UNIVERSIDADE DA CORUÑA

Facultad de Informática
Departamento de Computación

Trabajo de fin de Grado
de Ingeniería Informática

# *Herramienta para armonización musical mediante Answer Set Programming*

**Autor/a:** Martín Prieto, Rodrigo
**Director/a:** Cabalar Fernández, Pedro

*A Coruña, a 15 de enero de 2016.*

# Información general

| | |
|---|---|
| ***Título del proyecto:*** | **"Herramienta para armonización musical mediante Answer Set Programming"** |
| *Clase de proyecto:* | Proyecto Clásico de Ingeniería |
| *Nombre del alumno:* | Martín Prieto, Rodrigo |
| *Nombre del director:* | Cabalar Fernández, Pedro |
| *Miembros del tribunal:* | |
| *Miembros suplentes:* | |
| *Fecha de lectura:* | |
| *Calificación:* | |

D. Pedro Cabalar Fernández

CERTIFICA

Que la memoria titulada **"Herramienta para armonización musical mediante Answer Set Programming"** ha sido realizada por Rodrigo Martín Prieto con D.N.I. 54126010-H bajo la dirección de D. Pedro Cabalar Fernández. La presente constituye la documentación que, con mi autorización, entrega el mencionado alumno para optar a la titulación de Ingeniería en Informática.

*A Coruña, a 15 de enero de 2016.*

*Sigue rascando, hay millones de premios.*

# Agradecimientos

A mi tutor, Pedro Cabalar, no sólo por todo su esfuerzo y dedicación dirigiendo este proyecto sino por proponer la práctica que lo originó hace un par de años.
A la Facultad de Informática de A Coruña, por haber sido mi segundo hogar, por la gente que he conocido aquí en estos años y por haber hecho de mi un orgulloso Ingeniero.
Al grupo Potassco por crear clasp y gringo, herramientas sin las cuales no sería posible y que han cambiado mi modo de entender la programación.
A Vicente Moret por enseñarme la humanidad tras las máquinas.
A Gastón Rodríguez por poner una pizca de tango y de jazz en mi vida.
A mi familia por haber financiado mis ambiciones.
A mi familia en Internet, demasiado grande y demasiado dispersa como para mentarlos a todos sin dejarme a nadie, gracias por los ánimos, por aguantar las ristras infinitas de *tweets*, gracias por tener un puerto abierto para mi.
A Raquel por aguantar mi ausencia durante estos meses tan difíciles.
A Diego "Gellido" Fajardo, por las croquetas, las *FajiFajitas* y las charlas, gracias por haber hecho soportable el viaje, m8.
A Ismael "Filo" Barbeito y Daniel Ruiz por haber llegado los tres hasta aquí incluso cuando parecía imposible. GL HF.
Al Wok TWO, por haber sido mi comedor fuera de casa y el primer lugar de Coruña donde he podido pedir "lo de siempre". En serio, esa cosa está riquísima.
Y sobre todo a lo más importante y siempre presente en mi vida, la música, que sin ella este proyecto no existiría de ningún modo, no sólo por radicar en sus tripas, sino por haber sido mi combustible durante más de cinco años. Wes Montgomery, Diana Krall, In Flames, Ken Ashcorp, gracias de corazón.

**Resumen**

El objetivo de este proyecto...

**Palabras clave:**

- √ Programación Declarativa
- √ Programación Lógica
- √ Procesamiento de Lenguajes
- √ Representación del Conocimiento
- √ Answer Set Programming
- √ Música
- √ Música por computador
- √ Armonía
- √ clingo
- √ MusicXML

# Índice general

**Página**

# Índice de figuras

**Figura** **Página**

# Capítulo 1
# Introducción

## 1.1. Motivación

El aprendizaje de la teoría musical lleva estancado en los mismos métodos y sistemas desde hace años. Son métodos funcionales basados en ejercicios y repetición, para acostumbrar el oído y lograr soltura a la hora de componer o resolver los problemas propuestos en clase. El aprendizaje de armonía es uno de los pasos más importantes en música, ya que saber analizar de este modo una partitura es crucial para comprenderla, modificarla o componer nuevas piezas. El presente proyecto busca ayudar tanto en el aprendizaje como en la composición desde un punto de vista puramente armónico.

La Inteligencia Artificial estudia como crear sistemas que se comporten de manera inteligente, es decir, que sean capaces de razonar, deducir y resolver problemas del mismo modo que pudiera hacerlo una persona. Se busca que estos sistemas sean autónomos, se busca que estos sistemas sepan justificar sus resultados, se busca que estos sistemas sean capaces de aprender para poder desempeñar su tarea con mejores resultados o en menos tiempo. Pero la inteligencia conlleva más cosas, tales como la creatividad. La creatividad es el impulso por el cual alguien (o algo) decide crear una obra de la nada. No solo una obra artística, también obras funcionales como lo fueron los grandes inventos del pasado.

Existe, no obstante, controversia con respecto al campo, como suele suceder siempre que un ordenador empieza a conseguir hacer lo que antes solo podían hacer las personas. En el caso del sistema Emily Howell, por ejemplo, ha habido numerosos directores de orquestas que se han negado a interpretar sus composiciones al no provenir de un compositor humano. Existe, a ojos de los más

conservadores respecto al tema, el miedo a que el esfuerzo de la composición musical pierda su significado. Si no podemos distinguir además qué piezas han sido compuestas por máquinas y cuales no, el problema se acentúa.

En este caso, la motivación del trabajo es una mezcla entre creatividad y la necesidad de solucionar un problema. Se habla de creatividad porque el proyecto está aplicado a un campo inherentemente creativo, como es la música. Pero al mismo tiempo, pretende ser una herramienta que ayude a estudiantes de música a progresar en su trabajo. Este sistema inteligente, será capaz de razonar, deducir, y último lugar crear, la armonía de piezas musicales sencillas. Si bien esto no es un trabajo completo de composición, sí que debería ayudar a corregir partituras allí donde el sistema detecte incoherencias con la armonía creada o ya presente en la pieza.

El interés por la aplicación del *Answer Set Programming* en la música derivó de un trabajo previo con el Profesor Cabalar durante el curso de la asignatura de Representación del Conocimiento y Razonamiento Automático donde los alumnos construimos un sistema que dada una melodía, producía un canon polifónico. El presente trabajo se planteó como una extensión generalista del problema, orientado a composición completa, aunque fue reducido para poder entrar en los límites de un trabajo de fin de grado debido a la gran complejidad de crear un sistema así. No obstante, la lógica proposicional es idónea para esta tarea ya que el conjunto de reglas de la armonía clásica usada en los niveles más elementales del Conservatorio no ha cambiado desde los orígenes de la materia. Es un conjunto de reglas conciso, no muy grande y más o menos estricto. Simplemente traduciendo este conjunto de reglas a restricciones del lenguaje de lógica proposicional y siendo capaces de extraer los hechos lógicos de una partitura, el sistema debería ser capaz de detectar los errores de la misma y solucionarlos, así como rellenar huecos dejados a propósito en la partitura o completar otras líneas melódicas para formar, por fin, la armonía de la canción.

## 1.2. El proyecto

El proyecto consiste en la construcción de una herramienta software capaz de tomar como entrada una partitura polifónica (es decir, de múltiples voces) parcial, deducir los acordes correspondientes a la unidad de tiempo deseada, y si se

desea completarla respetando las reglas básicas de armonía, de modo similar a los ejercicios habituales en esta disciplina musical. Se intentará abarcar la estructura musical de forma horizontal (compases) y vertical (múltiples voces). Este problema de partida se puede especificar en términos de resolución de restricciones, un campo para el que existen distintos formalismos y herramientas disponibles. En concreto, el proyecto usará el paradigma de *Answer Set Programming* una variante de Programación Lógica de uso frecuente para la Representación del Conocimiento y la resolución de problemas. La principal ventaja de ASP para este caso es la facilidad que otorga el uso de predicados simples a la hora de definir ciertos sucesos dentro de la partitura así como añadir directamente las reglas usadas en armonía bajo la forma de reglas de programación lógica. Esto proporciona una enorme flexibilidad, ya que ASP es un paradigma totalmente declarativo, en el que sólo se realiza la especificación del problema, y no se describe el método de resolución que se aplica para el mismo. Otra ventaja importante de ASP para este escenario es la posibilidad de usar preferencias, ya que algunas reglas armónicas no son estrictas, sino que se busca que se respeten en la medida de lo posible. Existen además antecedentes de uso de ASP para composición musical. En concreto, el sistema ANTON permite también la composición polifónica (mayormente, dos voces) siguiendo el estilo musical de reglas "Palestrina"de la música renacentista. Esta herramienta es más elaborada que la presente propuesta, ya que realiza la composición completa de una pieza, incluyendo figuras rítmicas complejas. La principal diferencia es que el presente anteproyecto está orientado principalmente a armonización y, aunque es capaz de completar partituras, no busca obtener un buen resultado a nivel melódico.

El principal objetivo del proyecto es crear un conjunto de módulos que, funcionando como uno solo, sean capaces de ofrecer resultados a ejercicios sencillos de armonización aceptables desde el punto de vista de un experto, de modo que el software final pueda ser utilizado en el mundo real para ayudar a la composición y al aprendizaje de armonía.

Los módulos planteados, de forma general, para el proyecto son:

- **Entrada y preprocesado:** Haciendo uso de un editor musical capaz de exportar al formato de entrada se producirá un fichero que este modulo convertirá a hechos lógicos.

- **Armonización:** Escrito en ASP, mediante el uso de software que calcule las restricciones para el fichero de entrada, este modulo producirá soluciones al problema de armonización y completará la partitura en caso de ser necesario.
- **Salida y postprocesado:** Tomando como entrada las soluciones en forma de hechos lógicos, producirá un fichero en el formato de salida especificado para su posterior visualización.

## 1.3. Estructura

El resto de capítulos de la memoria del presente proyecto seguirán una estructura tradicional.

- **En el Capítulo 2** se enmarca el proyecto en el contexto tecnológico actual y se explicarán algunos términos que puedan no resultar familiares al estudiante medio del Grado en Ingeniería Informática. Se exponen además las nociones básicas musicales necesarias para una mejor comprensión del proyecto.
- **En el Capítulo 3** se trata el grueso del proyecto, es decir, la planificación y desarrollo del mismo, empezando por la definición del tipo de ciclo empleado así como una estimación en horas y presupuesto para el proyecto en general. A continuación se detalla el trabajo realizado en cada una de las iteraciones, comparando con la planificación inicial y comentando los objetivos logrados al final de cada una de ellas.
- **En el Capítulo 4** se realiza una evaluación objetiva del proyecto, tanto cualitativa como cuantitativa. Varios expertos proponen diferentes piezas musicales y juzgan los resultados ofrecidos por la herramienta.
- **En el Capítulo 5** se detallan las conclusiones extraídas del proyecto, su viabilidad, impresiones y se realiza una evaluación general subjetiva del mismo.

Por último, tras este capítulo final se incluye una referencia detallada de la Bibliografía empleada durante el desarrollo y diversos Anexos con diagramas que complementan a los incluidos en los capítulos de desarrollo y evaluación.

# Capítulo 2
# Contextualización

## 2.1. Contexto Tecnológico

La mayor parte del trabajo en la creación y composición musical en ordenadores se ha extraído de la relación entre la teoría musical y las matemáticas. No es difícil concluir que la música y sus reglas son fácilmente modelables de forma matemática.

Dentro de la música en computación existen varias ramas diferenciadas, aunque en el contexto de un mismo trabajo pueden verse mezcladas más de una. Hablamos de composición asistida y de de sistemas inteligentes orientados a composición y, aunque se tratarán con más detalle en los siguientes puntos, todos ellos, así como el software desarrollado en dichos campos, están destinados a la creación y composición musical. Se detallan, además, algunos de los formatos más comunes de representación musical.

Si bien el sistema planteado en el proyecto no es un compositor, se enmarca dentro de este mismo contexto, y por tanto es necesario desglosarlo para entender en qué punto se encuentra la tecnología desarrollada en el momento de la publicación de este documento.

### 2.1.1. Answer Set Programming

El módulo principal del proyecto se ha desarrollado usando técnicas conocidas como Answer Set Programming (ASP de ahora en adelante) basadas en modelos estables y lógica no-monótona. La gran ventaja de ASP reside en la facilidad para

separar las reglas de inferencia de los hechos lógicos. El lenguaje de entrada de los programas usados es una variante de PROlog pero con un preprocesado que permite crear reglas generales mediante el uso de variables. La metodología de ASP se conoce como generate and test, primero se definen reglas generadoras, que con ayuda de los hechos generan los predicados derivados de los predicados simples aprotados en la entrada y en la segunda etapa se comparan los predicados presentes, tanto simples como derivados con una serie de restricciones lógicas, restricciones cardinales y predicados de optimización para eliminar del conjunto de soluciones posibles aquellas que no las cumplan o no resulten óptimas. Los modelos estables en los que se basa ASP permiten que estos pasos sean extremadamente ágiles incluso en presencia de grandes volúmenes de datos.

Las herramientas desarrolladas por el grupo Potassco incluyen, entre otras, un *grounder* (Gringo) y un *solver* (Clasp).

#### 2.1.1.1. gringo

Gringo es el *grounder* de la suite. Se encarga de transformar reglas generales a reglas concretas y transforma el problema planteado a un lenguaje entendible por el *solver*.

#### 2.1.1.2. clasp

Clasp es el *solver* de la suite. Se encarga de decidir el conjunto final de soluciones válidas dado el problema procesado e interpretado por el *grounder*.

#### 2.1.1.3. clingo

Clingo es una herramienta combinada formada por clasp y gringo. Realiza el procesado completo de un problema planteado en ASP y permite indicar algunas preferencias adicionales.

