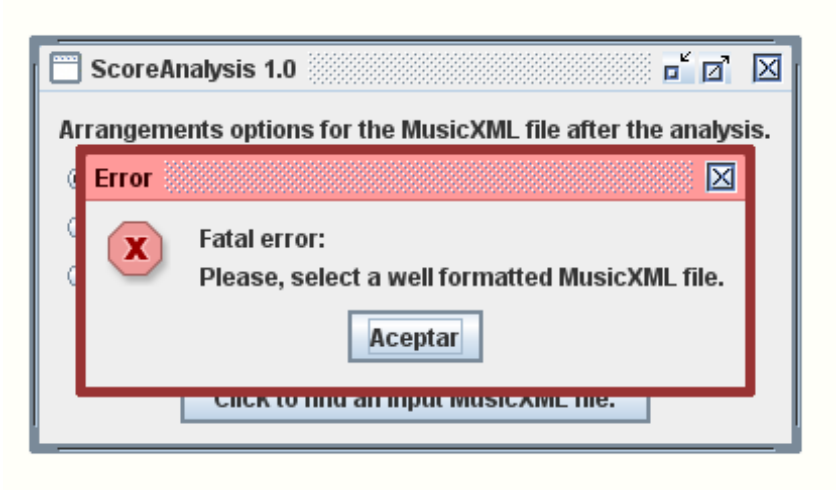


Figura 48 – Error en la aplicación

Es posible que surja algún error durante el análisis. En ese caso asegúrese de que se cumplen los siguientes requisitos:

- Se dispone de una conexión a Internet para que la aplicación pueda descargar la definición de tipo de documento (DTD) en la que se basa el archivo.
- El archivo que se desea analizar posee un formato correcto.
- El usuario que ejecuta la aplicación tiene permisos de acceso sobre el archivo que se desea analizar.
- El usuario tiene permisos de escritura sobre el directorio Output, y éste existe.
- La JVM está instalada y soporta Java 5 o superior.
- Se dispone de todos los archivos de reglas en su correspondiente directorio, así como de todos los archivos .jar usados por la aplicación.

Manual de instalación

Esta aplicación ha sido desarrollada en lenguaje Java, por lo que para poder ejecutarla es necesario disponer de la máquina virtual de Java (JVM) instalada en el equipo. La JVM puede ser descargada de la siguiente dirección web:

http://www.java.com/es/download/manual.jsp

Al acceder a la página, aparece la siguiente pantalla:

Figura 49 – Página de descarga de la JVM

Tras seleccionar el sistema operativo en el que se desee instalar, se descargará un archivo cuya ejecución iniciará el programa de instalación.

Una vez instalada la JVM, estaremos en disposición de ejecutar la aplicación de análisis armónico. Ésta viene comprimida en un archivo zip. Tras descomprimir los archivos, tendremos los siguientes archivos y directorios en la ruta donde se haya descomprimido el archivo:

\ScoreAnalysis.jar
\run.bat
\rules\AddChords.drl
\rules\ChangeChords.drl
\ScoreAnalysis_lib\...
\Output\

Tras esto, la aplicación estará correctamente instalada en el sistema. Para arrancarla habrá que ejecutar la siguiente línea en la consola de comandos desde la ruta donde se descomprimiera el archivo zip:

java -jar ScoreAnalysis.jar