



UNIVERSIDADE DA CORUÑA

FACULTY OF COMPUTER SCIENCE

TECHNICAL REPORT

Haspie: A Musical Harmonization Tool powered by Answer Set Programming

Authors: Martín Prieto, Rodrigo
Cabalar Fernández, José Pedro

A Coruña, March 28, 2017.

Abstract

Haspie is a musical harmonization and composition assistant based on Problem Solving and Constraint Logic through the use of Answer Set Programming. The tool takes scores in MusicXML format and annotates them with the best possible harmonization. If specified, it is also able to complete intentionally blank sections and create brand new parts of the score. At its current state, the tool is mostly functional, but has limitations. Despite this fact, it serves as a Proof of Concept of one of the many real world applications that Answer Set Programming can achieve to solve.

Keywords:

- ✓ Logic Programming
- ✓ Knowledge Representation
- ✓ Answer Set Programming
- ✓ Constraint Solving with Preferences
- ✓ Computer Music
- ✓ Harmony
- ✓ MusicXML

Contents

	Página
1 Introduction	1
2 Background	3
2.1 State of the Art	3
2.1.1 Answer Set Programming	3
2.1.2 Formatos	5
2.1.2.1 MusicXML	5
2.1.2.2 LilyPond	6
2.1.2.3 MIDI	6
2.1.3 Software	7
2.1.3.1 Herramientas	7
2.1.3.2 Composición Asistida	7
2.1.3.3 Sistemas Inteligentes	8
3 Trabajo Desarrollado	11
3.1 Metodología	11
3.2 Planificación	12
3.3 Desarrollo	14
3.3.1 Entrada	14
3.3.2 Núcleo ASP	16
3.3.3 Módulos de Preferencias	20
3.3.4 Ficheros de Configuración	21
3.3.5 Clases de Almacenamiento	21
3.3.6 Salida	23
3.4 Iteraciones	24
3.4.1 Iteración 1 – 28 Sept 2015/11 Oct 2015	24

3.4.2	Iteración 2 – 12 Oct 2015/25 Oct 2015	25
3.4.3	Iteración 3 – 26 Oct 2015/30 Oct 2015	26
3.4.4	Iteración 4 – 2 Nov 2015/17 Nov 2015	27
3.4.5	Iteración 5 – 25 Nov 2015/6 Dic 2015	27
3.4.6	Iteración 6 – 7 Dic 2015/20 Dic 2015	29
3.4.7	Iteración 7 – 21 Dic 2015/3 Ene 2016	30
3.4.8	Iteración 8 – 11 Ene 2016/24 Ene 2016	32
4	Evaluación	35
4.1	Comparativa	40
4.2	Problemas conocidos	41
5	Conclusiones	43
5.1	Trabajo Futuro	45
5.1.1	Estética e Interfaz	46
5.1.2	Procesado y Armonización	46
5.1.3	Modulación	47
5.1.4	Publicación	47
A	Diagramas	49
B	Análisis de MusicXML	55
C	Partituras	59
D	Instalación	69
E	Manual de uso	71
	Bibliography	79

List of Figures

Figura	Página
2.1 Ejemplo de nota representada en MusicXML	5
2.2 Ejemplo de una pequeña pieza musical en lilypond	6
3.1 Diagrama que muestra la secuencia de tareas	13
3.2 Arquitectura del sistema	14
3.3 Una pieza simple transformada a hechos lógicos	16
3.4 Acordes anotados sobre una pieza sencilla y los hechos lógicos correspon- dientes a esos acordes	18
3.5 Relación de tesituras corales y sus límites inferior y superior	19
3.6 Partitura armonizada con un tiempo incompleto y resultado del comple- tado	20
3.7 Partitura de ejemplo armonizada y con notas de paso coloreadas en azul	24
3.8 Diagrama del planteamiento inicial de la arquitectura del sistema	25
4.1 Comienzo de Greensleeves armonizado	36
4.2 Compás de Greensleeves completado sin y con preferencias melódicas activadas	37
4.3 Menuet con una voz de Bajo adicional (sin preferencias melódicas) . . .	37
4.4 Comienzo de Joy to the World armonizado	37
4.5 La salida de Joy to the World produce un error de clave en la voz de la Soprano	38
4.6 Comienzo de Twinkle Twinkle Little Star armonizado	39
4.7 Compás de Twinkle Twinkle Little Star completado sin y con preferen- cias melódicas activadas	39
A.1 Diagrama de clases de almacenamiento de la Iteración 3	50

A.2	Diagrama de clases de almacenamiento de la Iteración 4	51
A.3	Diagrama de clases de almacenamiento de la Iteración 5	51
A.4	Diagrama de clases de almacenamiento de las Iteraciones 6, 7 y 8	52
A.5	Diagrama de clases de almacenamiento del módulo de armonización de la iteración 8	53
C.1	Partitura de Greensleeves, por Enrique VIII	60
C.2	Partitura de Menuet, por Johann S. Bach	61
C.3	Partitura de Joy to the World, por George F. Handel	62
C.4	Partitura de Silent Night (Noche de Paz), por Franz X. Gruber	63
C.5	Partitura de la pieza popular infantil Twinkle Twinkle Little Star (Brilla Brilla Estrellita)	64
C.6	Partitura de ejemplo con una serie de acordes sencillos	65
C.7	Partitura de ejemplo con algunos silencios reales y otros invisibles	66
C.8	Partitura de ejemplo con acordes definidos por la quinta para ser com- pletada.	67
E.1	La información relevante de la partitura se muestra al comenzar la ar- monización	73
E.2	Se pide al usuario que fije la armonización a utilizar de entre varias posibles	74
E.3	Seleccionando una figura podemos ver sus propiedades, entre otras la casilla de visibilidad	76
E.4	Desmarcando la casilla de visibilidad, la figura pasa a un color más claro, indicando que no es visible	76
E.5	Se pide al usuario que seleccione la solución que más le guste entre varias posibles	77

Chapter 1

Introduction

Music Theory learning has been stuck in the same old-fashioned methods and systems for years. These methods are based on exercises and repetition, meant to train the ear and become fluent in composition or in solving the proposed exercises. Harmony learning is one of the most important steps in music, since being able to analyse scores in this way is vital for its comprehension, later interpretation and further development. Haspie aims to help Harmony students achieve a better understanding of the matter and lets them experiment earlier with composition from an harmonic point of view.

Constraint Logic fits well with the problem since the harmony rule set used in the most basic levels taught in music schools hasn't changed since the origins of the subject, it's a definite set of rules, not very big and quite strict. Simply translating this rule set to ASP constraints and being able to extract the facts and knowledge from any score, the tool is able to detect any mistake and propose solutions, as well as filling in any blank sections of the score to create, in the end the harmony of the piece.

The great advantage that Answer Set Programming provides is that there is no need to create nor optimise a solution searching algorithm, since it only needs these harmony rules. Answer Set Programming not only offers this simplicity and power, but also flexibility since a change in the rules or in the optimization configuration can lead to very different results and adapt better to the composition style of the user.

Chapter 2

Background

2.1 State of the Art

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 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* [4] (ASP de ahora en adelante) basadas en modelos estables de Gelfond & Lifschitz [5] y lógica no-monótona. ASP es un lenguaje

de programación declarativa orientado a problemas de búsqueda difíciles, principalmente NP-complejos. ASP ha demostrado ser particularmente útil en aplicaciones donde es necesaria la representación del conocimiento y el razonamiento a partir de dicho conocimiento. En ASP, los problemas de búsqueda se reducen al cómputo de modelos estables, usándose programas conocidos como *solvers* para realizar la búsqueda de las soluciones. El lenguaje de entrada de ASP es una variante enriquecida del lenguaje prolog. Para traducir el lenguaje de entrada a reglas de programación declarativa convencional y poder construir los modelos estables subyacentes del problema, ASP cuenta con *grounders* que se encargan de esta transformación.

La implementación de un programa que busque soluciones a un problema concreto pasa por la definición de las reglas de ese problema de modo general, mientras que la definición del problema concreto a solucionar se especifica mediante hechos lógicos que, en combinación con las reglas generales del problema generan todas las posibles soluciones del mismo. Finalmente, estas soluciones se podan mediante restricciones lógicas y/o restricciones de cardinalidad. Este proceso constituye la metodología general del desarrollo en ASP, conocida como *generate and test*

Las herramientas desarrolladas por el grupo Potassco¹ incluyen, entre otras, un *grounders* (Gringo) y un *solver* (Clasp), junto con una herramienta única que combina ambos llamada Clingo.

- **Gringo** es el *grounders* de la suite. Se encarga de transformar reglas generales a reglas concretas y transforma el problema planteado a un lenguaje entendible por el *solver*.
- **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 *grounders*.
- **Clingo** es una herramienta combinada formada por clasp y gringo. Realiza el procesamiento completo de un problema planteado en ASP y permite indicar algunas preferencias adicionales.

¹<http://potassco.sourceforge.net/>

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 etiquetas anidadas de XML para agrupar los diferentes bloques de información de una pieza, como las voces, los compases o la información individual de cada nota (Ver Figura 2.1). 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 acepta como entrada o salida.

```
<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>
```

Figure 2.1: Ejemplo de nota representada en MusicXML

2.1.2.2 LilyPond

LilyPond² es un conjunto formado por el software y el formato de fichero homónimos. LilyPond como formato usa su propio lenguaje de marcado. Las etiquetas de LilyPond se parecen más a las usadas 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 (Ver Figura 2.2). Es un formato ligero pensado para poder ser editado a mano, aunque la mayoría del software musical actual lo soporta como entrada y salida.

El software del mismo nombre es un programa de grabado musical (tipografía musical o edición de partituras), pensado para producir partituras de alta calidad. Lleva la estética de la música tipografiada de la forma tradicional a las partituras impresas mediante ordenador. LilyPond es software libre y forma parte del Proyecto GNU.

```
\version "2.14.1"
{
    <c' d'' b''>8. ~ <c' d'' b''>8
}
```

Figure 2.2: Ejemplo de una pequeña pieza musical en lilypond

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 para sincronizar entre dispositivos.

²<http://www.lilypond.org/>

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

- **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.
- **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.

- **Muscore 2** es un editor *WYSIWYG* capaz de exportar a varios formatos de representación musical digital, tales como MIDI, LilyPond o MusicXML.
 - **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.
 - **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

Aunque la música puede modelarse de forma matemática, con reglas estrictas que derivan en algoritmos de composición, también 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

- **EMI y Emily Howell** Desarrollado por David Cope [1], 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.
 - **ANTON** [2] 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.
-

-
- **Vox Populi** [6] 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.
 - **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.
 - **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 rítmicos.
-

Trabajo Desarrollado

3.1 Metodología

ES difícil hablar de una planificación del proyecto como tal ya que por sus particularidades se han amalgamado características de varias metodologías. Lo que sí está claro es que el proyecto ha seguido una metodología ágil debido a su carácter de desarrollo evolutivo y orientado a la creación de prototipos funcionales en cada iteración. Se podría considerar que está a medio camino entre SCRUM y un ciclo de desarrollo en espiral con prototipado, ya que ha sido un proceso iterativo. Sin embargo, no es correcto decir que se ha utilizado SCRUM como metodología al no ser fácil repartir las tareas de cada iteración ni tener con quién hacerlo por ser un trabajo desarrollado por un único alumno junto con su director.

La similitud con el ciclo de desarrollo en espiral reside en que en cada iteración se pasan por los mismo puntos, revisando y modificando cada uno de los componentes para llegar al prototipo objetivo. Tras el trabajo, dicho prototipo es revisado en la reunión con el “cliente”, que en este caso es el director del proyecto, revisando la funcionalidad implementada y dirigiendo los siguientes pasos.