### 2.1.2. Formatos

Debido al contexto en el que se enmarca este proyecto no se cree necesario considerar formatos de salida finales, como OGG, WAV o MP3 ya que como su nombre indica, son formatos utilizados solo para reproducción que no permiten extraer ni editar información musical de forma precisa. Así mismo tampoco se

contemplan formatos de representación de partituras en forma de imágenes como SVG, PNG o PDF por motivos similares.

#### 2.1.2.1. MusicXML

MusicXML, MXML o *Music Extensible Markup Language* es una extensión del formato XML usado en la representación de música occidental. No solo contiene información musical sino que también incluye información de su representación en papel, tal como márgenes, tamaños de fuente, posición de las notas en coordenadas, etc. Hace uso del sistema de tags anidados de XML para agrupar los diferentes bloques de información de una pieza, como las voces, los compases o la información individual de cada nota. Es un formato muy rico aunque difícil de escribir correctamente a mano, es por esto que normalmente se usa solo como formato de intercambio entre software que lo aceptan como entrada y salida.

#### 2.1.2.2. LilyPond

LilyPond es un conjunto formado por el software y el formato de fichero homónimos. LilyPond como formato usa su propio lenguaje de marcado, los tags de LilyPond se parecen más a los usados en LaTeX. De forma similar a MusicXML, incluye información de representación final, aunque en mucha menor cantidad (Sólo tamaño de papel, márgenes o sangrados). La información musical de la canción está anidada por secciones de forma similar a MusicXML, aunque ésta se organiza de forma mucho más intuitiva para el lector humano del fichero. Es un formato ligero pensado para poder ser editado a mano, aunque la mayoría del software musical actual lo soporta como entrada y salida.

#### 2.1.2.3. MIDI

MIDI son las siglas de *Musical Instrument Digital Interface*. Es un standard técnico compuesto de un protocolo, una interfaz digital y conectores que permiten a una gran variedad de instrumentos electrónicos, ordenadores y otros dispositivos conectarse y comunicarse entre sí, principalmente con fines musicales, pero no siempre.

MIDI transmite mensajes de eventos que especifican notación, tono y velocidad, aunque también incluye información de modificaciones sobre estos sonidos

como volumen, *vibrato* y marcas de tiempo con fines de sincronización entre dispositivos.

Estos mensajes pueden ser codificados en ficheros para reproducción, edición o simplemente como formato de representación musical pudiendo ser editado posteriormente. Dado que no contiene información final de audio, MIDI supone una gran ventaja en cuanto a espacio en disco, aunque el sonido final depende del equipo que reproduzca el fichero.

### 2.1.3. Software

#### 2.1.3.1. Herramientas

**2.1.3.1.1. Flex y Bison** Flex y Bison son utilidades Unix que permiten escribir *parsers* veloces para casi cualquier formato de archivo. Implementan procesado *Look-Ahead-Left-Right* de gramáticas libres de contexto no ambiguas.

**2.1.3.1.2. Music21** Music21 es un conjunto de herramientas que sirve de ayuda a estudiantes y músicos a responder preguntas sobre música rápida y eficazmente. No sólo posee una base de datos bastante completa para realizar análisis musicológicos sino que contiene herramientas para la composición programática de música.

#### 2.1.3.2. Composición Asistida

Dentro de la composición asistida encontramos principalmente software de composición general en forma de editores de partituras que incorporan herramientas para ayudar al compositor en el proceso. Estas herramientas pueden ser corrección de la métrica de los compases, transposición de secciones de la canción, cambios de tonalidad, construcción de acordes dada una nota generadora, etc.

**2.1.3.2.1. MuseScore** MuseScore es un editor *WYSIWYG* capaz de exportar a varios formatos de representación musical digital, tales como MIDI, LilyPond o MusicXML.

**2.1.3.2.2. Sibelius** Sibelius permite trabajar con gran variedad de modos de entrada de notas para sus partituras, desde los formatos convencionales hasta a

través de instrumentos con salida MIDI o mediante el escaneado de partituras en papel haciendo uso de OCR.

**2.1.3.2.3. Finale** Finale destaca por la cantidad de ajustes que permite realizar sobre el pentagrama a un nivel de detalle muy fino, aunque presenta una curva de aprendizaje muy elevada. Sus principales características tienen que ver con la visualización del pentagrama, ya que posee *plug-ins* que se encargan de que el espacio entre las notas sea el correcto o que no haya colisiones entre notas de diferentes voces entre otros.

### 2.1.3.3. Sistemas Inteligentes

La música si bien puede modelarse de forma matemática, con reglas estrictas que derivan en algoritmos de composición, requiere creatividad. Un algoritmo determinista no puede ser creativo, ya que para la misma entrada, siempre producirá la misma salida. Si bien existe la composición algorítmica como aproximación a la música compuesta por ordenadores, no es relevante para este trabajo.

Dentro de la inteligencia artificial, se han realizado aproximaciones a la composición musical desde gran parte de las ramas principales del campo

**2.1.3.3.1. EMI y Emily Howell** Desarrollado por David Cope, EMI o Experiments in Music Composition es un sistema capaz de identificar el estilo presente en una partitura incompleta y completar la cantidad de notas restantes que el compositor requiera. El trabajo de Cope estudia la posibilidad de emplear gramáticas y diccionarios en la composición musical. EMI derivó en el software Emily Howell. Emily Howell utiliza EMI para crear y actualizar su base de datos, pero cuenta con una interfaz a través de la cual se puede modificar, a través de *feedback*, la composición. Cope enriqueció y pulió Emily Howell con su propio estilo musical para crear varios discos que después fueron publicados.

**2.1.3.3.2. ANTON** ANTON es un sistema de composición rítmica, melódica y armónica basado en Answer Set Programming. ANTON compone breves piezas musicales desde cero o partiendo de partituras incompletas utilizando un estilo basado en el del compositor renacentista Giovanni Pierluigi da Palestrina.

Recibe como entrada ficheros con las notas codificadas como hechos lógicos para después rellenar las secciones incompletas o añadir nuevas notas hasta que la pieza está completa. ANTON crea y completa dichas piezas teniendo en cuenta el número de tiempos rítmicos de las mismas y seleccionando la nota correspondiente de acuerdo a la nota o estado anterior

**2.1.3.3.3. Vox Populi** Vox Populi utiliza algoritmos evolutivos para componer música en tiempo real. En este sistema, se parte de una población de acordes codificados mediante el protocolo MIDI para después mutarlos y seleccionar los mejores acordes a criterios puramente físicos relevantes para la música. Su interfaz gráfica permite al usuario controlar la función de *fitness* del proceso evolutivo así como los atributos del sonido producido.

**2.1.3.3.4. CHORAL** CHORAL es un sistema experto que funciona como armonizador en el estilo clásico de Johann Sebastian Bach. Las reglas que utiliza el sistema representan conocimiento musical desde varios puntos de vista de la coral. El programa armoniza melodías corales mediante un sistema de generación y prueba con *backtracking*. La base de conocimiento de CHORAL permite realizar modulaciones propias del estilo, crear patrones rítmicos e impone restricciones complejas para mantener el interés melódico en las voces intermedias.

**2.1.3.3.5. CHASP** CHASP es una herramienta creada por el grupo Potassco para calcular progresiones de acordes mediante Answer Set Programming partiendo de cero, pudiendo especificar clave y duración. A diferencia del presente proyecto no toma un fichero de entrada para armonizar piezas, pero sí que es capaz de dotar a la salida del programa de diferentes estilos rítimicos.

## 2.2. Contextualización musical

El presente proyecto, pese a estar realizado para Ingeniería Informática, posee una importante carga de teoría musical. Dado que los lectores del proyecto podrían no conocer en profundidad los conceptos musicales empleados en el documento, se cree importante hacer una introducción a modo de glosario y punto

de consulta, partiendo de los elementos más básicos de la teoría musical y llegar hasta aquellas reglas más complejas utilizadas para el desarrollo del proyecto.

### 2.2.1. Ritmo y Figuras

La parte más elemental de la música es la estructura rítmica de la pieza, es decir, los patrones de golpes sonoros que están presentes en la canción y que normalmente se repiten a lo largo de toda ella. Para componer estas secuencias rítmicas se utilizan una serie de figuras que representan sonidos de una duración relativa al *tempo* de la pieza, en combinación con otra serie de figuras que representan la ausencia de sonido durante esas mismas duraciones. Las primeras son denominadas figuras y las segundas, silencios.

Tomando una figura de referencia, en este caso la redonda, y mediante un proceso de escisión binaria se obtienen el resto de figuras presentes en las partituras. Una redonda equivale a dos blancas, una blanca equivale a dos negras, una negra equivale a dos corcheas, una corchea equivale a dos semicorcheas... etc. A modo de detalle, para facilitar la lectura de figuras fraccionarias pequeñas, estas pueden ser unidas mediante una barra horizontal si aparecen de forma consecutiva. Es importante tener en cuenta que aunque la duración total de dos figuras iguales sea equivalente a otra figura de mayor duración, rítmicamente sí existe diferencia, no se debe olvidar en ningún momento que cada figura es un golpe de sonido con una duración determinada y por tanto, pese a que en cuanto a duración una redonda equivale a cuatro negras, a la hora de interpretar la pieza, una redonda será un único sonido largo mientras que cuatro negras serán cuatro sonidos más cortos. Esto en combinación con los silencios, componen los patrones de ritmo de los que se hablaba anteriormente.

Dado que en inglés estas figuras se llaman mediante el nombre de la fracción de redonda que representan también pueden ser representadas mediante el número del denominador de dicha fracción. De este modo la redonda sería 1, la blanca 2, la negra 4, la corchea 8, la semicorchea 16, etc. A mayores existen modificadores de la duración de las figuras como el puntillo, un pequeño punto situado a la derecha de las figuras que alarga la duración de las mismas durante la mitad de la duración de la figura a la que acompaña.

El *tempo*, anteriormente mencionado, es una relación que indica la cantidad aproximada de figuras de negra que suenan en un minuto, debido a eso la mag-

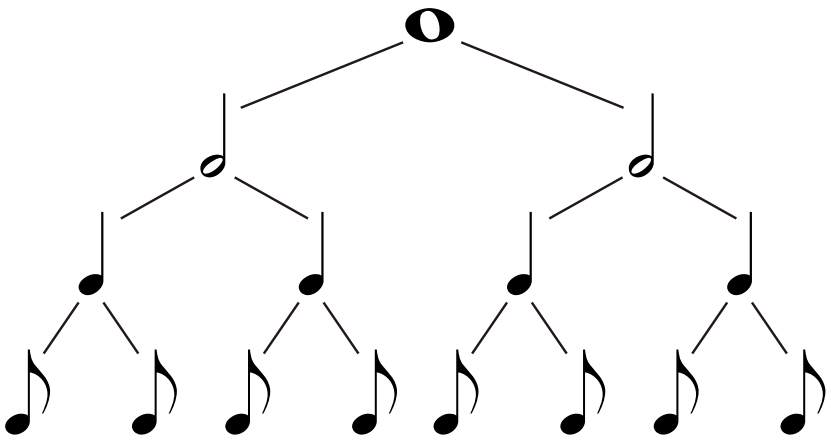

Figura 2.1: Diagrama de subdivisión de figuras

nitud que representa el *tempo* se denomina BPM o *Blacks per Minute*.

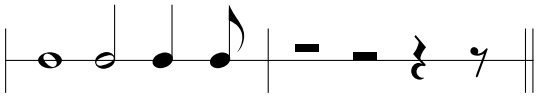

Figura 2.2: Figuras con sus respectivos silencios, de izquierda a derecha: Redonda, blanca, negra, corchea

La partitura está dividida en compases, y al principio de la pieza se indica el tipo de compás de la misma. Se representa con dos números colocados en vertical que indican respectivamente la cantidad de figuras que caben en cada compás y el tipo de dicha figura, utilizando el sistema del numerador de la fracción de redonda detallado anteriormente. A modo de aclaración, el tipo de figura utilizado en el compás solamente indica la figura base del mismo, pero eso no impide que se utilicen fracciones o figuras de mayor longitud equivalentes siempre y cuando se respete que la suma de sus duración no supere la especificada en el tipo de compás.

Según la cantidad de figuras que caben en cada compás, estos reciben diferentes nombres. En la música moderna lo más habitual es encontrar compases cuaternarios, aunque en algunas corrientes de la música clásica como el *Waltz* predominan los ternarios.

El acento en la música es un mero hincapié interpretativo en los tiempos

Figura 2.3: Ejemplos de tipos de compases: Cuaternario y Ternario

fuertes de la pieza. Aunque puede existir una acentuación arbitraria a lo largo de la partitura, los compases determinan además los tiempos fuertes y normalmente acentuados. Dependiendo del tipo de compás estos varían, aunque la primera figura de cada compás suele estar acentuada y ser considerada fuerte. No obstante, la diferenciación de tiempos débiles y fuertes es importante en la melodía y por tanto en la armonía, por ello se revisará este concepto en los puntos siguientes.

### 2.2.2. Melodía y pentagrama

La melodía de la pieza es la secuencia de notas que siguen las diferentes voces a lo largo de la partitura. Para ello se utilizan siete notas fundamentales diferentes: Do, Re, Mi, Fa, Sol, La y Si (C, D, E, F, G, A, B en la notación internacional). Entre cada uno de estas notas fundamentales se establece una distancia de un tono, excepto entre Si y Do; y entre Mi y Fa, entre las cuales sólo hay medio tono (también denominado semitono). Las fundamentales pueden ser alteradas mediante modificaciones conocidas como Sostenido y Bemol, que suben o bajan, respectivamente, un semitono a dichas notas, por eso podemos entender que contamos, en total, con 12 sonidos distintos.

Estas siete notas fundamentales (o doce sonidos), tienen asociadas a su vez una octava. Cada octava es un conjunto formado por esos mismos sonidos pero en un registro más agudo o más grave. Si bien esto produce un sonido diferente, el mismo sonido de diferentes octavas es, de mdo general, equivalente en términos de armonía.

La representación de estos sonidos se hace sobre un pentagrama, bandas formadas por cinco líneas con sus respectivos cuatro huecos donde pueden colocarse las notas de la partitura, representando cada línea y hueco consecutivos una nota fundamental diferente. Ya que esto solo permitiría representar nueve notas, se pueden utilizar líneas y huecos adicionales situados encima y debajo del pentagrama para indicar notas más graves o más agudas. En conjunción con las alte-

raciones ya mencionadas (Sostenido y Bemol) en un mismo pentagrama pueden representarse más de una veintena de sonidos diferentes.

Para poder interpretar el sonido correspondiente a cada línea o hueco existe la clave. La clave es un símbolo situado al principio de cada pentagrama e indica la nota correspondiente a una de las cinco líneas. El resto se calculan subiendo o bajando huecos y renglones a partir de esa línea.

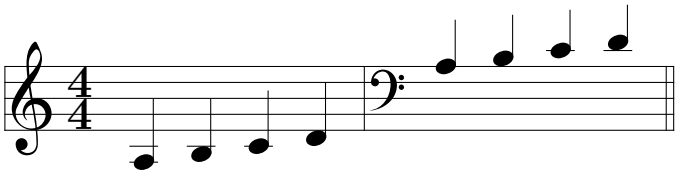

Figura 2.4: La misma secuencia de notas en clave de Sol y de Fa

En la Figura 2.4, la clave de Sol indica que en la segunda línea, contando desde abajo, estaría ubicada la nota sol, mientras que la clave de Fa indica que en la cuarta línea, contando también desde abajo, se ubicaría el sonido Fa. La finalidad de las diferentes claves es ofrecer diferentes puntos de referencia, ya que no solo indican nota sino también octava, evitando así sobrecargar la partitura con notas fuera de las cinco líneas principales. La clave de Sol utiliza sonidos más agudos, mientras que la de Fa utiliza sonidos más graves. Es habitual encontrar, por ejemplo, partituras para piano donde la mano izquierda estará escrita en clave de Fa al ser notas más graves y en clave de Sol la de la mano derecha, al ser notas más agudas.

La transposición es un proceso mediante el cual una nota es desplazada una cantidad de semitonos hacia arriba o hacia abajo. Aplicado a una sección de una pieza musical, todas las notas de dicha sección subirían o bajarían la misma cantidad de semitonos especificada.

Un salto melódico es la diferencia de frecuencia entre dos sonidos consecutivos de una misma melodía. De este modo, si el siguiente sonido es más agudo se producirá un salto ascendente, y si es más grave, descendente. Las secuencias ascendentes y descendentes se combinan con el ritmo de la canción para crear tramos de tensión melódica, de resolución, de reposo... Para entender estos conceptos es necesario introducir el concepto de escala.

Una escala es un subconjunto de los doce sonidos disponibles donde sus miembros cumplen una distribución concreta de separaciones tonales. La escala más

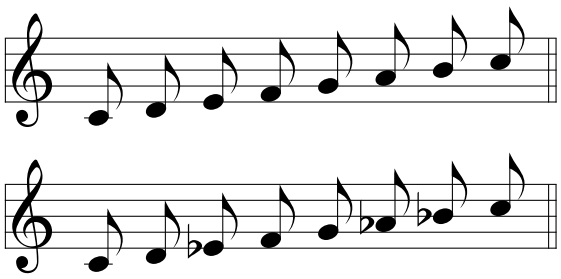

Figura 2.5: Escalas de Do Mayor y Do Menor

conocida es la de Do Mayor, ya que empezando desde Do, pasa por las siete notas fundamentales, pero si se mira de desde la distribución de tonos y semitonos entre sus notas (Tono, Tono, Semitono, Tono, Tono, Tono, Semitono) se puede construir cualquier otra escala mayor dada una nota arbitraria de partida. Las escalas pueden ser, según su distribución tonal Mayores o Menores, así como recibir diferentes nombres según la cantidad de sonidos que incluyan. La escala de Do Mayor es en realidad la escala Heptatónica de Do Mayor ya que incluye siete sonidos, pero otras escalas como las Pentatónicas también son frecuentemente utilizadas en la música moderna. La escala sobre la que se construye una pieza puede ser denominada tonalidad y recibe el nombre de la nota raíz junto con su modo, así una pieza construida con la escala de Do Mayor puede decirse que está en tonalidad de Do Mayor.

En aquellas piezas en las que se trabaja con una determinada tonalidad, resulta cómodo especificar una armadura. La armadura es un conjunto de alteraciones al principio del pentagrama, justo a la derecha de la clave, que indica qué alteraciones tendrán las notas de ese pentagrama a lo largo de toda la canción o hasta que se especifique lo contrario. De este modo, para la escala de Do Menor de la Figura 2.2.2 podría especificarse una armadura que indicase tres bemoles en las líneas de mi, la y si. Si quisiéramos escribir esas notas sin la alteración, para contrarrestar la armadura, habría que usar otra alteración denominada becuadro. El becuadro, también conocido como natural, elimina las alteraciones establecidas por la armadura del pentagrama. Esta armadura no solo resulta conveniente para el compositor, al tener que escribir las alteraciones de la snotas una única vez al comienzo, sino que también es muy útil a la hora de identificar la tonalidad de la pieza, ya que suele ser un indicativo fiable de la misma.

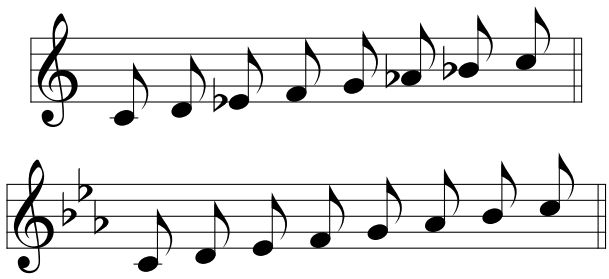

Figura 2.6: Escala de Do Menor sin armadura y con armadura

### 2.2.3. Armonía

Si la melodía es la sucesión de notas en el tiempo, se puede entender la armonía como los conjuntos de notas que suenan a la vez a lo largo de un intervalo de tiempo. Por eso se habla de que el análisis melódico es horizontal y el armónico vertical, aunque esto no es del todo cierto, pues al fin y al cabo la armonía analiza también intervalos horizontales de tiempo (normalmente acotados en compases) y la estructura general de la pieza. Para poder estudiar o crear armonía se debe contar siempre o bien con varias voces o bien con instrumentos polifónicos capaces de producir sonidos simultáneos (Como el Piano o la Guitarra)

Uno de los conceptos más importantes dentro de la armonía es la tonalidad. La armonía define siete grados para una tonalidad, y están relacionados con la escala de dicha tonalidad. Cada grado está referido a cada una de las notas de las calas y se representa mediante un número romano de forma consecutiva, además de un nombre. Los grados más relevantes son el I, IV y V, mientras que los menos importantes o débiles son el II y el VI. A mayores, los grados muy débiles son el III y el VII.

- I (Tónica)
- II (Supertónica)
- III (Mediante)
- IV (Subdominante)
- V (Dominante)
- VI (Superdominante/Submediante)

- VII (Sensible)

Según la teoría, un acorde son tres o más sonidos simultáneos a una distancia de tercera entre sí de forma ascendente. Analizando el tercer grado de la escala de las notas del acorde, podemos determinar si este es mayor o menor. La nota generadora del acorde suele ser la tónica y además la más grave del mismo, si no es así se dice que el acorde está invertido.

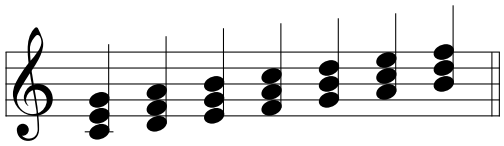

Figura 2.7: Acordes de la tonalidad de Do Mayor

En una escala de modo mayor, el I, IV y V grado son acordes perfectos mayores, el II, III y VI acordes perfectos menores, y el VII un acorde disminuido. En una escala de modo menor, el I y IV grado son acordes perfectos menores, el II y VII son acordes disminuidos, el III es aumentado.

Ciertas combinaciones de acordes producen sensación de tensión, mientras que otras producen sensación de reposo. Algunos acordes, según el contexto, tienen un sentido conclusivo y otros, un sentido transitorio. Estos conceptos son muy importantes en composición, así como en armonización.

La armonización es el proceso de construir una armonía mediante varias voces que, sin modificar la melodía de la pieza, enriquezca la misma reforzando la tonalidad en cada instante. Esto se consigue identificando la tonalidad de la pieza en cada tramo, tomando las notas del mismo y averiguando la escala sobre la que se ha construido la melodía. Dada la escala podemos identificar la tonalidad, y con la tonalidad construir por fin los acordes.

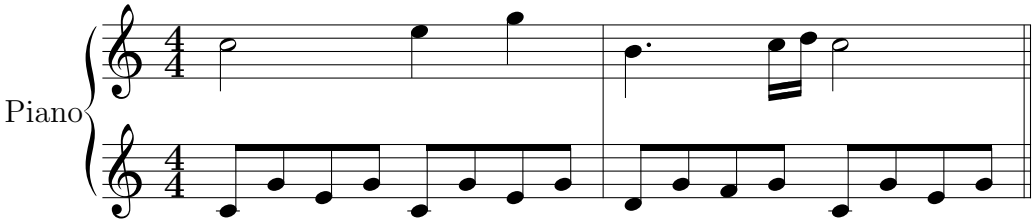


Figura 2.8: Ejemplo de polifonía para Piano (Sonata en Do Mayor, Mozart)

No obstante, existen reglas a respetar, ya que no es tan sencillo como tomar cada nota de la melodía como nota generadora de un acorde y construir el acorde mayor o menor correspondiente a la misma. Por otra parte no basta con tocar el acorde principal de la tonalidad a lo largo de toda la pieza, ya que si bien esto no sería erróneo, podría producir problemas con las fluctuaciones de la melodía, y no cumpliría, en último extremo, la finalidad de enriquecer la melodía.

Los acordes no solo deben respetar la melodía, sino que además deben ser escogidos de tal forma que se enlacen correctamente, normalmente debido a las notas compartidas entre dos acordes consecutivos. Las notas distintas al enlazar dos acordes pueden producir sonidos indeseados con respecto a la melodía o causar saltos melódicos no agradables al oído. Para evitar estos problemas se estudia a su vez, horizontalmente, las nuevas líneas melódicas formadas por las diferentes voces de los acordes. Los diferentes movimientos de estas voces podrían ser, entre otros:

- Paralelo: Dos voces de varios acordes siguiendo una distancia, realizando los mismos saltos melódicos.
- Oblicuo: Una voz representada con una nota de larga duración y otra voz moviéndose libremente.
- Directo: Dos voces moviéndose a la vez de forma ascendente o descendente, pero no en paralelo.
- Contrario: Dos voces moviéndose en sentidos distintos.

En movimientos como el paralelo o el contrario pueden surgir problemas armónicos, como formar un intervalo de octava o quinta justa sobre las mismas voces. El movimiento directo también presenta problemas armónicos, en voces extremas (Bajo y Soprano), si la voz de la Soprano (más aguda) no se mueve por grados conjuntos, se presenta ese problema. En partes intermedias (voces centrales o una voz central y otra extrema), si una de esas dos voces no se mueve por grados conjuntos, ese enlace sería incorrecto. En voces seguidas, se busca evitar los saltos de octava

### 2.2.4. Tesitura

La tesitura define el rango de notas que un instrumento o tipo de voz es capaz de alcanzar, siendo el límite inferior la nota más grave que puede producir y el superior la más aguda. Referido a voz, a veces no se refiere tanto a los límites de las frecuencias sonoras que puede alcanzar sino a aquel rango en el que la voz logra una mejor calidad, es por esto que a veces un mismo cantante puede cantar diferentes partes de la pieza asignadas a tesituras diferentes si fuera necesario.

En música coral se diferencian Tesituras por el género de la persona que canta y se les asignan roles en la polifonía del mismo modo que se asignan a los diferentes instrumentos en una orquesta. Las tesituras corales más comunes son:

- Bajo: Voz masculina encargada de las partes más graves
- Barítono: Voz masculina intermedia
- Tenor: Voz masculina más aguda
- Contra-Tenor: Voz masculina especialmente aguda, normalmente cantada en falsete
- Contralto: Voz femenina más grave, no obstante, situada por encima de la de Contra-Tenor
- Mezzo-Soprano: Voz femenina intermedia entre la de Contralto y Soprano
- Soprano: Voz coral encargada de las secciones más agudas

# Capítulo 3
# Desarrollo

## 3.1. Planificación y Presupuesto

Se ha seguido un proceso de desarrollo en espiral con prototipado. El diseño y desarrollo de los diferentes componentes del sistema se presta a este tipo de ciclo ya que permite crear y probar en cada iteración un producto prototipo, centrándose en las primeras iteraciones en diseñar más e implementar menos y en las últimas solo refinar el software final. Cada iteración cuenta con una planificación de objetivos a alcanzar, detalles del trabajo y correcciones realizadas así como pruebas efectuadas y análisis de objetivos logrados en la iteración.

La principal diferencia con un ciclo de desarrollo en espiral convencional es que al no haber un cliente al que presentar el prototipo y los resultados de cada iteración, esta sección se sustituyó con las reuniones periódicas con el tutor, sirviendo, generalmente, cada una de ellas como cierre de iteración y comienzo del siguiente.