Pese a las dificultades, disponiendo de una agenda de producto similar a la de SCRUM a modo de requisitos, se han podido trazar una serie de iteraciones. Cada iteración cuenta con su agenda de iteración en la cual se detallan los requisitos del prototipo a crear en ese paso haciendo evolucionar cada uno de los componentes. Se habla de hacer evolucionar los componentes y no de incrementarlos porque, pese a que cada iteración ha procurado incrementar la funcionalidad total del

sistema, algunos módulos del mismo han sido refactorizados¹ en determinados pasos.

En cuanto a la planificación de pruebas del sistema, si bien se ha contado con diferentes entradas con las que probar cada uno de los módulos que se han ido desarrollando junto con el proyecto, no se puede decir que el desarrollo de una herramienta de estas características pueda ser dirigido por la prueba, ya que las soluciones halladas mediante *Answer Set Programming* no pueden ser establecidas de antemano debido a su carácter no determinista.

Se estableció que la duración de cada iteración sería, en promedio, de semana y media. Esto equivale a unas 40 horas de trabajo por iteración, aunque se decidió permitir la flexibilización de las iteraciones ante imprevistos en el desarrollo. Esto no se corresponde con una metodología SCRUM típica en la que habría que desplazar tareas a las siguientes iteraciones, pero al ser solo una persona encargada de todo el trabajo, se decidió dotar a la metodología de cierta flexibilidad para no arrastrar trabajo. No obstante, esta flexibilización permite también ganar tiempo, ya que de completarse el trabajo antes de lo previsto se podría acortar el tiempo dedicado a alguna iteración.

Gracias a esta flexibilidad se logró depender solo de completar los objetivos de las diferentes agendas, logrando así independencia del factor tiempo, que ya venía fijado por los plazos de entrega propios del proyecto.

3.2 Planificación

Debido a estos factores derivados de lo particular de desarrollar un sistema basado en *Answer Set Programming*, se carece de una planificación inicial cerrada y completa que marque el tiempo estimado para cada una de las tareas en forma del tradicional diagrama de Gantt. No obstante, a modo de registro, sí se ofrece un diagrama similar que muestra el tiempo invertido en cada una de las iteraciones (Ver Figura 3.1).

Dado este diagrama, es relativamente sencillo realizar una primera estimación del coste del proyecto. Hay que tener en cuenta, no obstante, horas extra de desarrollo que se incluyen en el diagrama y en el presupuesto ya que el trabajo no empezó de cero, al haberse realizado dos prácticas durante los años tres y cuatro

¹Es decir, se ha cambiado su diseño, sin alterar su propósito funcional

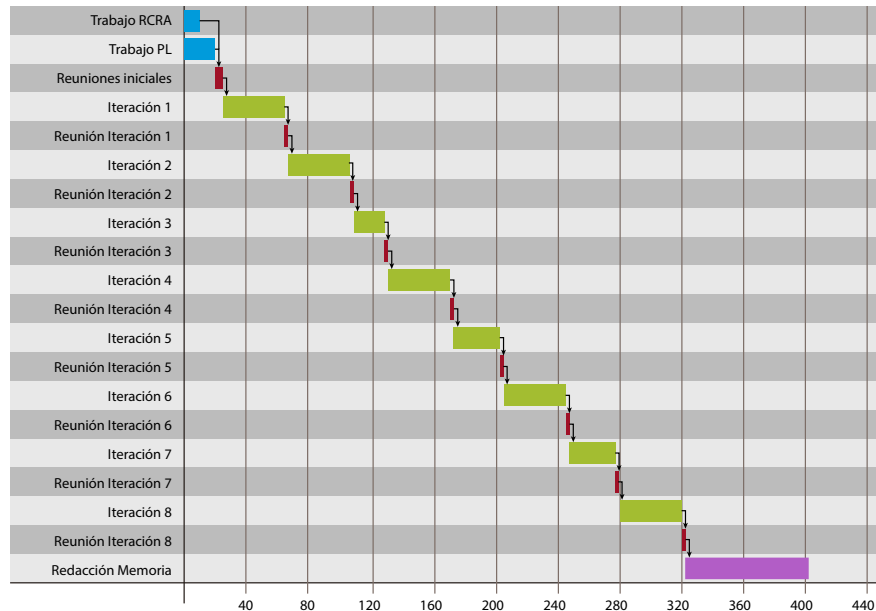


Figure 3.1: Diagrama que muestra la secuencia de tareas

del grado que se han reutilizado en menor o mayor medida para comenzar el desarrollo del presente trabajo. En la asignatura de “Representación del Conocimiento y Razonamiento Automático” se desarrolló una pequeña herramienta en *Answer Programming* capaz de completar y componer pequeñas piezas musicales en forma de *Canon*, mientras que en la asignatura de “Procesamiento de Lenguajes” se creó una primera versión del procesador de MusicXML a hechos lógicos. Se ha realizado una estimación de que, de no haber trabajado en la práctica de RCRA, se habrían necesitado unas 10 horas adicionales en los primeros pasos del desarrollo del módulo de armonización de este proyecto, mientras que de no haber realizado la práctica de PL se habrían necesitado unas 20 horas adicionales.

Sumado a 40 horas por iteración y contando con 8 iteraciones, arrojan un total de 350 horas de desarrollo. Las reuniones de final de iteración duraron una hora escasa cada una, pero al haber habido más de una por iteración y que las primeras reuniones fueron más extensas al tener que definir y planificar el proyecto se estima que el proyecto contó con unas 12 horas de reuniones para las que hay que sumar el coste por hora del tiempo del director. Puede verse una estimación del coste del proyecto en la siguiente tabla:

Perfil	Coste/Hora	Horas	Total
Proyectando	5.5€	362	1991€
Director	9€	12	108€
Total			2099€

El coste de producción del proyecto es esencialmente el mismo que el de desarrollo, ya que las licencias de todo el software utilizado son gratuitas y no requiere ningún tipo de hardware adicional para funcionar. Incluso el editor de partituras que se recomienda utilizar junto con el programa (MuseScore2) es libre y de código abierto.

3.3 Desarrollo

La arquitectura de `haspie` (Ver Figura 3.2) es un sencillo pipeline escrito en Python con un único camino que se ejecuta siempre que se llama a la herramienta, este pipeline se encarga de llamar a cada uno de los submódulos, enmascarando así el comportamiento interno de cada uno de ellos. Este pipeline además permite en su llamada a través de línea de comandos especificar diferentes opciones que se pasan después a cada submódulo de forma individual, estas opciones pueden verse en detalle en el anexo E.

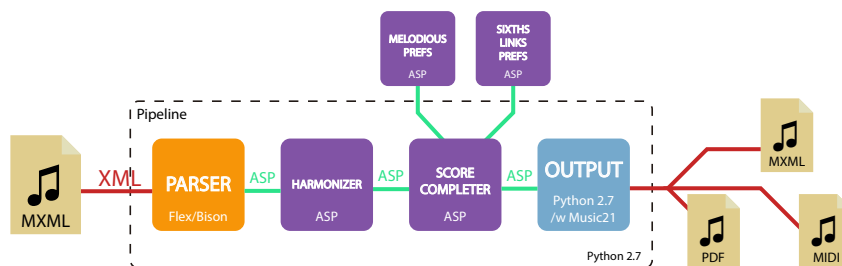


Figure 3.2: Arquitectura del sistema

3.3.1 Entrada

La entrada del sistema se realiza en formato MusicXML y el pipeline pasa la ruta del archivo especificado al módulo de entrada: un *parser* escrito en C junto con las librerías Flex y Bison que se encarga de transformar los elementos de la

partitura a hechos lógicos en Answer Set Programming. Este conversor no solo traduce las figuras y silencios a hechos sino que además:

- Subdivide todas las figuras de la partitura a la duración de la nota más breve de la pieza. Esta división es necesaria para que los módulos ASP puedan encajar correctamente todos los sonidos que se producen en cada tiempo de las diferentes voces de la pieza.
- Crea hechos lógicos adicionales **figure** que indican la duración original de cada una de las figuras con el fin de conservar esta información para que la pieza pueda ser reconstruida tras su procesado.
- Identifica las definiciones de la métrica de los compases que aparecen a lo largo de la partitura y convertirlos a los hechos lógicos correspondientes de modo que se pueda identificar correctamente el tipo de subdivisión de la partitura.
- Interpreta los nombres de los diferentes instrumentos de cada parte de la pieza y asigna hechos que vinculan cada una de las voces a un nombre instrumento o tipo de voz para poder detectar su tesitura.
- Lee la armadura de la partitura y determina la tonalidad y modo más probables correspondientes a la misma.
- Detecta el nombre y el autor de la partitura para poder representarlos en la salida.

Aquellos datos procesados que no se convierten en hechos lógicos y se añaden al fichero ASP que servirá de entrada al núcleo del programa se anotan en un fichero de configuración temporal que sirve a los distintos módulos del sistema para su funcionamiento.

El *parser* funciona de un modo convencional, identificando las etiquetas relevantes que contienen la información necesaria para crear los hechos lógicos y almacenándolos en tipos de datos propio que se corresponden con los elementos de la partitura tales como tipos de compás, anotaciones de acordes, nombres de instrumentos, nuevas voces, notas o silencios. A su vez, estos datos se almacenan en diferentes estructuras de cola (una para cada tipo de dato) mientras que se va procesando toda la pieza para calcular la longitud de la figura más breve.

Una vez se ha terminado de leer la partitura, se extraen los datos de cada una de las colas y se procesan junto con la información general extraída al procesar toda la pieza para escribir los diferentes hechos lógicos (Ver Figura 3.3) en el fichero correspondiente o los metadatos y otros valores de salida en el fichero de configuración.

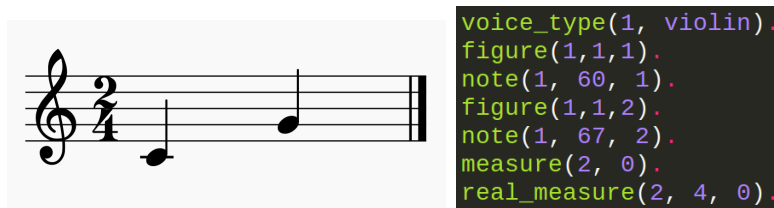


Figure 3.3: Una pieza simple transformada a hechos lógicos

Además cuenta con parámetros en su llamada que modifican la salida producida:

- **-k** Indica manualmente la clave en la que se armonizará la pieza en vez de detectarse automáticamente.
- **-s** Especifica la cantidad de figuras consecutivas que se tienen en cuenta para la armonización.
- **-o** Indica el nombre del fichero de hechos lógicos de salida.

3.3.2 Núcleo ASP

Los hechos lógicos creados por el *parser* son la entrada directa del armonizador, una de las mitades del núcleo de procesado ASP de **haspie**. Este módulo utiliza los hechos lógicos para expandir las reglas generales con las que cuenta, infiere nuevos predicados intermedios y asigna acordes de entre los posibles a cada tiempo de armonización especificado.

```
1 { chord(HT,C) : pos_chord(C) } 1 :- htime(HT).
```

Los posibles acordes se especifican en un archivo denominado **major_chords** o **minor_chords** según el modo en el que se esté realizando la armonización. Los

acordes contemplados inicialmente por la herramienta son los acordes fundamentales de tres notas de la tonalidad, aunque se incluyó el de Dominante séptima² para dotar al sistema de mayor naturalidad.