## 3.2. Iteración 1

### 3.2.1. Planificación

El objetivo de esta primera iteración es identificar los formatos, herramientas y componentes del proyecto, así como el diseño inicial de la arquitectura del proyecto, junto con el desarrollo del procesador de ficheros en formato MusicXML a hechos lógicos. Este primer paso tiene una fuerte carga de contextualización del

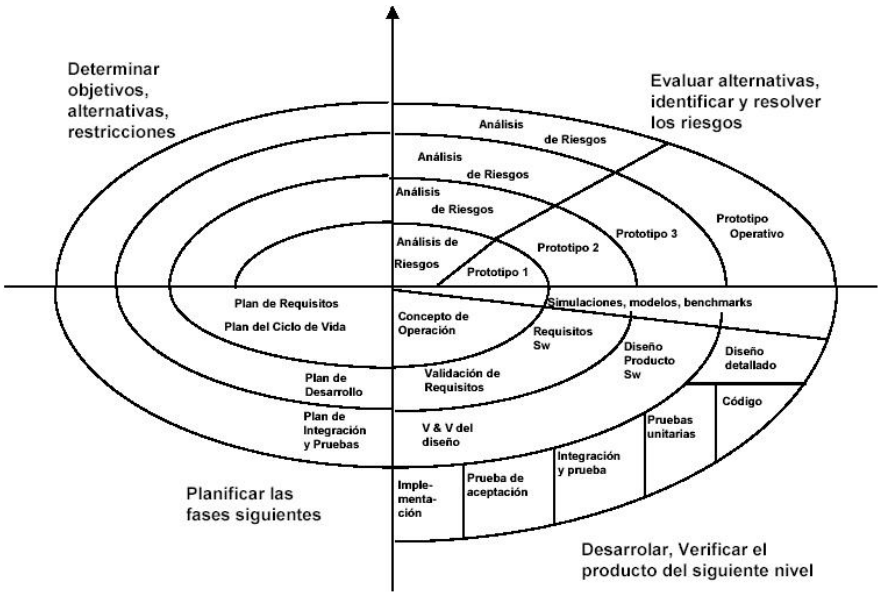


Figura 3.1: Ciclo de desarrollo en espiral típico

trabajo de desarrollo así como la creación del espacio de trabajo en el que se sustenta todo el proyecto.

### 3.2.2. Trabajo

Como software de entrada para las partituras se escogió MuseScore, principalmente por ser *opensource* y por su sencillez. MuseScore no posee una curva de aprendizaje difícil, ya que la introducción de las notas se puede hacer de forma visual en su interfaz mediante el ratón o con el teclado y no requiere mayor preparación para crear y exportar una pieza musical sencilla. Su soporte para *plugins* también resulta interesante, ya que el proyecto podría llegar a transformarse en un *plugin* para la herramienta si se desease una mayor integración con la misma. MuseScore además ofrece soporte para los tres formatos de ficheros musicales contemplados, así como para PDF y otros formatos finales de imagen.

Para el formato de entrada y salida, se compararon las propiedades de MIDI, LilyPond y MusicXML. Los tres formatos ofrecen posibilidades de edición, aunque cada uno sirve a un propósito diferente. MIDI no trabaja con ficheros textuales, sino que codifica de forma binaria toda la información de los eventos de la canción, LilyPond es un formato de texto plano pensado para que el usuario pueda editarlo a mano cómodamente, como si programase. Posee una estructura

de marcado a través de etiquetas de solo apertura o autocerradas y trabaja con identaciones para formar la jerarquía del fichero, esto requiere al usuario escribir mucho menos, pero puede suponer problemas para un *parser* convencional debido a que la identación de LilyPond no está estandarizada. Por último MusicXML se presenta como el formato idóneo para la tarea, ya que al ser una extensión de XML, está orientado a que una máquina pueda leerlo y crear una estructura en memoria con toda la información que necesita para poder extraer los datos de la partitura. Además, la implementación de un *parser* de un lenguaje etiquetado como XML es un problema convencional y fácilmente abarcable.

En cuanto a la tecnología escogida para diseñar y construir el procesador de XML a ASP se optó por las bibliotecas Flex y Bison para C. EL principal motivo para ello fue que fueron las tecnologías empleadas en la asignatura Procesamiento de Lenguajes y durante una de las prácticas de la misma se desarrolló una primera versión de este mismo procesador que ahora se usa en el proyecto. Además Flex y Bison garantizan velocidad y eficiencia en el procesado. Por último pero no menos importante, y ya que el proyecto se está enfocando desde un punto de vista de desarrollo ágil, actualizar los ficheros de código de Flex y Bison es realmente sencillo, lo cual permitirá añadir nuevos elementos a reconocer cuando sea necesario.

Analizando la especificación del esquema de MusicXML de cara a desarrollar el *parser* se identificaron las diferentes partes del mismo. MusicXML incluye información tanto musical como de representación gráfica de los diferentes elementos de la partitura. Todos esta información gráfica es generada automáticamente por el software que exporta el fichero MusicXML (MuseScore para el caso) y resulta irrelevante a la hora de extraer los hechos lógicos presentes en la partitura, por tanto se decidió obviarla.

MusicXML declara inicialmente el número de voces presente en la partitura mediante la etiqueta part-list y sus etiquetas anidadas score-part. Más adelante, las etiquetas part se encargan de contener los compases mediante la etiqueta measure que a su vez contienen las etiquetas note. Son estas últimas etiquetas las que hay que desglosar para extraer los hechos lógicos, aunque la información de las etiquetas measure también es relevante, así como poder asignar un identificador a cada voz de la partitura para poder diferenciarlas a nivel lógico.

La etiqueta note posee dos parámetros visuales, default-x y default-y, que

```
<note default-x="74.65" default-y="-25.00">
    <pitch>
        <step>A</step>
        <octave>4</octave>
    </pitch>
    <duration>1</duration>
    <voice>1</voice>
    <type>quarter</type>
    <stem>up</stem>
</note>
```

Figura 3.2: Ejemplo de nota representada en MusicXML

indican su posición en coordenadas x e y, como ya se ha comentado antes, esta información es meramente visual. La etiqueta pitch describe el sonido de la nota mediante el nombre de la nota usando notación internacional y la octava de la nota. Duration especifica la duración en tiempos del compás de la nota actual, voice asigna este sonido a la voz correspondiente, type indica la figura mediante el sistema fraccionario y stem dice si la plica de la nota es ascendente o descendente, de nuevo esto es meramente visual.

Ya que, inicialmente, el armonizador solo contemplará partituras uniformes (es decir, con un solo tipo de figura presente), unicamente necesitamos saber la voz, el compás, el tiempo en el que ocurre la nota relativo al compás y el sonido de la nota.

Con esta información en mente se procedió a desarrollar una versión actualizada del procesador de MusicXML a hechos en ASP. Se hizo uso de Flex para el análisis léxico y de Bison para el análisis sintáctico. El primer paso fue identificar los diferentes tags relevantes para el análisis léxico, estos son note para las notas, step para el ritmo, rest para los silencios, chord para los acordes, octave para la octava de cada nota, alteration para identificar sostenidos y bemoles, part para las diferentes voces de la pieza y por último elementos relevantes para el análisis de XML pero carentes de significado real para los hechos lógicos de la partitura tales como la apertura y cierre de etiquetas, campos textuales o símbolos varios.

En el código de Bison se diseñó la gramática primero y se integró junto con el código C encargado de tomar los datos extraídos por el *parser* y exportarlos a un fichero de hechos lógicos listo para ser interpretado. La gramática diseñada

para ello parte de una regla inicial que puede derivar en un bloque compuesto por etiquetas anidadas o en un error si el fichero no tiene el formato y estructura adecuados. La gramática inicial empleada fue la siguiente:

Los bloques de etiquetas se descomponen en tokens según su contenido o en dos partes (part1 y part2) dependiendo de si la etiqueta es autocerrada o no. Los errores que se pueden encontrar en este punto son que la etiqueta se abra pero no se cierre o que haya elementos de más no reconocidos tras la segunda parte de la misma. Las dos partes de cada etiqueta no autocerrada se corresponden respectivamente con la apertura de la etiqueta (part1) y con el contenido y cierre de la misma (part2). Por último se hace uso de una regla recursiva que permite anidar los bloques de etiquetas y contenido.

Se ha incluido en este procesador un par de opciones que pueden ser utilizadas en la llamada del mismo mediante línea de comando y que resultarían útiles más adelante. La opción -o permite especificar el nombre del fichero de salida resultante, mientras que la opción -s permite establecer una subdivisión arbitraria para las notas de la partitura.

### 3.2.3. Pruebas

Las pruebas realizadas al *parser* revelaron que existía un problema de análisis al no poder verificar de forma sencilla que cada etiqueta se cerraba de modo correcto, es decir, que el nombre de la etiqueta que cierra un bloque sea el mismo del que la abrió, se implementó una pila en C para esta tarea. La implementación se realizó de modo que la pila no tuviese un tipo definido de partida, por flexibilidad, mediante el uso de punteros a void y el *typecast* de los mismos en ejecución, aunque en el caso del *parser*, solo se usó el tipo char*.

La implementación de la misma se hace mediante estructuras típicas de C enlazadas mediante punteros unidireccionales y el acceso a los datos de la pila se realiza mediante los conocidos métodos para operar con este tipo de datos (new, isEmpty, push, pop, etc.)

Además de la adición de la estructura de pila para verificar esto se incluyeron más opciones de error en varias de las reglas gramaticales de modo que resultase más fácil la depuración del *parser*. Esto reveló que había fallos en ejecución, ya que los resultados no eran los esperados. Este problema se debía a que la recursión se realizaba inicialmente mediante la posibilidad de que block derivase en otro

```
S : version doctype block
    block
version :  OPTAG QUES TEXT attr QUES CLTAG
doctype :  OPTAG EXCL DOCTYPE doctags docurl CLTAG
doctags :  /*...*/
            TEXT doctags
docurl :  /*...*/
            KVOTHE TEXT docwords KVOTHE docurl
docwords :  /*...*/
            SLASHTAG docwords
            TEXT docwords
block : OPTAG REST SLASHTAG CLTAG
        OPTAG TEXT attr SLASHTAG CLTAG
        OPTAG ALTER CLTAG TEXT OPTAG SLASHTAG ALTER CLTAG
        OPTAG CHORD SLASHTAG CLTAG
        OPTAG OCTA CLTAG TEXT OPTAG SLASHTAG OCTA CLTAG
        OPTAG STEP CLTAG TEXT OPTAG SLASHTAG STEP CLTAG
        part1 part2
part1 : OPTAG NOTE attr
        OPTAG PART_ID KVOTHE TEXT KVOTHE
        OPTAG TEXT attr
part2 : CLTAG body OPTAG SLASHTAG NOTE CLTAG
        CLTAG OPTAG SLASHTAG TEXT CLTAG
        CLTAG body OPTAG SLASHTAG TEXT CLTAG
        TEXT EQUAL KVOTHE TEXT KVOTHE attr
body :  body block
        body TEXT
        block
        TEXT
```

Figura 3.3: Gramática inicial utilizada por el procesador de MusicXML a hechos lógicos

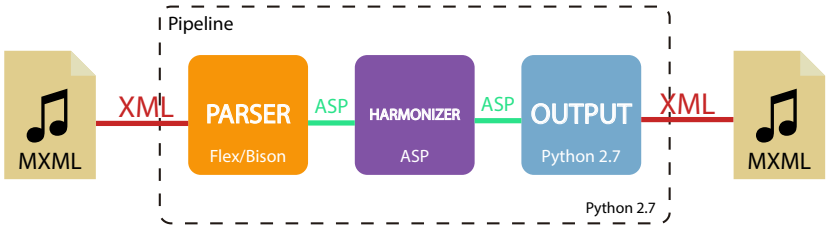


Figura 3.4: Diagrama del planteamiento inicial de la arquitectura del sistema

block, así que se descompuso esta regla en un nuevo elemento body que añadía un paso más y abstraía los bloques de etiquetas y contenido pero a un nivel de granularidad algo más pequeño que *body* en sí. Tras la inclusión de este paso intermedio, el *parser* aún no terminaba de comportarse como era esperado y hubo que incluir reglas específicas para comprobar tags con formatos especiales tales como <?xml version> o <!DOCTYPE>. Por último se comprobó que el *parser* no contemplaba inicialmente tags autocerrados en su forma <tag></tag> es decir, sin contenido. Al corregir estos detalles, cualquier pieza musical era transformada a hechos musicales a la perfección.

### 3.2.4. Resultados

En esta primera iteración se completaron todos los objetivos marcados en la planificación de la misma. Habiendo preparado correctamente el espacio de trabajo y definido bien los formatos sobre los que se trabajaría y habiendo implementado una primera versión del procesador de MusicXML a hechos lógicos.

## 3.3. Iteración 2

### 3.3.1. Planificación

El objetivo de esta iteración fue implementar una primera versión del pipeline que incluyese un módulo de armonización básica en ASP. Para ello se fijaron una serie de restricciones que simplificaban algunos problemas detectados durante el análisis. Inicialmente no se tendrán en cuenta las notas adyacentes en la partitura ni subdivisiones de las mismas, no obstante y de cara a futuras iteraciones, se establece una figura base como período de análisis horizontal de la melodía y

una figura base de subdivisión, aunque para esta iteración ambas se establecerán a negra (*quarter note*) de modo que se asignará el acorde correspondiente a cada golpe rítmico de la partitura teniendo en cuenta las múltiples voces que la componen de manera vertical.

### 3.3.2. Trabajo

Se modificó el procesador de MXML a ASP y se ha incluido una opción para subdividir la partitura de forma automática en base a la nota más breve de la partitura o forzar toda la partitura a un solo tipo de figura omitiendo aquellas figuras de menor duración, aunque esto lleve a resultados incorrectos musicalmente. La salida deseada y el formato de entrada para el módulo de procesado ASP usan el formato `note(voz, tono, tiempo)` siendo voz el número de la voz que interpreta la nota, tono un valor numérico asociado a la nota, no en frecuencia sino calculado como el número semitonos a partir de una nota base A0, es decir, el La más grave que puede interpretar MIDI; y tiempo la posición de la nota en la partitura.

Para el desarrollo del módulo de procesado ASP se sigue el proceso conocido como *Generate and Test*, muy habitual en el paradigma. Como su nombre indica, consta de dos partes: generación y prueba. En la generación se usan reglas que definen todas los resultados posibles del problema para después, en la fase de prueba, restringirlos mediante reglas que prohíben total o parcialmente ciertas combinaciones al exponerse a hechos lógicos aportados por el procesador de MXML a ASP. Esta metodología se usó desde el principio en cada iteración a partir de la presente para desarrollar la parte en ASP correspondiente ya que aunque las reglas de generación permanecerán intactas, estas serán revisadas por si fuera necesario cambiarlas o añadir alguna regla nueva.

En esta primera aproximación, las reglas de generación establecen los posibles acordes que compondrán la solución y se realiza una asignación acorde-unidad rítmica (como ya se comentó, la unidad rítmica base inicialmente será la negra) en base a las notas presentes para un instante dado en cada voz. Para realizar esto es necesario interpretar cual es el grado de cada nota presente en la escala de la pieza, (de nuevo, por restricciones iniciales se considera Do mayor) ya que de este modo podremos especificar una serie de reglas que abstraigan el hecho de qué nota suena en si y lo sustituyan por qué grados aparecen en cada momento dado de la

partitura, pudiendo deducir así qué acorde es el más correcto de entre todas las posibilidades, a más notas presentes de la escala correcta, menos posibilidades, de aparecer notas no pertenecientes a la escala, no se podrá resolver el acorde y el programa se detendrá con un error.

La fórmula empleada para derivar los grados de la escala es $[(valor - base) mod 12]$ siendo valor el valor númerico asignado a la nota y base la nota de la que se parte para calcular los grados. Esta expresión resulta en los semitonos de distancia entre la nota base y la nota actual. La operación de módulo es necesaria para abstraer la octava a la que pertenece la nota, aunque más adelante esto pueda ser un hecho relevante. Con una comparación directa del valor obtenido con una distribución dada según el modo (Mayor o Menor) de la escala es posible asignar los grados correspondientes a cada nota. Por último se generan hechos que indican cuantas voces toman parte en la pieza, así como los tiempos en los que suena al menos una nota.

Se crearon dos ficheros a mayores que especifican los acordes a considerar, otorgándole un nombre en notación numérica romana y creando múltiples predicados que especifican qué grados de la escala pertenecen a dicho acorde. De este modo no es necesario especificar en ningún momento cuantas notas tendrán los acordes con los que se trabaje, y aunque principalmente se trabajará con acordes de tres notas, esto permitiría ampliar la complejidad en el futuro al incluir acordes de dos o cuatro notas. Teniendo en cuenta los posibles acordes, y mediante una restricción de cardinalidad, se generan todas las soluciones iniciales, haciendo que solo se asigne un acorde a cada tiempo en el que suena al menos una nota. A mayores se especifica una restricción de integridad que anula cualquier solución en la que, para un tiempo en el que esté presente una nota concreta, el acorde asignado en la solución comprobada no contenga esa nota.

A mayores se ha incluído como parte de esta iteración, el diseño e implementación de un pipeline en python que automatice las llamadas al procesador MXML a ASP y al módulo de armonización. En este primer prototipo se implementó una pequeña funcionalidad de interpretación de la salida del módulo ASP a un vector de soluciones. En futuras iteraciones, se añadirá más funcionalidad a dicho pipeline, como más opciones de entrada o diferentes representaciones de las soluciones ofrecidas por el módulo ASP haciendo uso de alguna librería que permita representar de nuevo la partitura en el formato deseado. El pipeline agrega y combina

las diferentes opciones de los módulos a los que llama, en este caso se incluyeron únicamente las opciones -n, que especifica a clingo cuantas soluciones mostrar, y -m, encargada de alternar entre los modos de armonización mayor y menor. El pipeline no contepla aún opciones relacionadas con el módulo de procesado ya que la opción -s incluida en la anterior iteración está fijada a 1 por las restricciones especificadas para este prototipo.

Incluir diagrama de secuencia

### 3.3.3. Pruebas

Se han realizado pruebas individuales a los módulos de procesado y armonización, además se han realizado pruebas al pipeline creado, sirviendo de este modo de pruebas de integración de los diferentes módulos y de la herramienta al completo.

Para el procesador se han probado múltiples ficheros de entrada así como las distintas opciones que contempla. No presenta problemas para ficheros MusicXML estándar generados por Musescore y programas similares, ante la ausencia de fichero, ficheros incompletos, ficheros XML que no sean MusicXML o ficheros que directamente no sean XML el procesador detiene la ejecución y produce el mensaje de error adecuado en cada caso. En cuanto a las tres opciones que presenta esta primera versión, la opción de ayuda -h detiene la ejecución independientemente de la presencia de otros parámetros y muestra el modo de uso del módulo parser por sí solo, la opción para especificar el fichero de salida -o funciona correctamente en cualquier caso, excepto cuando el directorio especificado para almacenar el fichero de salida no existe. En la siguiente iteración se planteará la posibilidad de dejar esto de este modo intencionadamente o si forzar la creación de los directorios necesarios. Por último la opción de subdivisión -s, que para este primer prototipo ha de ser especificada manualmente o dejada en su valor por defecto funciona para cualquier valor. Esto no es del todo correcto ya que la subdivisión solo debería ser posible para potencias de 2, en el segundo prototipo se corregiría esto.

El módulo de armonización no presenta una gran complejidad de prueba ya que solo depende del fichero de entrada. Si este no está presente no produce resultado alguno, y en caso de estarlo y no ser correcto, produce un resultado de insatisfacibilidad. En caso de existir alguna solución esta es correctamente

devuelta por pantalla. Las diferentes piezas probadas, pese a ser sencillas por las limitaciones de este primer prototipo, ofrecieron resultados de armonización correctos.

Por último, en las realizadas para el pipeline se analizaron las diferentes opciones y casos de entrada de la herramienta. Ante los casos erróneos donde el módulo de procesado fallaba y no generaba fichero LP de salida, el pipeline no detenía la ejecución y llamaba al módulo ASP igualmente, aunque este fallaba al no tener un fichero de hechos lógicos con el que funcionar, no obstante el pipeline si que debería comprobar esto y por tanto se corrigió. Con ficheros de entrada adecuados el pipeline realizaba una ejecución completa y recogía los resultados ofrecidos por el módulo ASP, ofreciendo por pantalla una representación más amigable así como un resultado de satisfacibilidad. La opción -m (alternar entre modo mayor y menor) funcionaba como se esperaba, restringiendo los dos posibles valores de la opción a los especificados ("major" o "minor") y deteniendo la ejecución en caso de encontrar un valor incorrecto, mientras que la opción -n (número de soluciones exploradas) restringe también correctamente los valores de N a únicamente enteros y no hace falta mayor comprobación ya que cualquier valor entero es válido como opción para el módulo ASP. Ambas opciones producían el efecto deseado en la ejecución del pipeline.

### 3.3.4. Resultados

Este primer prototipo se completó en dos semanas, siendo los mayores problemas de la iteración la falta de soltura con el paradigma y el lenguaje, así como detalles técnicos de falta de librerías y software. Se logró implementar una versión inicial del pipeline junto con una temprana pero funcional versión del módulo de armonización.

## 3.4. Iteración 3

### 3.4.1. Planificación

El objetivo de esta iteración es completar el desarrollo de un segundo prototipo de la herramienta que incluya las siguientes mejoras:

- Subdivisión real y automatizada de las notas de la pieza a la longitud de la nota mínima presente en la partitura
- Especificación en el pipeline de la longitud del tiempo de análisis horizontal
- Inclusión de la posibilidad de realizar análisis horizontal en ASP
- Flexibilización de los resultados del módulo de armonización, es decir, en vez de prohibir las soluciones erróneas, se anotarán los errores en la partitura y se escogerá aquella solución que minimice el número de errores

Además se incluirán en este prototipo las correcciones a los errores de ejecución de los módulos no corregidos en el primero.

### 3.4.2. Trabajo

Se modificó el *parser* substancialmente ya que este imprimía a un fichero según procesaba las notas. Esto no planteaba problema alguno si la subdivisión se especificaba de antemano mediante el parámetro correspondiente, pero sí que resultaba complicado mantener esta aproximación si la unidad de subdivisión debía calcularse al mismo tiempo que se procesaba la partitura en MXML. Se plantearon dos soluciones, o bien incluir en el pipeline en python un análisis previo a la conversión de MXML a hechos en ASP que dedujese cual era la nota de menor longitud y la usase como parámetro en la llamada al *parser* o bien se modificaba el comportamiento del anterior para realizar simultáneamente ambas tareas.

Se optó por la segunda opción por motivos de coherencia con el sistema, es decir, no incluir funcionalidad innecesaria y replicada en el pipeline, cuya tarea es simplemente manejar las entradas y salidas de los diferentes módulos, y por motivos de eficiencia, ya que como se ha mencionado no hay necesidad de procesar el mismo fichero dos veces, siendo una de ellas en un lenguaje interpretado en vez de compilado, lo que añadiría un sobrecoste temporal evitable.

Los cambios implementados en el *parser* implicaron principalmente incluir un nuevo tipo de dato nota para almacenar la información de las notas de la partitura y una nueva pila que contuviese las notas extraídas del MXML. Una vez procesado todo el fichero de entrada, hallada la nota más breve y almacenadas las notas en la pila, ésta se vacía y se imprimen en el fichero de hechos lógicos de

salida teniendo en cuenta la subdivisión pertinente, bien sea esta la calculada o la especificada por parámetro. En caso de especificar una subdivisión no válida, es decir, de mayor longitud que alguna de las notas presentes en la partitura, se imprime por pantalla un mensaje de error y la nota no es subdividida. Esto produce comportamientos no deseados a la hora de realizar la deducción de la armonía, ya que es necesario trabajar siempre con piezas con notas de longitudes iguales a lo largo de toda la partitura.

El módulo de armonización incluye una nueva constante que indica la longitud del intervalo de tiempo mínimo de análisis armónico horizontal, a su vez es posible especificar en la llamada al módulo el valor de esta constante. Se ha modificado, por tanto, la regla que restringe las posibles soluciones para analizar en dichos intervalos de tiempo. Además dicha regla se ha suavizado y en vez de ser una restricción de integridad, esta activa un nuevo predicado error(voz, grado, tiempo) que indica los grados erróneos presentes en la partitura que no encajan con el acorde asignado para la solución. Posteriormente se realiza un proceso de optimización consistente en la minimización de el número de estos predicados de error, es decir, aquellas soluciones con menor número de errores serán las óptimas. Además en caso de no encontrar una solución con cero errores, los errores aparecen también en la salida para que el usuario pueda conocer qué tiempos contienen notas equivocadas.

Se han incluido en el pipeline opciones tanto para indicar al procesador de MusicXML a hechos lógicos una subdivisión específica (opción -d) como para indicar al módulo de armonización el intervalo horizontal de armonización (opción -s). Se han implementado en el pipeline, con vistas al futuro de módulo de salida, una serie de clases para almacenar los resultados y poder devolverlos más tarde en el formato más conveniente. `Error` y `Chord` son clases para almacenar y representar los acordes asignados en la solución así como los errores presentes en la partitura, es decir, aquellas notas que no encajan con los acordes asignados. A su vez, HaspSolution es una clase orientada a almacenar y representar soluciones completas incluyendo una colección de objetos `Chord` y otra de objetos Error, así como el grado de optimización de dicha solución. Por último, la clase `ClaspResult` almacena la salida sin procesar de una ejecución del módulo ASP, aunque cuando esta es instanciada, dicha salida se procesa y se crean una colección de objetos `HaspSolution` conteniendo sólo aquellas soluciones que alcancen

un cierto valor de optimización. La relación entre las clases así como sus métodos se detallan en el diagrama de clase pertinente.

### 3.4.3. Pruebas

Se han continuado con las pruebas a los diferentes módulos, dando por válidos los resultados de las pruebas de la anterior iteración se procedió a probar las nuevas funcionalidades de cada módulo.

En el *parser* se comprobó que la autosubdivisión funcionaba correctamente al introducir partituras con diferentes longitudes de notas, además en caso de forzar una subdivisión incorrecta se producía el mensaje de aviso adecuado explicando el problema. Si la subdivisión forzada era correcta funcionaba como debía. El nuevo tipo de datos, nota, así como la pila para almacenar las diferentes notas de la partitura fueron probados junto con esta nueva funcionalidad, ya que fueron implementados expresamente para su funcionamiento.

En el módulo ASP se probaron piezas más y menos complejas, algunas creadas de forma artificial para producir errores y se vio que el nuevo análisis funcionaba y detectaba dichos errores, no obstante seguía produciendo soluciones válidas marcando aquellas notas erróneas.

En cuanto al pipeline se comprobó que las nuevas opciones en la llamada a éste funcionasen correctamente, ya que las dos nuevas opciones son simplemente parámetros que se pasan más adelante al módulo *parser* o al módulo ASP, bastó con realizar una comprobación de validez del valor pasado en la llamada. Se probaron además las cuatro clases de almacenamiento de las soluciones diseñadas e implementadas para este prototipo, siendo `ClaspOutput` la única que tuvo que ser probada más exhaustivamente ya que es la que realiza el procesado de la salida del módulo ASP, el resto simplemente cuentan con constructores y funciones de representación textual.

### 3.4.4. Resultados

Durante el desarrollo se cumplieron los objetivos marcados en la planificación.

La subdvisión, tanto como fijada por parámetro como automatizada, ha sido implementada y probada freciendo resultados correctos. El análisis horizontal produce los resultados esperados para las longitudes especificadas. Los resultados

de aquellas piezas creadas expresamente para fallar ya no detienen el programa si no que se ofrecen resultados marcando los errores. Se han corregido errores detectados en la iteración anterior.

## 3.5. Iteración 4

### 3.5.1. Planificación

El objetivo de esta iteración es crear un tercer prototipo que busca no sólo refinar la armonización como el anterior si no ampliar funcionalidad de la herramienta. Se busca incorporar un módulo de salida escrito en python al cual el pipeline se encarga de pasarle los datos formateados correctamente para que dicho módulo exporte al formato deseado.