Para ello, lo más importante es la conversión de los valores de las notas a grados de la escala junto con su octava, lo cual es posible conociendo la tonalidad y el modo. Para conocer el nombre de cada nota, se resta un *offset* al valor de la misma (este *offset* depende de la tonalidad en la que se armoniza la partitura) y se divide ese valor entre 12 (ya que son 12 posibles sonidos). Este valor representa el valor de la nota en semitonos, y conociendo el modo, puede ser convertido a un grado de la escala. La octava se calcula de modo similar, restando el *offset* anterior, pero en vez de dividir entre 12, se realiza una operación de módulo.

```
octave(V, ((N - base) / 12), T) :- note(V, N, T), N >= 0.
sem_tones(V, ((N - base) \ 12), T) :- note(V, N, T), N >= 0.
grade(V, 1, T) :- sem_tones(V, 3, T).
grade(V, 2, T) :- sem_tones(V, 5, T).
grade(V, 3, T) :- sem_tones(V, 7, T).
grade(V, 4, T) :- sem_tones(V, 8, T).
grade(V, 5, T) :- sem_tones(V, 10, T).
grade(V, 6, T) :- sem_tones(V, 0, T).
grade(V, 7, T) :- sem_tones(V, 2, T).
```

Con las notas transformadas en grados y todos los posibles acordes asignados, se cuantifican los errores cometidos, esto es, en vez de prohibir que se produzcan errores, se marcan aquellas notas que no pertenecen al acorde como error y se clasifican en errores fuertes y débiles atendiendo al tipo de tiempo del compás en el que coincide el error. El cálculo del tipo y subdivisión del compás, y por tanto de sus tiempos débiles y fuertes se realiza gracias a los hechos lógicos referentes al tipo de compás de la partitura y al intervalo de armonización de la misma. Mediante un predicado de optimización que minimiza los errores en tiempos fuertes, los acordes repetidos en tiempos de armonización consecutivos y los errores cometidos en tiempos débiles (inicialmente en ese orden de importancia) se busca la mejor armonización posible.

²Acorde de cuatro notas que incorpora una nota adicional que forma un intervalo de séptima con la nota dominante de la tonalidad

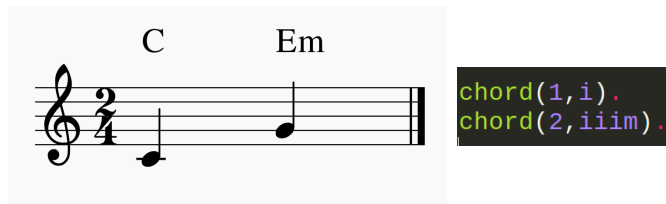


Figure 3.4: Acordes anotados sobre una pieza sencilla y los hechos lógicos correspondientes a esos acordes

De vuelta al pipeline, los mejores resultados son transformados a una representación objetual en Python (Ver sección 3.3.5) para almacenar la información de los diferentes elementos y poder representarlos de manera sencilla.

A continuación se pide al usuario que escoja un resultado de entre los mejores posibles, mostrándole la secuencia de acordes y especificando las notas erróneas, así como el grado de optimización del resultado. Una vez escogida la armonización deseada, se crea un fichero temporal con los acordes (Ver Figura 3.4) de la solución que se pasa, en conjunto con el fichero de hechos lógicos original, al módulo de completado de partituras.

Con ambos ficheros de hechos lógicos, se llama al módulo de completado de partituras, que actuará en caso de existir voces nuevas que completar o secciones en blanco que rellenar. Los intervalos a rellenar están representados por el hecho lógico especial **freebeat** que indica que ese tiempo no contiene una nota sino un hueco intencionalmente blanco. No obstante, estos hechos van acompañados de un hecho **figure** al igual que las notas o silencios, de modo que se puede especificar el patrón rítmico de la sección a completar. El módulo de completado de partituras funciona de un modo similar al de acordes, asignando nuevas notas a los espacios en los cuales se le pida que lo haga. La principal diferencia es que no cuenta con un fichero de notas posibles como el módulo de armonización con los acordes, sino que utiliza el valor numérico de la nota directamente.

```
1 { freebeatfigure(V,N,1,FB) : N=VL..VH } 1 :- freebeat(V,FB),
    voice_limit_low(V,VL), voice_limit_high(V,VH).
```

Para limitar el espacio de búsqueda, estos valores son únicamente los comprendidos entre los límites definidos por la tesitura del instrumento de la voz que se está completando. De manera parecida al fichero de definición de

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

Figure 3.5: Relación de tesituras corales y sus límites inferior y superior

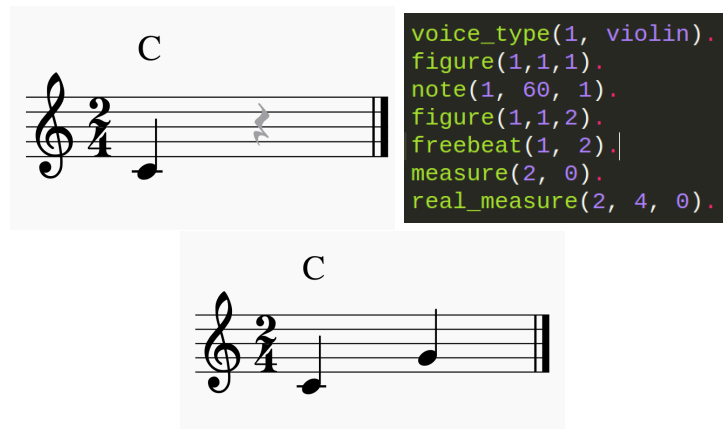
los acordes de cada modo, se han definido las tesituras de algunas de las voces corales más frecuentes (Ver Figura 3.3.2) junto con algunos instrumentos como el violín o el violonchelo, así mismo este fichero puede ser fácilmente editado por el usuario para incluir nuevas tesituras o instrumentos.

Estas nuevas notas se transforman a grados y octavas como ya se hace con las notas de entrada del módulo de armonización para poder ser cotejadas contra la armonía especificada por el módulo anterior. Del mismo modo que en el módulo de armonización se cuantifican y categorizan los errores producidos por estas nuevas notas en relación a la armonía establecida y, minimizándolos, se buscan los mejores resultados. No obstante, en este módulo sí existen restricciones de integridad:

```
octave_jump(V,B1,B2) :- ex_note(V,N1,B1), ex_note(V,N2,B2),
                        (B1+1) == B2, N2 > (N1+12), beat(B1+1).
octave_jump(V,B1,B2) :- ex_note(V,N1,B1), ex_note(V,N2,B2),
                        (B1+1) == B2, N2 < (N1-12), beat(B1+1).
:- octave_jump(_,_,_).
```

Finalmente, gracias a los hechos lógicos **figure** de la entrada, el módulo de completado es capaz de reconstruir las figuras originales.

Repitiendo el proceso anterior, el pipeline procesa la salida de clasp y la almacena en las clases de almacenamiento correspondientes. Tras esto muestra las mejores soluciones de completado al usuario y permite escoger una de ellas. Al hacerlo los objetos de almacenamiento generados se pasan al módulo de salida para su representación final. (Ver Figura 3.6)



```
voice_type(1, violin).
figure(1,1,1).
note(1, 60, 1).
figure(1,1,2).
freebeat(1, 2).
measure(2, 0).
real_measure(2, 4, 0).
```

Figure 3.6: Partitura armonizada con un tiempo incompleto y resultado del completado

3.3.3 Módulos de Preferencias

Se han especificado, además de los módulos principales ya descritos, dos submódulos adicionales que mejoran los resultados del módulo completador de partituras.

El primero es un módulo de preferencias melódicas. Sirve para compensar la ausencia de composición melódica en **haspie** creando partituras más cantables o melódicas. Contiene reglas que:

- Definen la tendencia de una voz ya existente en la partitura y permiten al completador de partituras componer de forma que se imite dicha tendencia.
- Miden los saltos melódicos entre las notas de una misma voz e intentan acortarlos haciendo que la progresión melódica sea más cantable.

Para la tendencia, se mide si en un intervalo determinado, el valor de notas consecutivas crece o decrece y mediante una regla que establece el tipo de tendencia (semejante u opuesta) al procesar múltiples voces al mismo tiempo. Por otra parte, para los saltos melódicos se crean nuevos predicados **melodic_jump** teniendo en cuenta la diferencia de los valores de notas consecutivas en una misma voz.

```
melodic_jump(V,J,B1,B2) :- out_note(V,N1,B1), out_note(V,N2,B2),
                           (B1+1) == B2, beat(B1+1), J = #abs(N1-N2).
```

El segundo módulo detecta progresiones de un determinado tipo de acordes (inversiones de cuarta y sexta). Este tipo de progresión es muy común en música coral, y por tanto, los resultados usando este módulo con piezas corales resultan mucho más realistas. Para esto se realiza una armonización a tiempo, es decir, se asigna un acorde a cada instante de la pieza y se intentan detectar estas progresiones concretas de acordes con el fin de crearlas o continuarlas si ya existen con la ayuda de las nuevas notas generadas. Es debido a esta armonización a tiempo que este módulo es tremendamente pesado computacionalmente.

3.3.4 Ficheros de Configuración

Las reglas de optimización, ya sean maximización o minimización, presentes en los módulos y submódulos ASP asignan un peso y un orden de relevancia a los predicados que se cuantifican de cara a la optimización. Este peso por defecto puede ser sobrescrito mediante ficheros de configuración. Cambiando el peso de cada hecho se altera cómo de importante es la aparición de cada uno de esos hechos, mientras que cambiando el orden de relevancia se determina cómo se realizará el proceso de optimización al completo. Se presenta un fichero de configuración de ejemplo comentado para que el usuario pueda cambiar el funcionamiento de la herramienta a su antojo, produciendo cambios drásticos en el estilo con un esfuerzo mínimo y permitiendo almacenar estas preferencias en un fichero que determine su estilo personal de composición.

3.3.5 Clases de Almacenamiento

Para poder almacenar temporalmente los resultados de los dos submódulos ASP de **haspie** se diseñaron e implementaron clases de almacenamiento que se corresponden con los diferentes hechos lógicos que los módulos ofrecen en su salida. El diagrama de clases de estos objetos de almacenamiento puede consultarse en el anexo A. Se dividen en dos jerarquías, una para los resultados del módulo de armonización y otra, considerablemente más grande para el módulo de completado de partituras. En cuanto al módulo de armonización, éste cuenta con las siguiente clases:

- **ClaspChords**: Almacena todas las soluciones del armonizador, tanto en texto plano como en un array de objetos **ChordSolution**. Posee un método

para transformar la salida de clasp a los diferentes objetos de la jerarquía.

- **ChordSolution**: Contiene dos listas ordenadas, una de objetos **Error** y otra de objetos **Chord**, y un valor de optimización que mide cómo de buena es la solución.
- **Error**: Representa una nota errónea en la partitura, especificando voz y tiempo para ubicarla en ella, así como el grado musical que produce el error.
- **Chord**: Representa un acorde en la pieza, especificando el nombre del mismo y el tiempo al que ha sido asignado.

Las clases para almacenar la información de la salida del módulo de completado de partituras siguen una estructura similar al de armonización, pero la jerarquía es algo más compleja al tener mucha más información que representar:

- **ClaspResult**: Funciona de modo muy similar a **ClaspChords**. Almacena todas las soluciones del completador de partituras, tanto en texto plano como en un array de objetos **HaspSolution**. Posee un método para transformar la salida de clasp a los diferentes objetos de la jerarquía.
 - **HaspSolution**: De forma parecida a **ChordSolution**, almacena los diferentes objetos de la solución. La principal diferencia radica en que hace uso de un diccionario de datos, siendo los índices cada una de las voces de la pieza y almacenando en la entrada de cada índice una lista ordenada de los diferentes objetos de la partitura.
 - **Error**: Funciona igual que **Error** en la jerarquía de clases del módulo de armonización. Representa una nota errónea en la partitura, especificando voz y tiempo para ubicarla en ella, así como el grado musical que produce el error.
 - **PassingNote**: Nota errónea que cumple unas características particulares, como su ubicación en un tiempo débil de la partitura. Especifica la voz y el tiempo en el que se produce, de modo similar a **Error**.
 - **Chord**: Similar a **Chord** en la jerarquía de clases del módulo de armonización. Representa un acorde en la pieza, especificando el nombre del mismo y el tiempo al que ha sido asignado.
-

- **VoiceType**: Indica el nombre asignado a una voz, es decir, el instrumento que la interpreta.
- **Note**: Especifica la voz y tiempo en los que suena la nota, así como su valor numérico y duración.
- **Rest**: Especifica la voz y tiempo en los que se produce el silencio así como su duración.
- **Measure**: Indica la métrica y cantidad de notas del tipo de compás así como el tiempo en el que aparece. Se interpreta que el cambio de tipo de compás se produce en todas las voces a la vez.
- **VoiceChord**: Representa un conjunto de notas que suenan en un mismo tiempo de una sola voz, produciendo un acorde (Por ejemplo en instrumentos polifónicos como el piano o la guitarra). Contiene un array de objetos **Note** ordenados por su valor numérico, así como la voz y el tiempo en el que se produce junto con su duración.

3.3.6 Salida

El último módulo al que llama el pipeline. Haciendo uso de los objetos de representación interna, el módulo de salida genera un fichero en el formato deseado por el usuario. Para la implementación de este módulo se ha hecho uso de la librería Music21³ desarrollada por miembros del MIT. Se traduce la representación interna de la partitura que el pipeline pasa al módulo de salida a la representación propia de Music21 para después con una simple llamada, reproducir o almacenar el resultado. En este proceso de conversión, no solo se traducen las notas, voces y anotaciones de compás al formato propio de Music21, sino que además se combinan algunos de ellos para enriquecer la salida.

- Los acordes, especificados en forma de grado de la tonalidad, se traducen al nombre y tipo de acorde correspondiente para ser anotados en la partitura.
- Los errores en tiempos fuertes se marcan en la partitura coloreando en rojo las notas que los contienen.

³<http://web.mit.edu/music21/>

- Los errores en tiempos débiles se marcan en azul en la partitura indicando que son notas de paso o adornos. (Ver Figura 3.7)

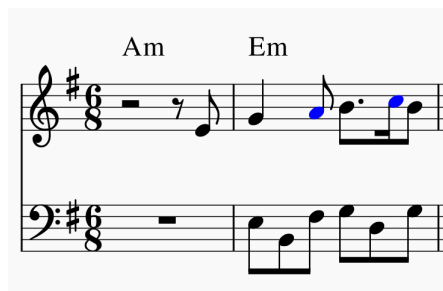


Figure 3.7: Partitura de ejemplo armonizada y con notas de paso coloreadas en azul

3.4 Iteraciones

Se detalla un breve resumen de cada una de las iteraciones de desarrollo del proyecto, indicando cuáles fueron los progresos, cambios y refactorizaciones en cada una de ellas. Junto al número de iteración se indican las fechas de inicio y final de las mismas.

3.4.1 Iteración 1 – 28 Sept 2015/11 Oct 2015

Una vez esbozada la arquitectura del sistema (Ver Figura 3.8), se analizaron y escogieron las diferentes tecnologías que se usarían en cada uno de los módulos de la herramienta. Como software de entrada para las partituras se escogió MuseScore 2, principalmente por ser *opensource* y por su sencillez. 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. MusicXML se presentó como el formato idóneo para la tarea, ya que al ser una extensión de XML, está orientado a que una máquina pueda procesarlo y crear una estructura en memoria con toda la información que necesita para poder extraer los datos de la partitura. Además, la implementación del *parser* para un lenguaje etiquetado como XML es un problema convencional y relativamente sencillo.

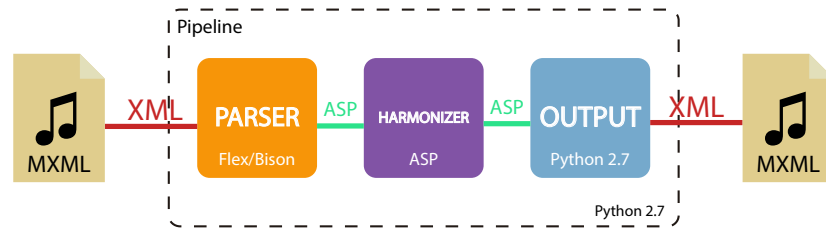


Figure 3.8: Diagrama del planteamiento inicial de la arquitectura del sistema

Para el procesador de XML a hechos 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 permitiría añadir nuevos elementos a reconocer cuando sea necesario.

Se procedió a desarrollar una versión actualizada de dicho parser. 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.

3.4.2 Iteración 2 – 12 Oct 2015/25 Oct 2015

Se modificó el procesador de MXML a ASP y se incluyó 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.

En esta primera aproximación, se crearon las primeras reglas del módulo de armonización, que realizan una asignación acorde-unidad rítmica en base a las notas presentes para un instante dado en cada voz. Se crearon dos ficheros adicionales que especifican los acordes a considerar por la herramienta, según si la armonización se realiza en modo mayor o menor.

Además se incluyó como parte de esta iteración, el diseño e implementación de

un pipeline en Python que automatiza las llamadas al procesador MXML a hechos 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.

3.4.3 Iteración 3 – 26 Oct 2015/30 Oct 2015

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 MusicXML. 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 cuál 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* conllevaron 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.

En el módulo de armonización se incluyó una nueva constante que indica la longitud del intervalo de tiempo mínimo de análisis armónico horizontal. Se modificaron, por tanto, las reglas de asignación de acordes para poder usar el intervalo especificado. Además se suavizaron las restricciones que podaban las soluciones erróneas y en vez de ello, generan 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.

Se han implementado en el pipeline, con vistas al futuro de módulo de salida, una serie de clases para procesar y almacenar los resultados de clasp y poder

devolverlos más tarde en el formato más conveniente. Concretamente, en esta iteración se implementaron las primeras versiones de `Error`, `Chord`, `HaspSolution` y `ClaspResult`.

3.4.4 Iteración 4 – 2 Nov 2015/17 Nov 2015

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 refactorizaron algunas reglas y se creó un fichero de conversiones encargado de traducir valores de notas a grados, octavas y viceversa. 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 más de una octava.

Se optó 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 `HaspSolution` y, previa transformación a la representación interna de Music21, lo representa en el formato adecuado para ser reproducido o almacenado.

En el *pipeline* se ha incluyó la opción de especificar el número de voces adicionales que deben ser añadidas y otra opción para especificar el formato de salida. Adicionalmente, se incluyó una llamada al módulo de salida en el pipeline.

3.4.5 Iteración 5 – 25 Nov 2015/6 Dic 2015

En el procesador de MusicXML a hechos lógicos ASP 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.
- **Distinción de tipos de silencios:** Diferenciar silencios completables de aquellos que deben ser respetados como silencios.

Para la métrica del compás, se implementó en el procesador la capacidad de identificar los diferentes tipos 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. El tipo de compás además es

modificado según la nota mas breve de la partitura para que no existan problemas a la hora de comparar la cantidad y el tipo de figuras del compás.

Para poder especificar en el editor de partituras silencios “verdaderos” y silencios completables se optó 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, que representan los tiempos completables de cada voz. Además se crearon reglas para identificar la subdivisión de los compases con el fin de inferir los tiempos fuertes y débiles de cada compás en la partitura. Gracias a esto se han creado predicados que matizan los diferentes errores de la partitura según el tipo de tiempo en el que 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 incluido el acorde de dominante séptima (V7) en los modos mayor y menor.

Se ha incluido una clase de almacenamiento **Rest** para diferenciarla de **Note**. Ya que las entradas del diccionario que representa las diferentes voces de la partitura puede almacenar cualquier combinación de tipos de elementos, se estableció 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 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.

3.4.6 Iteración 6 – 7 Dic 2015/20 Dic 2015

Para facilitar la entrada de silencios que representan huecos completables por el módulo ASP se cambió el enfoque, y se abandonó 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 *ad-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.

Se creó el 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 incluyó también en este sub-módulo de preferencias melódicas la detección de patrones de secuencias de sextas.

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.

3.4.7 Iteración 7 – 21 Dic 2015/3 Ene 2016

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 cuál es la última voz presente.

El procesador extrae una nueva pieza de información de la partitura, conocida como **figure**. Figure 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. Se han incluido reglas adicionales 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.

Además el módulo de armonización se añadieron reglas para tener en cuenta los nuevos hechos **figure** y utilizarlos para generar los patrones rítmicos adecuados en las secciones a completar. 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 trabajar con un nuevo hecho lógico que conjuga las notas con las figuras para producir un único hecho conjunto con toda la información necesaria.

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 hechos lógicos de la iteración, 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 en 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 configuración 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. Se incluyó una opción 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. Cuenta además con otra opción que permite incluir la preferencia de los enlaces de sextas.

Tras probar el prototipo de esta iteración 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.

3.4.8 Iteración 8 – 11 Ene 2016/24 Ene 2016

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 hechos `freebeat`, generados por el procesador y por tanto con un `figure` relacionado, de los pulsos a rellenar de las voces nuevas. Estos `newvoicebeat` se utilizan para realizar la asignación de las nuevas figuras y notas, de modo que ya se cree la figura de salida desde el principio.

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. Se investigó sobre el orden de precedencia de los valores asignados a constantes en ASP y 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 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 de la armonización y el completado de forma separada. De modo paralelo a las clases de almacenamiento `ClaspResult` y `HaspSolution` se crearon `ClaspChords` y `ChordSolution`.

Se revisaron errores producidos en la interpretación de los valores de las notas alteradas. El error se debía a una mala identificación por parte del *parser* de los valores que podía tomar la etiqueta `alteration`. Se revisó también este módulo para corregir otro error que afectaba a instrumentos de varios pentagramas, donde las diferentes partes se dividían en diferentes voces pero no se asignaba el nombre del instrumento de forma correcta a otras voces que no fuesen la primera.

Durante las pruebas finales se detectó un tipo de silencio no reconocido correctamente, este silencio usaba el símbolo de un silencio de blanca pero ocupaba todo el compás siempre, fuese cual fuese su duración. Tal y como estaba diseñado el procesador, que identificaba el tipo de figura y la subdividía cuando fuese necesario, este tipo de figuras eran irreconocibles y se tuvo que implementar un caso específico para estos silencios especiales.

En el pipeline se incluyeron dos nuevas opciones. Una para establecer el número máximo de soluciones deseadas y otra que permite pasarle a los módulos de armonización y completado de partitura el fichero de configuración deseado. Se cambió ligeramente el comportamiento del pipeline al tener que realizar las llamadas a los dos nuevos sub-módulos ASP, parando tras hallar las mejores armonizaciones y ofreciendo al usuario seleccionar la deseada y pasando el resultado al sub-módulo de completado, donde la ejecución continúa de modo similar a los anteriores prototipos. Por último se modificó la opción que permite especificar el tiempo límite de búsqueda para ser solo usada en caso de querer buscar todos los óptimos en vez de quedarse con el primero encontrado.

Chapter 4

Evaluación

EN este capítulo exponen los resultados de la evaluación del sistema. Se han seleccionado tres piezas musicales conocidas para realizar las diferentes pruebas. A continuación se describen las tres piezas y las pruebas realizadas. Para realizar una prueba de carga que establezca los límites del sistema se ha añadido una última partitura suficientemente sencilla sobre la que trabajar para poder realizar estas pruebas cómodamente sin preocuparse por la calidad de los resultados.