### 3.5.2. Trabajo

El módulo ASP se ha aumentado para incluir generación de notas en un número de voces adicionales que puede ser especificado por parámetro. Además se creó un fichero de conversiones encargado de traducir valores de notas a grados, octavas y viceversa. De este modo los grados generados durante el procesado de la partitura pueden ser traducidos de vuelta a un valor de nota para que el módulo de salida llamado desde el pipeline reconstruya la pieza. Para la generación de notas en las nuevas voces se ha impuesto una única restricción fuerte, que dos notas consecutivas no realicen un salto melódico de dos octavas o más, mientras que mediante predicados de minimización se controla la cantidad de saltos de una quinta realizados por una misma voz.

Se ha diseñado implementado una nueva clase de almacenamiento `Note` y se ha incluido un método a `HaspSolution` que transforma una solución al tipo de dato requerido por el *toolkit* Music21 para producir los diferentes formatos de salida. El almacenado de las notas en la coleccion de voces se ha realizado mediante una estructura de diccionario, utilizando el identificador numérico de la voz como índice, resultando muy cómodo añadir ordenadamente a cada entrada en el diccionario los objetos `Note` o futuros objetos similares, al ser colecciones multi-tipo.

Se ha optado por incorporar **Music21** al módulo de salida principalmente por la cantidad de formatos con los que puede trabajar, tanto en entrada como en salida, y aunque lo ideal será exportar un fichero MusicXML, la idea de poder generar PDF, MIDI o Lilypond resulta más que atractiva. Este módulo toma un objeto `ClaspResult` como entrada y haciendo uso del método de una de los objetos `HaspSolution` contenidos en él, representa dicha solución en el formato adecuado. Music21 posee una representación interna de representación de objetos musicales demasiado compleja como para utilizarla directamente con la salida del módulo de armonización, no obstante la conversión de los objetos de almacenamiento definidos es relativamente sencilla y gracias a esto resultó cómodo construir una serie de métodos en el módulo de salida que contruyesen la partitura en términos de Music21 para poder ser representada.

En el *pipeline* se ha incluido una opción para especificar el número de voces adicionales que deben ser añadidas y otra opción para especificar el formato de salida. Este componente ahora se encarga también de llamar al módulo de salida. Además la llamada al módulo de salida produce un objeto de partitura Music21 que puede ser manejado de diferentes modos según nuevos parámetros, -f permite especificar el formato de salida (Principalmente PDF, MIDI o XML) siempre y cuando se disponga de herramientas en el PC para generar dicho formato. Por otra parte, usando la opción -S se puede pedir al pipeline que no genere un fichero nuevo, si no uno temporal para mostrar o reproducir al terminar la ejecución. La opción -o permite especificar el nombre del fichero, de no estar presente, tomará el mismo nombre que el del fichero MusicXML de entrada.

### 3.5.3. Pruebas

Se probó el nuevo conjunto de acordes con las piezas utilizadas anteriormente y se vio que estos nuevos acordes se tomaban en consideración correctamente a la hora de armonizar.

Así mismo se probaron clases de almacenamiento de modo individual y en combinación unas con otras a traves del parser. Se hizo especial hincapié en el nuevo método de procesado de la salida de la clase `HaspSolution`. Las clases de almacenamiento dieron ligeros problemas que se solucionaron inmediatamente quedando corregidas y funcionales.

El módulo de saida se probó exhaustivamente pero no produjo ningún fallo

que no se puediera depurar en el momento.

### 3.5.4. Resultados

Esta iteración resulto excepcionalmente corta y se alcanzaron los objetivos relativamente rápido, ya que la decisión de usar Music21 se tomó bastante más rápido de lo esperado al ver que se adaptaba especialmente bien al trabajo a realizar.

Por otra parte, el trabajo a realizar en el módulo de armonización fue más un estudio sobre el papel para incorporar los nuevos cambios. Las voces nuevas se añadían correctamente, aunque los tiempos de ejecución se disparaban en algunos casos para partituras muy largas. Los nuevos acordes se usaban con buena frecuencia, ya que son más fáciles de encajar al tener más notas posibles.

El módulo de salida funcionó excepcionalmente bien, aunque no producía una salida fidedigna al no tener información sobre la longitud original de las notas y algunos detalles como los instrumentos que debían interpretar cada voz y, aunque menos importante, tampoco conservaba ni el nombre de la pieza ni su compositor.

## 3.6. Iteración 5

### 3.6.1. Planificación

El cuarto prototipo tiene como objetivo tener en cuenta los tiempos débiles y fuertes de la partitura para refinar ciertas preferencias ya incluidas anteriormente o posibles nuevas preferencias basadas en dichos tiempos. Para esto será necesario modificar el módulo ASP, el *parser* y el pipeline. Además se busca refinar el módulo de salida para ajustar la representación en partitura de la solución e incluir mejoras visuales. Como en los anteriores, este prototipo corregirá los errores detectados en el anterior.

### 3.6.2. Trabajo

En el procesador de MusicXML a hechos lógicos se han incluido dos nuevas funciones principales:

- **Análisis de medida de compás:** Determinar cuantas figuras de qué longitud posee el tipo de compás base de la pieza.
- **Distincisón de tipos de silencios:** Diferenciar silencios completables de aquellos que deben ser respetados como silencios.

La primera atiende a la necesidad de indicarle al módulo ASP la métrica del compás, así como su subdivisión para poder identificar tiempos débiles y fuertes, mientras que la segunda atiende a la carencia de una manera sencilla de representar en música con un símbolo aquellos huecos que siendo silencios en la partitura original deben ser completados por el módulo ASP y no ser tratados como silencios en sí.

Para la métrica del compás, se implementó en el procesador la capacidad de itentificar las diferentes métricas de compases así como el tiempo en el que ocurren, aunque por comodidad y sencillez, se ha asumido que un cambio rítmico en el compás debe ocurrir en todas las voces a la vez. Esto crea predicados `measure(T,N)` que indican para un tiempo T, el número N de figuras que componen el compás. No es necesario especificar la longitud de la figura base ya que esta se fija y normaliza durante el análisis de la partitura. De este modo, aunque el compás fuese un 4/4 (compases de cuatro negras), si la subdivisión detectada y empleada es de corchea, será traducido a 8/8 (compases de ocho corcheas).

Para poder especificar en el editor de partituras silencios "verdaderos" y silencios completables se ha optado por una solución relativamente sencilla, no sólo de identificar por el *parser* si no también fácil de usar por el usuario final de la herramienta. Mediante la notación de letras de la pieza, se pueden indicar en cada voz los intervalos de tiempo en los cuales los silencios deben ser tratados como completables. Para ello solo hace falta escribir los símbolos `[` y `]` al principio y final del intervalo respectivamente. Para poder detectar este intervalo se incorporó un tipo de dato cola genérica al procesador para poder interpretar bien el momento de inicio y de cierre de estos símbolos y marcar así los intervalos a completar de manera adecuada.

En el módulo de armonización se han definido acordemente varios predicados nuevos, como `busybeat(V,B)` siendo B un tiempo de la voz V que indica aquellos tiempos en los que inicialmente ya hay una nota o silencio, todos aquellos `beat(B)` que no son `busybeat(V,B)` son `freebeat(V,B)`, y de-

ben ser completados. Además se han definido los predicados `strong_beat(B)`, `semistrong_beat(B)` y `weak_beat(B)` que corresponden a los tiempos fuertes, semifuertes y débiles de cada compás en la partitura. Gracias a esto se han creado predicados que matizan los diferentes errores de la partitura `error_in_strong(V,G,B)`, `error_in_semistrong(V,G,B)` y `error_in_weak(V,G,B)` que definen en qué tipo de tiempo ocurren los errores de la pieza y permite minimizarlos con diferentes prioridad (a más fortaleza de tiempo, más prioridad). Por último y para dar más flexibilidad en la búsqueda de la armonización correcta se ha incluído el acorde de dominante séptima (V7) en los modos mayor y menor.

Se ha incluido una clase de almacenamiento `Rest` para diferenciarla de `Note` y no tener que usar valores negativos en el valor del objeto `Note`. Ya que las entradas del diccionario que representa las diferentes voces de la partitura puede almacenar cualquier combinación de tipos de elementos, se seguirá una convención para poder almacenarlos e iterar sobre ellos sin problema.

El módulo de salida se ha refinado para representar mejor las notas incluyendo información del tipo de compás, clave y duración de las figuras. Además se colorean en rojo los errores detectados y se anotan en los tiempos adecuados los acordes inferidos por el módulo ASP.

Por último, el pipeline cuenta con una nueva opción -t, que permite especificar un tiempo máximo de búsqueda del óptimo en el módulo ASP, ya que para piezas largas, el espacio de búsqueda crece muchísimo y es necesario poder limitar el tiempo de ejecución. Además, al final de la ejecución, y tras enumerar todas las soluciones escogidas por el módulo ASP, permite al usuario escoger qué solución exportar o mostrar, teniendo por defecto la última, ya que se supone más refinada.

### 3.6.3. Pruebas

Las pruebas realizadas esta iteración se centraron en verificar que el procesador identificase de modo correcto los tipos de compás y los tiempos completables. Se plantearon casos de partituras solo con silencios completables, solo con silencios "verdaderos" y otras mixtas. Todas ofrecieron los resultados esperados.

En cuanto al módulo de armonización se probó que los tiempos marcados como `freebeat` se completasen correctamente haciendo uso de las piezas creadas para probar también el procesador. Así mismo se verificó que se produjesen los resultados de subdivisión en tiempos fuertes y débiles adecuados.

Se probó la nueva opción de timeout del pipeline, comprobando que aceptaba valores numéricos correctamente y que esto se reflejaba en el tiempo de búsqueda de las diferentes soluciones.

### 3.6.4. Resultados

Se cumplieron los objetivos marcados para esta iteración, pero la reunión con el tutor al final de esta reveló unos errores de diseño que serían corregidos en la siguiente.

La identificación de tiempos débiles y fuertes no se estaba realizando correctamente en términos musicales al faltar un dato importante: el tipo de subdivisión del compás. Siendo esta binaria o ternaria según el tipo de compás, la figura base del mismo y el intervalo de armonización. Al no tenerse en cuenta no podía ofrecer resultados realmente buenos en este aspecto.

Al trabajar con diferentes partituras se detectó un inconveniente importante en la identificación de intervalos a completar, y es que al hacer uso de las letras para definir los intervalos, no se podían poner estas marcas de inicio y fin bajo silencios, al no tener sentido que un trozo de letra a cantar figure bajo un silencio. Además resultaba complejo trabajar con intervalos abiertos por alguno de los extremos.

## 3.7. Iteración 6

### 3.7.1. Planificación

En esta iteración se perfeccionarán algunas funcionalidades de los diferentes módulos y se añadirá un sub-módulo de preferencias melódicas al ya creado módulo ASP. El quinto prototipo busca ser, esencialmente, una mejora del anterior que corrija los errores detectados en la anterior evaluación del mismo.

### 3.7.2. Trabajo

Para facilitar la entrada de silencios que representan huecos completables por el módulo ASP se ha cambiado el enfoque, y se ha abandonado el delimitado de secciones completables con corchetes en las letras de la canción por suponer

algunos problemas al no poder ubicar dichas letras en tiempos en los que haya un silencio, por claridad y por no interferir con las posibles letras de una partitura real no creada ni modificada textitad-hoc para el programa. MusicXML permite marcar elementos de la partitura como no visibles, esto solo afecta a la hora de imprimir en papel dicha partitura y a nivel musical no interfiere con ningún elemento. Además, la visibilidad de una nota o silencio puede ser fácilmente alterada en cualquier editor de partituras desmarcando una casilla al clicar sobre dicho elemento, lo cual facilita mucho el marcado de estos tiempos completables. Esto se reflejó en el procesador de MusicXML a hechos lógicos, que en vez de contemplar los dos símbolos utilizados anteriormente, ahora solo tiene que comprobar la visibilidad de un elemento para determinar si asignar un tiempo completable a dicho tiempo.

Para refinar la subdivisión en tiempos débiles y fuertes, se reimplementaron tanto en el módulo ASP como en el procesador de hechos lógicos a MusicXML, esto fue debido a que no es sencillo establecer dichos tiempos aritméticamente sólo teniendo en cuenta la cantidad de notas del compás, sino que también se ha de tener en cuenta el tipo y subdivisión del compás con respecto a la nota de referencia usada para armonizar. Para esto es importante no normalizar el compás leído en el fichero XML y generar nuevos predicados indicando los valores del compás sin modificar. En el módulo ASP se ha incluido, de modo similar a los acordes, una tabla de tipos de compás y su subdivisión teniendo en cuenta el compás y la longitud del tiempo de armonización. Se han incluido en dicha tabla los compases más habituales, pero de nuevo, al igual que con los acordes incluidos en los ficheros correspondientes, pueden ser ampliados si el usuario lo necesitase. Una vez inferido el tipo de subdivisión del compás, calcular los tiempos débiles y fuertes se puede realizar de forma aritmética, de modo similar a como se hacía en el prototipo anterior. Debido a la pérdida de información resultante de uniformizar las notas de la partitura no es posible detectar los subtiempos fuertes y débiles dentro de secuencias de corcheas, semicorcheas, etc. Además se ha optado por descartar la inferencia de tiempos semifuertes ya que solo añadían complejidad a la búsqueda del óptimo y no aportaban mejores resultados, los que en el prototipo anterior eran semifuertes, en este son directamente tiempos fuertes.