- **Menuet en Sol Mayor:** Famosa pieza de Johann S. Bach, destaca por su simpleza y es interesante para ver cómo funciona el programa ante compases ternarios. Se sugiere añadir una voz nueva de tesitura más grave a la presente en la pieza. (Ver Figura 4.3)
- **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 de alguna voz y ver cómo la completa. (Ver Figuras 4.1 y 4.2)
- **Joy to the World:** Conocido villancico de Georg F. Händel, sería interesante escuchar una reinterpretación de la voz más grave para la pieza, ya sea completando secciones o bien añadiendo una voz de bajo (Ver Figura 4.4).
- **Twinkle Twinkle Little Star:** Esta sencilla pieza popular es adecuada para realizar, además de pruebas similares a otras piezas, pruebas de carga

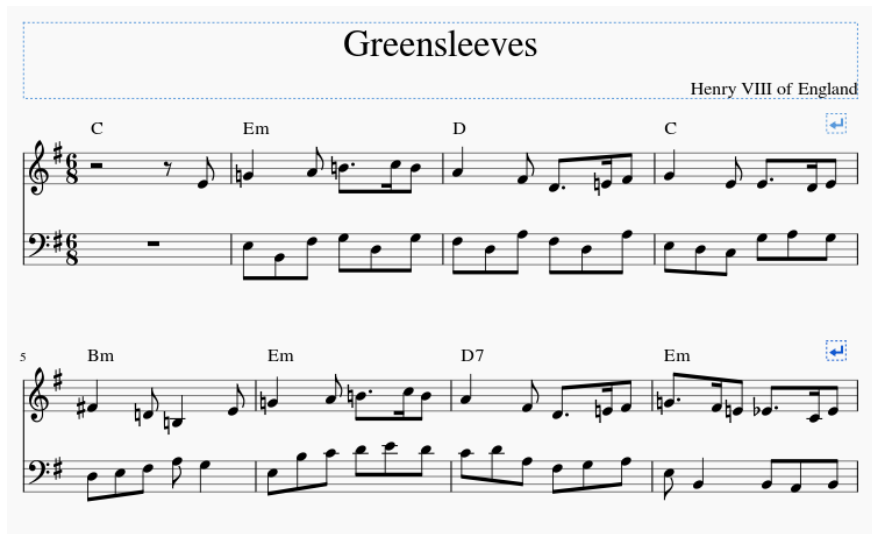


Figure 4.1: Comienzo de Greensleeves armonizado

para comprobar cómo crecen los tiempos de búsqueda y hallar los límites de la herramienta (Ver Figuras 4.6 y 4.7).

Para la evaluación se han realizado armonizaciones de todas las piezas y se han medido los tiempos y la calidad de las mismas, además se han realizado pruebas de completado de un compás en cada pieza junto con la inclusión de una nueva voz, también se han medido calidad y tiempo. Por último, con la última partitura se han realizado pruebas para comprobar hasta donde funciona bien en cuestión temporal, para ello se ha ido vaciando porcentualmente una de las voces para ser completada y después se han ido añadiendo diferentes voces. Las figuras de esta sección sirven para ilustrar algunos resultados obtenidos, se puede comparar con las partituras originales presentes en el Apéndice C Partituras.

Nótese que debido al carácter no-determinista de *Answer Set Programming* y a los tiempos de entrada y salida estos tiempos sirven para dar una idea aproximada de los tiempos de funcionamiento de la herramienta. Para suavizar los valores dispares se ha ejecutado cada medida 100 veces, se ha restado el tiempo de usuario y se ha calculado el tiempo promedio.

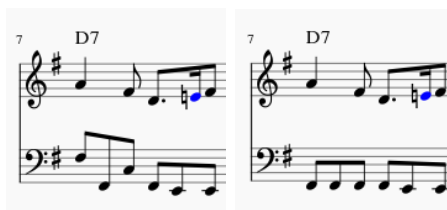


Figure 4.2: Compás de Greensleeves completado sin y con preferencias melódicas activadas



Figure 4.3: Menuet con una voz de Bajo adicional (sin preferencias melódicas)



Figure 4.4: Comienzo de Joy to the World armonizado

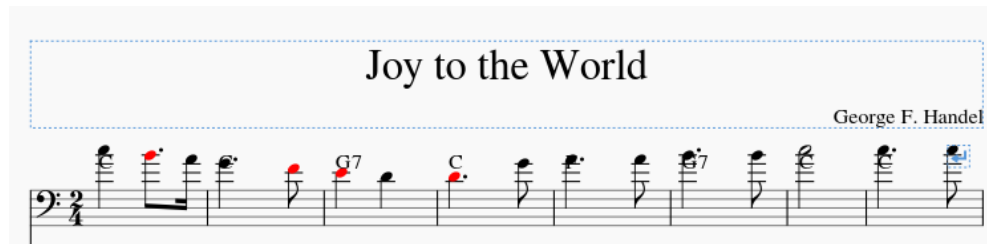


Figure 4.5: La salida de Joy to the World produce un error de clave en la voz de la Soprano

Pieza	T. Armonización	T. Compás	T. Nueva voz
Greensleeves	1.016s	1.926s	4m 49.032s
Menuet	0.631s	0.726s	3m 50.376s
Joy to the World	2.381s	3.813s	7m 17.115s
Twinkle Twinkle	0.685s	0.716s	2m 31.299s

Cuantitativamente, los resultados del sistema son los esperados. Los tiempos de armonización de partituras son excepcionalmente buenos y los de completado de secciones obtienen resultados muy interesantes ya que para un compás vacío se puede observar un escaso segundo de diferencia. No obstante, se ha podido comprobar que el crecimiento de tiempo necesario para completar una partitura no es lineal, ya que al haber cada vez más figuras que completar, surgen más posibilidades combinatorias.

Esto queda reflejado en el tiempo necesario para añadir una nueva voz a la partitura, que crece hasta el orden de los minutos para una voz. No resulta problemático, ya que teniendo en cuenta la cantidad de posibilidades sigue siendo un orden de tiempo aceptable para hallar el mejor resultado posible, si tenemos en cuenta que podemos especificar un timeout, podríamos obtener un resultado aceptable (por ejemplo una voz nueva con una ratio de errores por nota de alrededor del 10%) en prácticamente la mitad de tiempo.

Cualitativamente, los resultados de armonización son adecuados y el completado de secciones o la inclusión de nuevas voces ofrece soluciones interesantes y correctas armónicamente. En algunos casos, por culpa del módulo de salida, se producen errores visuales al asignar la clave de forma equivocada (Ver Figura 4.5), pero este error es subsanable en el editor de partituras y no presenta ningún fallo real en la armonización ni completado.

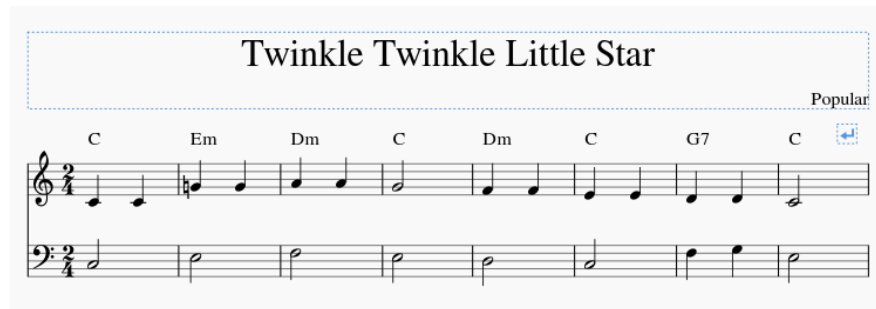


Figure 4.6: Comienzo de Twinkle Twinkle Little Star armonizado

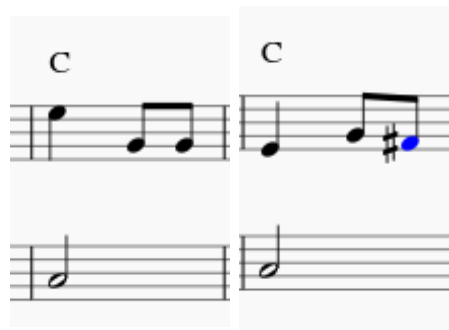


Figure 4.7: Compás de Twinkle Twinkle Little Star completado sin y con preferencias melódicas activadas

Las pruebas de carga sobre “Twinkle Twinkle Little Star” se han realizado primero vaciando progresivamente una de las voces. La pieza cuenta con 24 compases, se han vaciado 4 compases en cada iteración, aunque no se ha vaciado del todo la voz ya que eso corresponde al siguiente punto. Por último se han ido añadiendo voces, una voz de una tesitura diferente en cada iteración.

Prueba	Tiempo
4 compases	1.481s
8 compases	2.394s
12 compases	3.978s
16 compases	3.982s
20 compases	5.966s
1 voz	2m 31.299s
2 voces	25m 17.298s

No se realizaron ejecuciones con más de dos voces adicionales ya que el tiempo de ejecución ya se dispara a las decenas de minutos y no se consideró relevante, se

puede establecer el límite del funcionamiento usable en dos voces si se interrumpe la ejecución o en una voz completa. Puede resultar curioso que una pieza casi vacía (como en el ejemplo de 20 compases vacíos) siga en el orden de segundos mientras que una voz nueva se va a los minutos. La explicación de esto radica en las figuras completables contra las nuevas voces. Las figuras completables cuentan con un patrón rítmico determinado al que se ajustan las notas, reduciendo drásticamente la cantidad de notas que hay que calcular para la solución. En una nueva voz no existe este patronaje rítmico, las figuras son siempre la figura base con la que se ha armonizado y subdividido internamente toda la pieza, resultando siempre en tener que calcular el máximo de notas en cada voz. Para solucionar esto se pueden incluir voces nuevas en la partitura directamente en el editor, especificando los patrones rítmicos, en lugar de a través de la línea de comandos.

4.1 Comparativa

El presente proyecto es un sistema relativamente único en cuanto a qué hace y cómo lo hace. La única herramienta comparable sería el software ANTON [2] ya que ambos tienen su fundamento en *Answer Set Programming*, pueden trabajar sobre partituras ya creadas y funcionan mediante módulos destinados a las diferentes partes del proceso de composición. La gran diferencia radica en que ANTON es un sistema mucho más complejo que el presente, permitiendo componer no solo armónica sino también melódicamente y, además, ANTON contempla la creación de patrones rítmicos. Por otra parte, el proyecto aquí creado ahonda más en la armonía que ANTON, permitiendo simplemente armonizar una pieza completa y completar partituras de cualquier número de voces y estilos, mientras que ANTON se limita a la composición de piezas de dos voces en un estilo renacentista concreto.

Ya que ANTON trabaja siempre con dos voces resulta algo complejo comparar tiempos de ejecución, pero las pruebas realizadas al sistema para duetos de 32 compases completamente nuevos daban resultados en el orden de los minutos. El sistema actual presenta un rendimiento similar a la hora de crear una segunda voz para convertir una pieza monofónica en un dueto, pero al no contemplar ritmo y necesitar la primera voz para componerla, puede verse la inferioridad del sistema frente a ANTON.

4.2 Problemas conocidos

Las diferentes pruebas han revelado ciertos problemas que se producen si no siempre, sí lo hacen con una frecuencia relativamente alta y que por diseño de la herramienta no han sido atacados. En la sección de 5.1 Trabajo Futuro en el capítulo 5 Conclusión se detalla cuales de ellos se pretenden arreglar a corto o medio plazo.

- **Tresillos:** Los tresillos, dosillos y otras figuras irregulares no funcionan correctamente en la herramienta y han de ser editados en la partitura antes de procesarla. Esto se debe a que no es posible realizar una subdivisión directa a la nota base de la partitura de este tipo de figuras.
- **Clave:** Algunas voces aparecen en la salida con la clave mal identificada, produciendo un resultado poco legible. Esto es fácilmente solucionable cambiando la clave de la voz correspondiente en el archivo de salida, ya que pese a todo, esto es un fallo meramente visual y no afecta a los valores de las notas de la voz.
- **Nombres de Tesituras:** Si bien los nombres de las voces corales no varían sustancialmente, los nombres de los instrumentos sí lo hacen, por esto el *locale* de la herramienta usada para la edición de partituras puede producir incompatibilidades a la hora de restringir el rango de voces, por ejemplo, una partitura para escrita para Violonchelo en un sistema en castellano marcará esa voz como `voice_type(x, violonchelo)` mientras que un sistema en inglés producirá `voice_type(x, violoncello)`. Para solucionarlo se ruega editar el fichero `voice_types.lp` en la carpeta `asp/include` para que los límites de cada instrumento se correspondan con el locale del sistema.
- **Falsos positivos:** Debido a que en algunas figuras como los conjuntos de corcheas o semicorcheas consecutivas no se puede identificar los subtiempos débiles y fuertes que no dependen del compás si no de la propia figura junto con algunas dificultades a la hora de identificar los tiempos débiles y fuertes de un compás con las notas subdivididas y después transformarlo a otro tipo de compás, es probable que algunas notas marcadas como error en la partitura no lo sean realmente.

Conclusiones

EN el proyecto se ha construido una herramienta software capaz de tomar como entrada una partitura polifónica parcial, deducir los acordes correspondientes a la unidad de tiempo deseada, y si el usuario lo deseara, completarla respetando las reglas básicas de armonía, de modo similar a los ejercicios habituales en esta disciplina musical. Se ha logrado abarcar la estructura musical de forma tanto horizontal como vertical y también se ha probado empíricamente que este problema de armonización se puede especificar en términos de resolución de restricciones.

El proyecto hizo uso del 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 fue la facilidad que otorgaba 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 proporcionó 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. En el estado actual de **haspie**, pueden considerarse las reglas de cada módulo una entrada más del problema a resolver, junto con los hechos, lo que permite modificar su comportamiento con un esfuerzo mínimo.

Otra ventaja importante de ASP para este escenario fue la posibilidad de implementar y usar preferencias, ya que algunas reglas armónicas no son estrictas, sino que se busca que se respeten en la medida de lo posible.

En resumen, se diseñó e implementó con éxito un conjunto de módulos que, funcionando como uno solo, son capaces de ofrecer buenos resultados a ejercicios sencillos de armonización.

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

- **Entrada y preprocesado:** Haciendo uso de un editor musical capaz de exportar al formato de entrada se produce un fichero que este modulo convierte a hechos lógicos.
- **Armonización:** Escrito en ASP, mediante el uso de software que calcula las restricciones para el fichero de entrada, este módulo produce soluciones al problema de armonización.
- **Completado:** En caso de que el problema así lo requiera, este módulo completa la partitura en la medida de lo necesario.
- **Salida y postprocesado:** Tomando como entrada las soluciones en forma de hechos lógicos, produce un fichero en el formato de salida especificado para su posterior visualización.

Se han mantenido las restricciones establecidas al proyecto, tanto la de no buscar resultados con coherencia melódica como la de no implementar de ningún modo la capacidad de detectar y trabajar con modulación en piezas musicales.

A lo largo del desarrollo del proyecto no ha sido necesario descartar funcionalidad o tomar una dirección diferente a la planteada de forma inicial, aunque a pesar de ello, sí que ha sido necesario refactorizar el módulo ASP hacia el final del ciclo de desarrollo. Dicho módulo, encargado de armonizar y completar partituras a la vez, fue dividido en dos submódulos, uno encargado de armonizar y otro encargado de completar la partitura, para limitar el espacio de búsqueda y así mejorar sustancialmente el rendimiento general de la herramienta.

Los resultados de **haspie** son, cualitativa y cuantitativamente, realmente buenos, y compensan enormemente los defectos del proyecto en su estado actual. No obstante, el mayor logro obtenido ha sido la versatilidad de la herramienta. Esta flexibilidad se ha logrado al no depender del proceso de búsqueda ni de definir heurísticas, ya que de eso se encargan herramientas externas. El comportamiento de **haspie** puede ser alterado de forma limitada a través de sus ficheros de configuración que permiten alterar enormemente el estilo y los resultados obtenidos,

pero al mismo tiempo, el comportamiento de la herramienta puede ser alterado de forma ilimitada a través de la definición de nuevas reglas. Bajo este punto de vista, el usuario solo tiene que modificar el conjunto de reglas, definiendo o prohibiendo determinados predicados, sin preocuparse del procesamiento de la entrada ni de la búsqueda ni de la representación de la salida.

Aunque el balance resulta positivo, el proyecto no está libre de errores, algunas limitaciones, impuestas por el propio proyecto o por falta de recursos, hacen que flaquee en algunos aspectos:

- La subdivisión en tiempos débiles y fuertes, que falla para algunas secuencias rítmicas o tipos de compases.
- La interfaz, pese a ser plenamente funcional a través de la línea de comandos, es muy pobre.
- Pese a la flexibilidad, aún son necesarios conocimientos de ASP, del dominio particular del sistema y de teoría musical.
- No ha sido probado con una base de usuarios.
- Aún necesita ser pulido y sus resultados necesitan ser contrastados con expertos para poder ser utilizado en enseñanza.

Finalmente, aunque no por ello menos importante, el proyecto ha resultado ser tremendamente entretenido de desarrollar, ya que cada nueva iteración ha presentado nuevos retos a implementar mediante ASP. Esto ha servido de ayuda y guía para la implementación de pequeños proyectos adicionales bajo el mismo paradigma y ha incrementado la habilidad y soltura del alumno con Answer Set Programming. Además, la necesidad de aplicar conocimiento de teoría musical ha resultado muy interesante, ya que ha permitido ampliar conocimientos sobre la materia.

5.1 Trabajo Futuro

Las líneas marcadas para el trabajo futuro sobre este proyecto tienen que ver principalmente con solucionar o mejorar algunos de los puntos mencionados en la sección 4.2 Errores Conocidos. La guía principal sobre el trabajo pendiente tiene

que ver con la estética de los resultados del mismo e incluye la implementación de algún tipo de interfaz más amigable para el usuario. También se pretende mejorar la interpretación de partituras en MusicXML para mejorar los resultados de la armonización y los de la representación de la salida. Por último desea investigar también sobre ampliar el proyecto hacia alguna de las restricciones propuestas inicialmente, como la de contemplar modulación.

5.1.1 Estética e Interfaz

Ya que la librería en la que se sustenta la representación visual de las partituras en la salida aún se encuentra en desarrollo, se esperará a su anunciada futura versión 3.0 para continuar el trabajo en esta dirección. La propia librería ya debería solucionar en gran medida los problemas de representación, principalmente la inexplicable inclusión de becuadros frente a algunas notas que no tendrían por qué llevarlos o la notación correcta de la clave en cada una de las voces, que a veces da problemas.

Se quiere además transformar el proyecto a un *plug-in* para MuseScore 2, ya que el programa facilita mucho la creación de este tipo de módulos complementarios. Serviría además para ayudar al usuario a usar el programa, ofreciendo una serie de elementos gráficos con los que interactuar. La gran rapidez de armonización de la herramienta ofrecería resultados prácticamente en vivo, lo cual resulta realmente atractivo.

5.1.2 Procesado y Armonización

Se busca implementar en el módulo de entrada una mejor detección del tipo de clave de cada una de las voces, permitiendo a la salida ofrecer un resultado más fidedigno. Además en conjunción con el módulo de armonización se pretende detectar mejor los tiempos débiles y fuertes de la pieza, ya que esta es una de las grandes flaquezas en el estado final del proyecto al no poder identificar algunos patrones más complejos de subdivisión fuerte y débil en conjuntos de figuras como corcheas y semicorcheas. Por último, aunque esto quizás sea lo más complicado, se intentará atacar el problema de la detección y la correcta interpretación de los tresillos y otras figuras irregulares.

5.1.3 Modulación

La modulación fue una de las principales barreras fijadas desde el principio de la planificación del proyecto, principalmente por ser difícil de detectar y complejo lidiar con ella a nivel armónico. Se quiere investigar el comportamiento del proyecto ante esta técnica haciendo uso de nuevas herramientas como *iclingo*, un solucionador del estilo del utilizado en el proyecto pero iterativo, es decir, capaz de calcular nuevas soluciones haciendo uso de resultados obtenidos en iteraciones anteriores que van cambiando el dominio sobre el que se trabaja. Otra aproximación planteada, desde un punto de vista más similar al del proyecto sería ser capaz de subdividir la partitura en tramos según armonías, aunque se perdería mucha información y no siempre estos tramos estarían bien delimitados.

5.1.4 Publicación

Por último, y tras refinar algunos de los pasos citados anteriormente, se desea contactar a interesados en la herramienta del campo de la enseñanza musical para que puedan juzgarla y contribuir a perfeccionarla hasta que sea posible publicarla y, con suerte, ser usada en este campo.

Appendix A

Diagramas

La herramienta no hace un uso intensivo de la orientación a objetos, en lo referente a herencia, composición, clases abstractas y similares. No obstante sí que existe una jerarquía de objetos de almacenamiento y representación de los hechos lógicos representados a la salida de los diferentes módulos. Con respecto al diagrama de secuencia, su ausencia es justificable ya que no ilustra nada ni sirve a ningún propósito real. El diagrama de arquitectura aclara mucho mejor el funcionamiento y el flujo de datos entre los módulos del sistema que cualquier diagrama de secuencia, ya que los diferentes módulos se invocan secuencialmente y la salida de cada uno se recibe en el pipeline y se pasa al siguiente módulo hasta terminar.