Se creó un sub-módulo ASP de preferencias melódicas que busca alcanzar una

optimización mayor a la hora de generar nuevas voces o cubrir tiempos completables con algunas mejoras que atienden, principalmente, a la secuencia de notas de una misma voz. Se ha definido el salto melódico de forma genérica y éste es minimizado en notas consecutivas usando el tamaño del salto como peso a la hora de minimizarlo, con esto se logra una melodía más continua y sin saltos erráticos, al mismo tiempo que se limita el espacio de búsqueda considerablemente. Se ha inferido un nuevo predicado tendencia que analiza, dado un par de notas de una voz, si la tendencia melódica es ascendente, descendente o no varía. Además se ha definido otros predicados relacionados que analizan si la tendencia entre dos voces diferentes es la misma o si es contraria, maximizando estos predicados se consigue un ligero efecto de imitación de tendencia entre voces.

Se estudió la posibilidad de incluir también en este sub-módulo de preferencias melódicas la detección de patrones secuenciales armónicos y melódicos comunes como las secuencias de sextas, que siguen un determinado patrón de saltos melódicos en cada voz y que sería interesante completar de forma correcta para buscar una mayor naturalidad en las líneas melódicas de la pieza a la hora de completarla. Para implementarlo se ha incluido en el módulo una asignación de acordes a tiempo con su propia detección de errores (aunque estos acordes y errores no se representan en la salida). Con estos acordes se pueden analizar los saltos armónicos presentes entre las diferentes voces de la partitura y deducir la inversión del acorde. Solo se trabaja con las inversiones primera y segunda por sencillez, ya que solo se busca el enlace entre estos acordes. Dentro de las dos posibles segundas inversiones solo se tiene en cuenta a su vez la de cuarta y sexta, por ser la más usada en enlaces armónicos. Posteriormente se maximiza el enlace entre la primera y segunda inversión o entre segundas inversiones consecutivas.

Se ha descartado la detección de apoyaturas con el fin de no contemplarlas como errores por un motivo similar, al uniformizar la longitud de las figuras de la partitura, no es posible detectarlas bien, ya que una de las características de las apoyaturas es que "roban" brevemente el tiempo fuerte a una nota representativa del acorde de la armonía. Para detectar dicha brevedad sería necesario conservar la longitud original de las notas, lo que imposibilitaría una armonización correcta.

En el módulo de salida se ha incluido la representación mediante nombre del acorde y no mediante el numeral romano.

En el pipeline se ha incluido una nueva opción -M que permite activar o

desactivar este módulo de preferencias melódicas a gusto del usuario.

### 3.7.3. Pruebas

Al incluir nuevas piezas para probar los diferentes módulos se encontraron problemas con aquellas voces capaces de producir acordes en una sola voz, es decir, las de instrumentos polifónicos. Tal y como estaba diseñado inicialmente el procesador de MusicXML a hechos lógicos (pensado principalmente para música coral) se pasó por alto esta posibilidad. En la siguiente iteración se buscará un modo de identificar estos elementos en el procesador y transformarlos en un nuevo objeto de almacenamiento `VoiceChord`

Al realizar las pruebas de las nuevas funcionalidades incluídas en el módulo de armonización, se detectó que las preferencias de sextas causaban una gran carga computacional al tener que calcular una armonización a tiempo independientemente de la usada en el módulo de armonización en sí. Por esto, las reglas correspondientes a la detección y maximización de enlaces de sextas se separaron de las preferencias melódicas de carácter más general.

### 3.7.4. Resultados

Se alcanzaron los objetivos establecidos de modo satisfactorio. Los módulos de preferencias funcionan pese a que los enlaces de sextas generen mucha carga computacional.

## 3.8. Iteración 7

### 3.8.1. Planificación

En esta iteración se busca mejorar el módulo de procesado para que sea capaz de extraer más información y metadatos de la partitura original y así conseguir una salida más fiel. Además se busca incluir en este mismo módulo la auto-detección de la clave de la partitura basándose en la armadura presente. A mayores se busca un modo más conveniente de incluir voces de una determinada tesitura a la partitura. Como en otras iteraciones se busca corregir los fallos encontrados en las pruebas de la anterior. Por último, de cara a poder resolver

ejercicios de armonía en los que ya haya acordes dados en la partitura, se pretende implementar en el procesador de MusicXML a hechos lógicos el reconocimiento de acordes ya presentes.

### 3.8.2. Trabajo

Se incluyeron en el *parser* nuevos *tokens* y reglas en la gramática para poder extraer metadatos y otra información de la partitura y exportarlos a un fichero temporal usado por el módulo de salida. Los diferentes datos extraídos para cada partitura son:

- **title:** Título
- **composer:** Compositor
- **base_note:** Longitud de la nota más breve presente
- **key_name:** Nombre de la clave en la que se armonizará la pieza
- **mode:** Modo (mayor o menor)
- **last_voice:** Número que identifica cual es la última voz presente

Los dos primeros atienden a una cuestión estética, simplemente sirven para que el módulo de salida disponga de esta información y pueda completarla a la hora de reconstruir la partitura. Los tres siguientes son datos relevantes para el módulo de armonización, mientras que el último sirve para añadir voces nuevas correctamente a la partitura. Tuvieron que incluirse reglas especiales en la gramática para poder reconocer conjuntos de múltiples palabras.

El procesador extrae una nueva pieza de información de la partitura, conocida como `figure`. Figure en la forma `figure(voice,duration,beat)` es un hecho lógico que describe la duración de una figura para un pulso de una voz dada. De este modo, pese a subdividir las notas a la longitud de la más breve para el análisis, se pueden recomponer en la salida fidedignamente. Además esta nueva información puede usarse en la generación de nuevas notas en la pieza. Se han incluido reglas para reconocer los acordes presentes en la partitura y estos se plasman en el fichero de salida.

En el módulo de armonización se ha incluido un nuevo archivo similar al de acordes o tipos de compases que describe las diferentes tesituras de las voces presentes en la partitura. Este documento es ampliable al igual que los mencionados para incluir nuevos instrumentos o tesituras. Se han definido los principales tipos de voz coral (tanto masculinos como femeninos) y sus rangos de notas más frecuentes.

| Tesitura | Nota mínima | Nota máxima |
|---|---|---|
| Bajo | 40 | 64 |
| Barítono | 45 | 69 |
| Tenor | 48 | 72 |
| Contra-Tenor | 52 | 76 |
| Contralto | 53 | 77 |
| Mezzo-Soprano | 57 | 81 |
| Soprano | 60 | 84 |

Además el módulo de armonización tiene en cuenta los nuevos hechos `figure` y los utiliza para generar correctamente las secciones a completar. Es decir, si se especifica un patrón en los tramos a rellenar, las nuevas notas generadas seguirán dicho patrón. Para las voces nuevas donde no existen dichos patrones se asigna la nota más breve ya que crear patrones rítmicos es muy costoso computacionalmente se estudia la posibilidad de crearlos mediante un postprocesado en el módulo de salida en futuras iteraciones. Se ha eliminado la constante `extra_voices` al funcionar de un modo diferente la inclusión de voces nuevas en la partitura. Para este mismo módulo se corrigió el archivo de preferencias de enlace de sextas, ahora separado del archivo de preferencias melódicas. Por último se ha adaptado la salida para trabjar con un nuevo predicado `out_figure(voz, nota, duración, pulso)` que conjuga las notas con las figuras para producir un unico predicado conjunto con toda la información necesaria.

Al fijar determinados acordes en la entrada, los tiempos de armonización en los que se debería asignar un acorde, debido a como funciona la regla de asignación de estos, no asigna uno nuevo, funcionando como se espera.

Se ha definido una nueva clase `VoiceChord`, que representa un acorde realizado por un solo instrumento polifónico ya que los este tipo de acordes producían fallos en la anterior iteración. Para su correcto funcionamiento se modificó la

pequeña rutina de transformación de hechos lógicos de la salida del módulo de armonización para tener en cuenta la posibilidad de que existiesen varias notas en un mismo pulso de una voz y agregarlas en un objeto VoiceChord.

En el módulo de salida se han corregido algunos errores encontrados en las iteraciones anteriores y haciendo uso de los nuevos predicados `figure` y `out_figure` se puede reconstruir la partitura mucho mejor que antes. Además el módulo de salida tiene en cuenta y representa correctamente el nuevo elemento `VoiceChord` y la partitura además se escribe en la clave especificada o detectada por el procesador. Por último, se ha incluido una nueva funcionalidad para leer del archivo de temporal de configuracion de la partitura los metadatos de título y compositor, junto con los nombres de los instrumentos de cada pentagrama, que son asignados a cada una de las voces de la partitura de salida.

Los nuevos datos extraídos por el procesador son leídos desde el pipeline y son pasados a los subsiguientes módulos de armonización y salida respectivamente. Se ha modificado el parámetro -v del pipeline y su efecto en el resto de módulos. En vez de especificar una cantidad de voces a añadir, toma como mínimo un argumento indicando la tesitura (por nombre) o el rango de notas para las nuevas voces. El pipeline se encarga de crear, a partir de los datos de este parámetro -v un nuevo fichero temporal `extra_voices.lp` que será incluido en la llamada del módulo de armonización para que dichas nuevas voces se tengan en cuenta. Las voces se identifican mediante un número tal y como se hacía en anteriores iteraciones pero se añade un `voice_type(voz,tipo)` que da un nombre de una tesitura a cada voz (por defecto piano al ser la más extensa y común) y se añade un predicado `voice_limit_low(voz, límite)` que establece el límite de valor de nota inferior y otro `voice_limit_high(voz, límite)` que hace lo mismo para el límite superior. Estos predicados son opcionales y no aparecen si el valor correspondiente a ellos es cero o se especifica un nombre de tesitura, obviamente para que los límites puedan ser asignados correctamente, el nombre de la tesitura debe figurar en el archivo `voice_types.lp`. Se incluyó una opción -k que permite especificar manualmente la clave de la partitura mediante la letra de nota base de la escala en la que se quiere armonizar la pieza. De no ser especificada esta se calcula automáticamente, como ya sucedía con el parámetro -d usado para especificar la longitud fraccionaria de la nota en la cual se debían subdividir todas las de la partitura. Cuenta además con una opción nueva -6 que permite incluir la

preferencia de los enlaces de sextas, que pese a estar corregida, sigue siendo muy costosa por tener que realizar una armonización a tiempo independientemente del intervalo de armonización fijado, este archivo de preferencias ya se separó del de preferencias melódicas en la iteración anterior, pero fue en esta cuando se incluyó en el pipeline una opción para activarlo.

### 3.8.3. Pruebas

En el módulo de armonización se verificó que las notas generadas en los huecos a completar estuviesen comprendidas entre las de la tesitura especificada. En la salida de este módulo se pudo comprobar que no solo las figuras generadas se combinaban correctamente con las notas generadas de modo similar a prototipos anteriores en predicados `out_figure` si no que además las notas de entrada que no tenían que ser modificadas también lo hacían, facilitando la reconstrucción al módulo de salida.

La clase `VoiceChord` funcionó como se esperaba, pero durante las pruebas de la misma se encontraron problemas con las voces de instrumentos con varios pentagramas, ya que MusicXML no las organiza en diferentes voces sino en diferentes etiquetas `staff` dentro de una misma voz. De modo similar a la anterior iteración, este error se encontró al empezar a utilizar instrumentos no solo polifónicos si no con varios pentagramas asignados a un sólo instrumento, error arrastrado del enfoque inicial de utilizar la herramienta para música coral.

El módulo de salida funcionó correctamente, pero reveló que en algún punto del proceso los valores de ciertas notas con alteraciones no se estaban calculando correctamente, es decir ciertas notas con sostenido aparecían como notas sin alterar, produciendo errores en la armonía.

En el pipeline se probaron exhaustivamente las diferentes opciones incluidas, la inclusión de nuevas voces (opción -v) fue la que más problemas planteó pero se corrigieron durante este ciclo de pruebas. La opción -k no presentó mayor complicación y funcionó correctamente desde el principio, modificando la clave de armonización a la especificada en la opción o fallando si el valor no era correcto. La nueva opción -6 para activar las preferencias referentes a los enlaces de sextas funcionó correctamente pese a que su uso producía una gran carga computacional y ralentizaba demasiado el proceso de completado de partituras incluso en tramos pequeños.

### 3.8.4. Resultados

En término general, salvando los errores encontrados durante estas pruebas, se completaron los objetivos planteados para esta iteración y dejando un prototipo funcional. No obstante este prototipo presentaba un gran problema y es que al pedirle completar voces enteras especificando una tesitura, tardaba demasiado en ofrecer soluciones. Manipulando el valor del *timeout* y con diferentes métodos de depuración del módulo ASP se comprobó que realmente parecía quedarse colgado buscando soluciones. Este resultado era inaceptable así que se buscó una refactorización del código que produjese buenos resultados temporales.

Se llegó a la conclusión de que había dos factores ralentizando el proceso:

- La armonización se realizaba junto con el completado de voces y espacios en blanco, con lo cual ASP debía generar todas las posibles combinaciones de notas para todas las posibles armonizaciones.
- La generación de notas se calculaba para cualquier rango y después se restringía para la tesitura correcta en vez de generar notas entre los límites de la tesitura.

## 3.9. Iteración 8

### 3.9.1. Planificación

La última iteración que cierra el ciclo de desarrollo busca corregir los problemas encontrados durante las pruebas de la anterior iteración, al ofrecer unos resultados relativamente malos en cuanto a tiempo de ejecución al incorporar secciones a completar en la partitura. Además tiene como objetivo incluir la posibilidad del uso de ficheros de configuración que alteren el comportamiento del módulo de armonización ajustando los diferentes pesos de las preferencias a maximizar o minimizar. Por último, y para comodidad del usuario es necesario mostrar solo los N mejores resultados, permitiendo decidir cuantos resultados desea obtener como máximo.

### 3.9.2. Trabajo

Se tomó la decisión de dividir el módulo de armonización en dos sub-módulos ASP:

- **Armonización:** Busca fijar una armonización ofreciendo al usuario diferentes soluciones ordenadas según unos valores de optimización.
- **Completado:** Su trabajo será, una vez establecida la armonización deseada, completar la partitura de modo similar a como se realizaba antes.