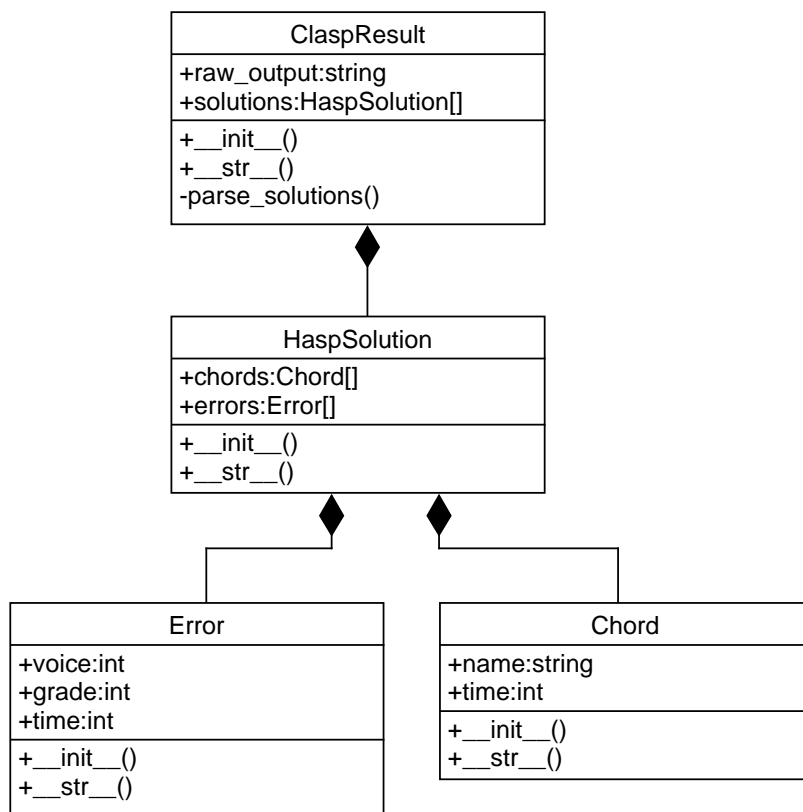


Figure A.1: Diagrama de clases de almacenamiento de la Iteración 3

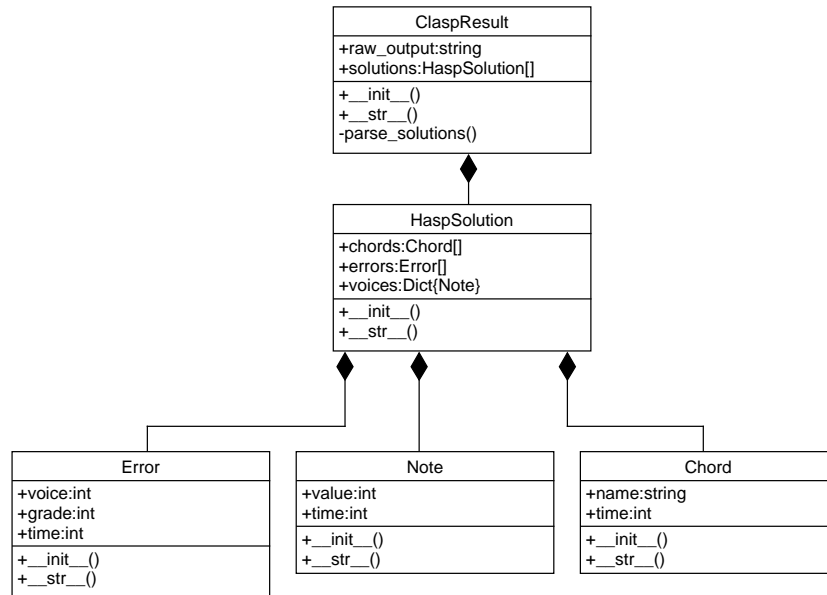


Figure A.2: Diagrama de clases de almacenamiento de la Iteración 4

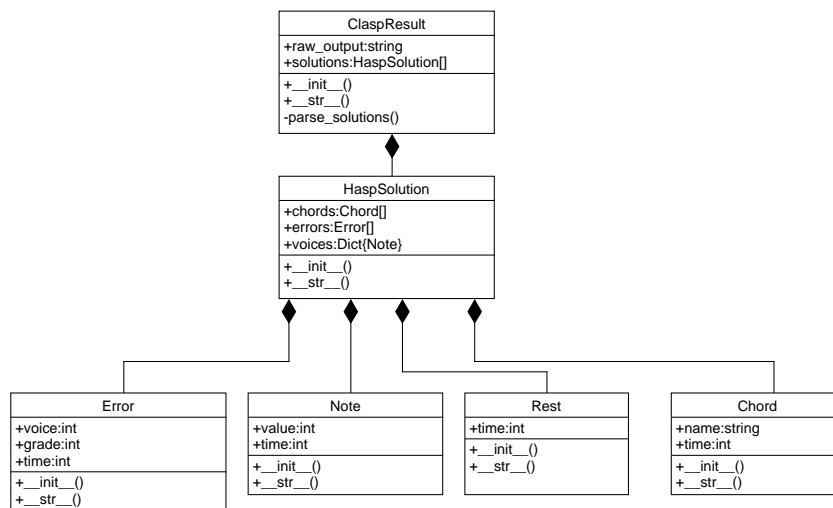


Figure A.3: Diagrama de clases de almacenamiento de la Iteración 5

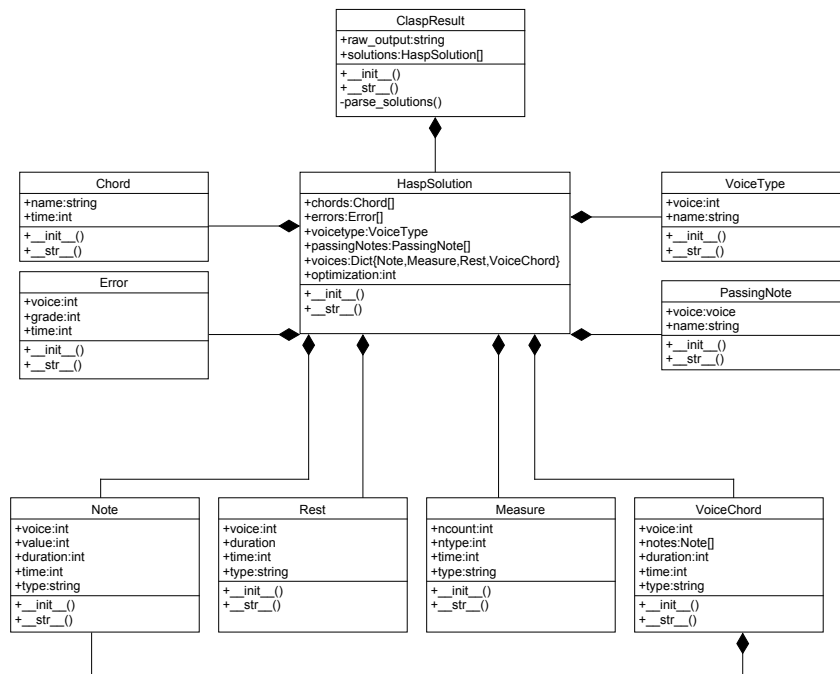


Figure A.4: Diagrama de clases de almacenamiento de las Iteraciones 6, 7 y 8

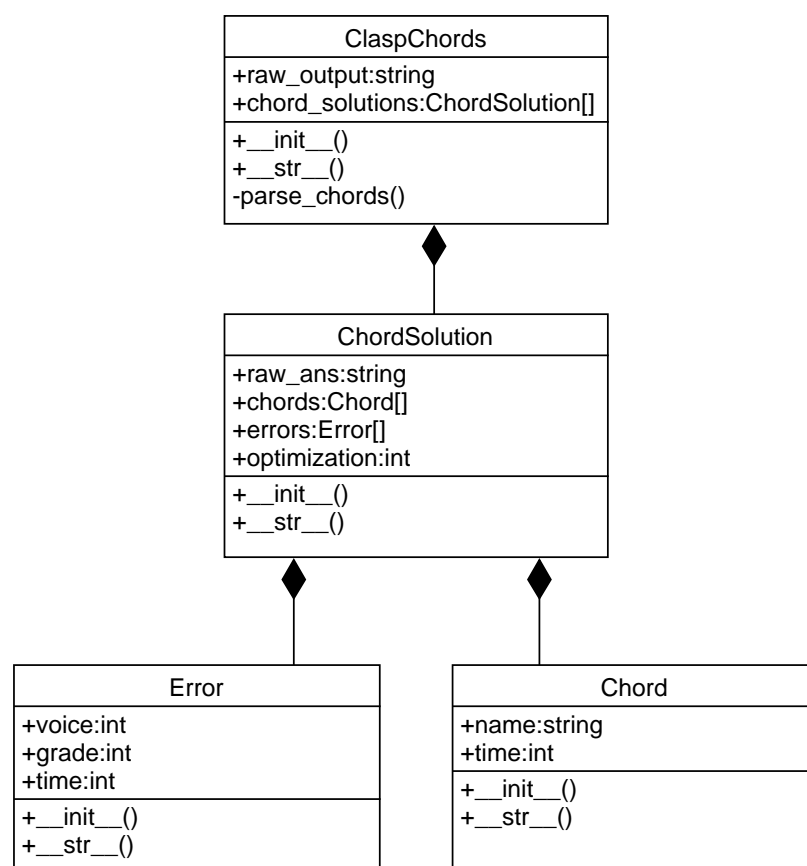


Figure A.5: Diagrama de clases de almacenamiento del módulo de armonización de la iteración 8

Appendix B

Análisis de MusicXML

Se detalla una relación de los elementos MusicXML identificados por el procesador y se especifica la finalidad de los mismos. Se indica también, el tipo de cada elemento procesado (Etiqueta, Etiqueta Autocerrada o Atributo) y, con fines organizativos, la etiqueta con la que están relacionados de forma directa.

Nombre	Pertenece a	Tipo	Uso
note	measure	Etiqueta	Delimita el bloque correspondiente a una nota o silencio
step	note	Etiqueta	Especifica el nombre de la nota en notación internacional
octave	note	Etiqueta	Indica el número de la octava de la nota
rest	note	Autocerrada	Indica si la figura es un silencio, en caso de aparecer, step y octave no se especifican
chord	note	Autocerrada	Indica si la nota forma parte de un acorde
type	note	Etiqueta	Especifica el tipo de figura mediante un string (whole, half, quarter...)
staff	note	Etiqueta	Identifica el número de pentagrama al que pertenece la nota, usado en instrumentos con múltiples pentagramas como el piano
grace	note	Autocerrada	Indica si la nota es una apoyatura

alter	note	Etiqueta	Especifica el valor de alteración usando números positivos para sostenidos y negativos para bemoles
duration	note	Etiqueta	Indica la duración de la figura
dot	note	Autocerrada	Indica si la nota está acompañada de un puntillo
part	score-partwise	Etiqueta	Delimita los bloques de cada voz o parte de la partitura
score-part	part-list	Etiqueta	Delimita los bloques de meta-información de cada una de las partes
instrument	score-part	Etiqueta	Especifica el nombre del instrumento la parte correspondiente
print-object	note	Atributo	Indica si la nota es visible o no
credit-words	credit	Etiqueta	Almacena metadatos sobre autor o título de la partitura
time	attributes	Etiqueta	Delimita bloques sobre la información de compases
beats	time	Etiqueta	Indica la cantidad de figuras del compás (numerador)
beat-type	time	Etiqueta	Indica la figura base del compás (denominador)
harmony	measure	Etiqueta	Delimita un bloque que especifica la notación de armonía del compás
root-step	harmony	Etiqueta	Indica la nota raíz del acorde que se anota sobre el compás
kind	harmony	Etiqueta	Indica el tipo de acorde anotado sobre el compás (mayor, menor, dominante séptima...)

fifths	key	Etiqueta	Indica la cantidad de sostenidos o bemoles presentes en la armadura de la partitura. Usa números positivos para los sostenidos y negativos para los bemoles
mode	key	Etiqueta	Indica el modo (mayor o menor) de la tonalidad especificada por la armadura de la partitura

Appendix C

Partituras

Además de las utilizadas en la evaluación del sistema, se utilizaron algunas otras partituras durante el desarrollo del proyecto, algunas descartadas por incompatibilidad y otras por no servir de ejemplo para la evaluación.

Greensleeves

Henry VIII of England

The musical score for "Greensleeves" is presented in a system of five staves, each with a Violin (Vln.) and Violonchelo (Vc.) part. The key signature is one sharp (F#) and the time signature is 6/8. The score is divided into measures by bar lines, with measure numbers 5, 9, 13, and 17 indicated at the start of their respective systems. The Violin part is written in treble clef, and the Violonchelo part is written in bass clef. The music features a mix of eighth and sixteenth notes, with some measures containing rests. The score concludes with a double bar line at the end of the fifth system.