Se realizaron cambios menores en ambas partes para adecuarlas a sus nuevas tareas, se eliminaron muchos componentes de salida y ciertas reglas de generación de predicados ahora inútiles en la asignación de acordes y se revisó el módulo de generación de notas para eliminar todo aquello referente a la asignación de acordes. Además en el módulo de generación de notas se restringió la generación de las mismas a aquellas posibles dentro de la tesitura de la voz.

Se diferenciaron los predicados `freebeat`, generados por el procesador y por tanto con un `figure` relacionado, de los pulsos a rellenar de las voces nuevas. Estos `newvoicebeat(voz, pulso)` se utilizan para realizar la asignación de los `newvoicefigure(voz, nota, duración, pulso)`, de modo que ya se cree la figura de salida desde el principio. No obstante para realizar las restricciones correspondientes, a partir de estos `newvoicefigure` se infieren después predicados como `ex_note` o `ex_octave` para poder ser analizados al igual que las nuevas notas asignadas a los `freebeat`. Esto atiende a una cuestión de claridad para evitar confusiones a la hora de analizar los resultados.

Para posibilitar la configuración de pesos y orden de optimización en los diferentes módulos y archivos de preferencias se cambiaron los valores constantes de las reglas de optimización por nombres de valores especificados en cada uno de los ficheros para que sirvan de valores por defecto. Investigando sobre la precedencia de los valores asignados a constantes en ASP se descubrió que no se puede redefinir valores constantes y que el primer valor que toma es el usado. Esto se aplica a ficheros de configuración y los parámetros pasados por línea de comandos. Teniendo esto en mente, se creó un fichero de configuración `sample.lp` en la nueva carpeta pref, destinada a almacenar los diferentes ficheros de configuración de preferencias. Si este fichero se incluye en la llamada a clingo antes que cualquier otro fichero, los valores definidos en él serán los que se usen en el proceso

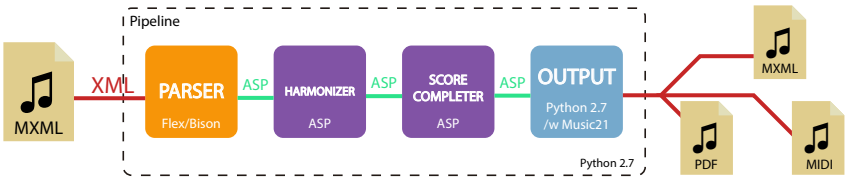


Figura 3.5: Diagrama de la arquitectura final del sistema

de armonización y completado. Si alguno de los valores se borra en este fichero, se usará el valor por defecto. Además para poder trabajar con los módulos de preferencias opcionales que también contienen pesos y orden de optimización, se modificó el orden en el que estos ficheros se incluyen en la llamada a clingo.

Debido a estos cambios se modificaron las clases de almacenamiento para reflejar los resultados únicamente armonización. De modo paralelo a las clases de almacenamiento `ClaspResult` y `HaspSolution` se crearon `ClaspChords` y `ChordSolution`. `ClaspChords` Realiza la transformación de los predicados `chord` de cada una de las soluciones de la salida del módulo de armonización a objetos `Chord` y los agrega en un objeto `ChordSolution` que a su vez se almacenan en un vector de soluciones, cada solución además contiene sus valores de optimización. De este modo mediante una representación textual de `ClaspChords` se pueden visualizar las soluciones calculadas y ofrecérselas de modo amigable al usuario.

Se detectó que el error de valores de las notas alteradas procedía de un problema en el procesador de MusicXML a hechos lógicos y se corrigió. El error se debía a una mala identificación de los valores que podía tomar la etiqueta `alteration`. Se revisó al mismo tiempo este módulo para corregir el error de los instrumentos con varios pentagramas, haciendo uso de un valor modificador del identificador de la voz que varía al ir encontrando las diferentes etiquetas `staff` para cada voz. Al estar entrelazadas las notas de cada uno de los pentagramas en una misma voz es necesario llevar un acumulador de los pulsos que ha realizado cada voz antes de cambiar de `staff` para poder compensarlo correctamente.

De modo ilustrativo se incluye la gramática final usada en el procesador de MusicXML a hechos lógicos.

En el pipeline se incluyeron dos nuevas opciones. La opción -O permite establecer el número máximo de soluciones deseadas, este valor se pasa a la llamada del constructor de `ClaspResult` y este ya se encarga de cribar la cantidad de

resultados ofrecidos al usuario en base a este número, dado que el único resultado que es deseable cribar por la gran cantidad de opciones presentes es el del módulo de completado de partituras, esta opción solo criba los resultados de dicho módulo. La opción -c por otra parte permite pasarle a los módulos de armonización y completado de partitura un fichero de configuración de pesos y orden de optimización para las diferentes preferencias. En este módulo se añadió un paso nuevo, ya que ahora hay dos sub-módulos ASP. Cuando se han calculado las diferentes armonizaciones estas son mostradas al usuario para que escoja la que desea utilizar, si el usuario no escoge ninguna se escoge por defecto la última, ya que se supone mejor que las demás al estar ordenadas por los valores de optimización. Con esta solución seleccionada se crea un fichero temporal que incluye la armonización seleccionada en forma de predicados `chord`. Después de esto se llama al módulo de completado de partituras incluyendo el anteriormente mencionado fichero con los acordes y se procede al completado de la partitura de modo similar al de anteriores prototipos. La opción -t que controla el tiempo límite de búsqueda limita únicamente el tiempo de ejecución del módulo de completado de partituras.

### 3.9.3. Pruebas

Las pruebas de esta iteración se centraron en observar los tiempos de ejecución cuando aparecían espacios para rellenar o nuevas voces, así como en comprobar que las nuevas funcionalidades (el filtrado de los N mejores óptimos, la inclusión de ficheros de configuración de los pesos de los parámetros y la nueva secuencia de ejecución del pipeline) funcionasen correctamente. Por supuesto se revisó que los errores de la anterior iteración corregidos en esta estuviesen realmente corregidos.

Se utilizó una nueva partitura para las pruebas de esta iteración, que no reveló mayor problema en ninguno de los módulos.

Las nuevas opciones incluidas en el pipeline funcionaron correctamente, restringiendo los valores incorrectos de forma adecuada y produciendo los resultados esperados.

Por último se verificó que los errores producidos por el procesador de MusicXML a hechos lógicos (tanto las voces con múltiples pentagramas asignados y las notas alteradas no identificadas correctamente) no se repitiesen tras aplicar los cambios de esta iteración.

### 3.9.4. Resultados

Se completaron correctamente todos los objetivos propuestos para esta iteración, solucionando el grave problema de tiempo de ejecución. Esta iteración cierró satisfactoriamente el ciclo de desarrollo y dejó paso a la evaluación de la herramienta en términos cualitativos y cuantitativos.

# Capítulo 4

# Evaluación

En este capítulo se van a exponer los resultados de la evaluación del sistema por diferentes usuarios relacionados con las dos materias relativas al proyecto Música e Informática. Cada uno de los expertos aportará una pieza de su elección y propondrá una modificación a la misma para ver como se comporta el sistema cuando tenga que completar la partitura escogida. Los expertos que formaron parte de la evaluación fueron:

| Nombre | C. Musicales | C.Informáticos | Pieza Escogida |
|---|---|---|---|
| Experto A | Medios | Altos | Death by Glamour |
| Experto B | Altos | Bajos | Menuet |
| Experto C | Altos | Altos | Greensleves |
| Experto D | Bajos | Medios | Joy to the World |

Además se incluye una breve explicación de cada una de las piezas, los motivos por los que fueron propuestas así como la modificación a realizar en cada una de ellas.

- **Death by Glamour:** Compuesta por Toby Fox para la banda sonora del videojuego "Undertale", Death by Glamour es una pieza interesante por su frenética melodía y por estar pensada para ser interpretada en un piano a cuatro manos. Se sugiere vaciar algunos tramos de diferentes voces.

- **Menuet:** Famosa pieza de Johann S. Bach, destaca por su simpleza y es interesante para ver como funciona el programa ante compases ternarios. Se sugiere añadir una voz nueva de tesitura más aguda a la presente en la pieza.

- **Greensleves:** Supuestamente compuesta por Enrique VIII, esta archiconocida partitura presenta una polifonía coral a cuatro voces, ideal para comprobar las capacidades de armonización del sistema. Se sugiere eliminar secciones grandes de la voz solista y ver cómo la completa.
- **Joy to the World:** Conocido villancico, sería interesante escuchar una reinterpretación de la voz más grave para la pieza, ya sea completando secciones o bien añadiendo una voz de bajo.

# Capítulo 5

# Conclusiones

El trabajo realizado...

# Bibliografía


[1] E. Herrera, *Teoría Musical Y Armonía Moderna - Volumen 1*, 1st ed. Antoni Bosch, 2015.

[2] MIDI Manufacturers Association, "Midi 1.0 detailed specification."

[3] G. Boenn, M. Brain, M. De Vos, and J. Ffitch, "Automatic music composition using answer set programming," *Theory and Practice of Logic Programming*, vol. 11, no. 2-3, pp. 397–427, 2011.

[4] ——, "Anton: Composing logic and logic composing," in *Logic Programming and Nonmonotonic Reasoning*, ser. Lecture Notes in Computer Science, E. Erdem, F. Lin, and T. Schaub, Eds. Springer Berlin Heidelberg, 2006, vol. 5753, pp. 542–547.

[5] A. Moroni, J. Manzolli, and R. Gudwin, "Vox populi: An interactive evolutionary system for algorithmic music composition," *Leonardo Music Journal*, vol. 10, pp. 49–45, 2000.

[6] D. Cope, *Experiments in Musical Intelligence*, 2nd ed. A-R Editions, 1996.

[7] V. Tran, "Music composition using artificial intelligence," 2009.

[8] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[9] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," in *The Logic Programming Paradigm*, ser. Artificial Intelligence, K. Apt, V. Marek, M. Truszczynski, and D. Warren, Eds. Springer Berlin Heidelberg, 1999, pp. 375–398. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-60085-2_17

[10] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 241–273, 1999. [Online]. Available:

# Apéndices

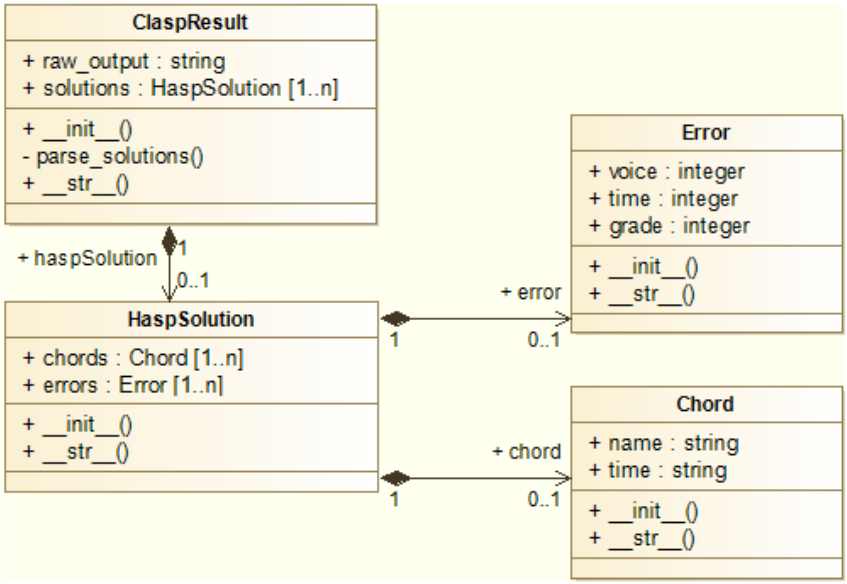


Figura 1: Diagrama de clases de almacenamiento de la Iteración 3

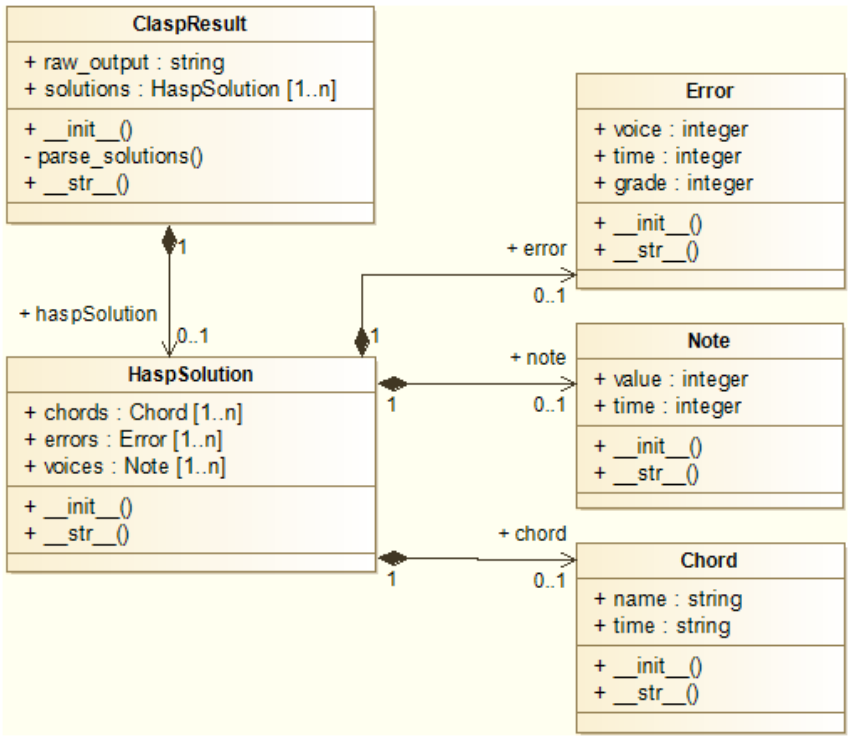


Figura 2: Diagrama de clases de almacenamiento de las iteraciones 4, 5 y 6