Figure C.1: Partitura de Greensleeves, por Enrique VIII

Menuet

Johann Sebastian Bach



Figure C.2: Partitura de Menuet, por Johann S. Bach

Joy to the World

George F. Handel

The image displays a musical score for the hymn "Joy to the World" by George F. Handel. The score is written for four vocal parts: Soprano, Alto, Tenor, and Bass. The key signature is one sharp (F#), and the time signature is 3/4. The score is divided into two systems. The first system contains measures 1 through 8, and the second system contains measures 9 through 12. The Soprano part begins with a treble clef and a key signature of one sharp. The Alto part begins with a treble clef and a key signature of one sharp. The Tenor part begins with a treble clef and a key signature of one sharp. The Bass part begins with a bass clef and a key signature of one sharp. The score is written in a standard musical notation style, with notes, rests, and bar lines clearly visible. The Soprano part has a final measure with a repeat sign. The Alto part has a final measure with a repeat sign. The Tenor part has a final measure with a repeat sign. The Bass part has a final measure with a repeat sign.

Figure C.3: Partitura de Joy to the World, por George F. Handel

Silent Night, Holy Night

Gruber

The image displays a musical score for the hymn "Silent Night, Holy Night" by Franz X. Gruber. The score is written for four vocal parts: Soprano, Alto, Tenor, and Bass. The music is in 8/8 time and G major. The first system shows the first five measures of the piece. The Soprano part begins with a half note G, followed by quarter notes A and B, then a half note C. The Alto part begins with a half note E, followed by quarter notes F and G, then a half note A. The Tenor part begins with a half note C, followed by quarter notes B and A, then a half note G. The Bass part begins with a half note G, followed by quarter notes F and E, then a half note D. The second system shows measures 6 through 10. The Soprano part has a half note G, followed by quarter notes A and B, then a half note C. The Alto part has a half note E, followed by quarter notes F and G, then a half note A. The Tenor part has a half note C, followed by quarter notes B and A, then a half note G. The Bass part has a half note G, followed by quarter notes F and E, then a half note D.

Figure C.4: Partitura de Silent Night (Noche de Paz), por Franz X. Gruber

Twinkle Twinkle Little Star

Popular



Figure C.5: Partitura de la pieza popular infantil Twinkle Twinkle Little Star (Brilla Brilla Estrellita)

Single two



Figure C.6: Partitura de ejemplo con una serie de acordes sencillos

Free Times and Rests



Figure C.7: Partitura de ejemplo con algunos silencios reales y otros invisibles

Soprano



Soprano



Appendix D

Instalación

Se recomienda la instalación de la herramienta en un sistema operativo Linux. Esto se debe a que las librerías Flex y Bison se compilan fácilmente en dicha plataforma. No obstante se proporciona el código fuente completo, por lo tanto es posible compilarlo para otros sistemas operativos.

El requisito es tener instalado Python 2.7, las ultimas versiones de Flex y Bison, además de un compilador de C (gcc sirve perfectamente). Además es necesario descargar tanto gringo 3.0.5 como clingo 3.0.5 de la página en Sourceforge del grupo Potassco [7] y el módulo Music21 en su versión 2.1.2 o superior de la página de la librería. Todas estas herramientas, compiladores y librerías deben ser accesibles mediante en el PATH del sistema operativo.

La instalación de tanto gringo como clingo no tiene mayor complicación, sólo deben ubicarse en alguna carpeta accesible y añadir dicha ruta al PATH. En la página web de Music21 [8] se incluyen instrucciones para su instalación automatizada en el sistema. Se aconseja, no obstante instalar primero diferentes herramientas para transformación y visualización o reproducción de los diferentes formatos (Léase PDF, MIDI, Lilypond, etc.) ya que de este modo, la instalación de Music21 las reconocerá y podrán ser usadas sin configuración posterior. Si se desean cambiar las preferencias sobre estas herramientas vinculadas a Music21, ha de editarse el fichero `.music21rc` presente en la carpeta `home` del usuario activo del sistema.

Se recomienda la instalación de un editor de partituras para poder manipular correctamente los ficheros de entrada y salida de la herramienta, esto es opcional, pero resulta conveniente. En el desarrollo se ha utilizado MuseScore 2 al ser *OpenSource* y gratuito, pero cualquier editor que pueda importar y exportar

ficheros en formato MusicXML servirá.

Con la herramienta, además del código fuente, se ofrece el procesador MusicXML a hechos lógicos precompilado, si no funcionase este archivo siempre puede compilarse el mencionado código fuente haciendo uso del archivo Makefile presente en la carpeta **parser/source**. un simple comando **make** desde esa ruta es suficiente. El archivo binario compilado del procesador debe estar presente en la carpeta **parser** para que la herramienta funcione correctamente.

Appendix E

Manual de uso

Esta herramienta está pensada para ser usada desde la línea de comandos en sistemas operativos Linux. El comando básico para su ejecución es `python haspie.py <fichero.xml>` y las opciones que permite la herramienta, junto con una pequeña explicación de cada una de ellas, pueden ser invocadas mediante la opción `-h`, como es habitual. No obstante se detalla una ejecución completa a modo de guía de uso.

```
usage: haspie.py [-h] [-n N] [-s S] [-v V [V ...]] [-S] [-f xml|pdf|midi|ly]
                [-o output/dir/for/file] [-t T] [-k A~G+~?] [-m major|minor]
                [-M] [-6] [-a] [-O O] [-c config_file_name.lpf]
                XML_SCORE
haspie - Harmonizing music with ASP
positional arguments:
  XML_SCORE            input musicXML score for armonizing
optional arguments:
  -h, --help          show this help message and exit
  -n N, --num_sols N  max number of ASP solutions, by default all of them
  -s S, --span S      horizontal span to consider while harmonizing, this
                      takes in account subdivision, by default 1
  -v V [V ...], --voices V [V ...]
                      extra instruments, these can be input by name or by
                      numerical note range (i.e soprano,guitar,(65,90)...)
                      to leave one of the sides of the range unespecified
                      use 0
  -S, --show          show result in editor instead of writing it to a file
                      in the desired format
  -f xml|pdf|midi|ly, --format xml|pdf|midi|ly
                      output file format for the result
```

```
-o output/dir/for/file, --output output/dir/for/file
                                output file name for the result
-t T, --timeout T              maximum time (in seconds) allowed to search for
                                optimum when searching for all optimums, by default
                                there is no time limit
-k A~G+--?, --key A~G+--?
                                key in which the score should be harmonized, if not
                                specified, parser will autodetect it
-m major|minor, --mode major|minor
                                mode of the scale, if not specified, parser will
                                autodetect it
-M, --melodious                turns on melodic preferences in ASP for a more melodic
                                result
-6, --sixthslink                turns on sixth-four chord linking in ASP for a more
                                natural result (very heavy)
-a, --all_optimums             turns on the search for all optimums when completing
                                and not just the first found, disabled by default
-O 0, --max_optimums 0
                                max number of optimum solutions to display in score
                                completion, by default it's 10
-c config_file_name.lp, --config config_file_name.lp
                                reads preference order and weights for parameters from
                                the desired *.lp file stored in /pref folder
```

Lo primero que se debe tener en cuenta es que el fichero de entrada debe proporcionarse en formato MusicXML estándar, este tipo de ficheros son uno de los más comunes en intercambio de partituras y no es difícil encontrar la partitura que deseamos armonizar en este formato. De no ser así, casi cualquier herramienta moderna de composición musical por ordenador puede exportar archivos a este formato. Ya que durante el desarrollo del proyecto se ha usado MuseScore 2, las figuras que acompañan esta sección son capturas de esta herramienta. Para opciones similares en otros editores, consúltase el manual de cada uno.

Una vez obtenido el archivo XML que se desea armonizar y/o completar, puede invocarse el comando básico ya mencionado `haspie.py <fichero.xml>` para obtener una armonización en la que se asignará un acorde a cada tiempo subdividido de la partitura. Es sumamente importante tener en cuenta la subdivisión de las notas de la partitura ya que la herramienta, para funcionar correctamente, subdivide todas las figuras a la nota más breve. Con esta llamada se

calculan automáticamente todos los parámetros de armonización de la partitura: Clave y Modo. En la misma consola de comandos se imprime una serie de datos de mayor o menor utilidad para el usuario como el título de la pieza, la clave y modo detectados así como la longitud de la nota más breve de la partitura, usada como subdivisión.

```
trigork@deus-machina:~/haspie-haspist$ time python haspie.py test_input/Greensleeves.xml -s 12
Greensleeves by Henry VIII of England
Base note - 1/16
Detected G major key from key signature
Harmonizing in G major
OK - Correctly generated music logic file in asp/generated_logic_music/Greensleeves.lp
Extra score information can be found in tmp/score_meta.conf
```

Figure E.1: La información relevante de la partitura se muestra al comenzar la armonización

Tras la armonización se pedirá al usuario que escoja una de las soluciones propuestas por el módulo de armonización. Para esto se muestran todas numeradas y en orden inverso de optimización, siendo las últimas las mejores, de modo que el usuario solo tiene que introducir el número de aquella que desee utilizar. Si sólo se pulsa Enter, se escogerá por defecto la última ya que debería ser mejor (Aunque podría haber varias soluciones empatadas).

A continuación se pasa al módulo de completado de partitura, ya que no se han especificado voces adicionales ni se ha modificado la partitura para contener secciones a completar, no se pedirá escoger una solución de modo similar a la armonización, al existir solo un posible resultado. Finalmente, se genera en la carpeta `out` el fichero de salida de nombre igual al de entrada, por defecto en formato MusicXML.

De todos modos las armonizaciones más frecuentes no suelen ser a tiempo si no a compás o medio compás. Para cambiar este comportamiento se puede hacer uso de la opción `-s N` donde `N` indica la cantidad de tiempos subdivididos que se tomarán en cuenta horizontalmente a la hora de asignar un acorde. Por ejemplo, en la pieza *Joy to the World* se usa un compás **2/4** (es decir, dos negras por compás), pero la nota más breve de la partitura es una semicorchea, haciendo que la longitud, en semicorcheas, de este compás sea de 8. El cálculo es sencillo de realizar, solo hay que dividir la longitud fraccionaria de la nota más breve entre la longitud fraccionaria del compás y multiplicar este resultado por la cantidad de figuras originales del compás. Para este ejemplo podemos calcular $(16/4) * 2 = 8$.

Normalmente la armadura es un buen indicativo de la clave en la que está

```

Answer 62:
Errors: Voice: 3, 169 // Voice: 3, 121
iv iiim v7 vim iiim i v vim iv vim v iv iiim vim v7 vim iim
Optimization - Errors: 2 // Repeated Chord: 0 // Errors (weak beat): 112

Answer 63:
Errors: Voice: 3, 169 // Voice: 3, 121
iv iiim v vim iiim i v7 vim iv vim v iv iiim vim v7 vim iim
Optimization - Errors: 2 // Repeated Chord: 0 // Errors (weak beat): 112

Answer 64: ke, A. Neumann, and T. Schaub. The nomore++
Errors: Voice: 3, 169 // Voice: 3, 121
iv iiim v vim iiim i v vim iv vim v iv iiim vim v7 vim iim
Optimization - Errors: 2 // Repeated Chord: 0 // Errors (weak beat): 112
Select a chord solution to complete the score (1..64) [64]:
Errors: Voice: 3, 169 // Voice: 3, 121
iv iiim v vim iiim i v vim iv vim v iv iiim vim v7 vim iim
Optimization - Errors: 2 // Repeated Chord: 0 // Errors (weak beat): 112

```

Figure E.2: Se pide al usuario que fije la armonización a utilizar de entre varias posibles

escrita la pieza y por tanto de la tonalidad y modo en los que es correcto armonizarla, aun así estos parámetros pueden ser modificados usando `-k CLAVE` y `-m major|minor` respectivamente. El formato en el que se debe especificar la clave es el nombre internacional de la tonalidad (A-G) junto con un símbolo opcional `+` o `-` en caso de querer alterarla con un sostenido o un bemol respectivamente. Modificar estos parámetros alterará drásticamente la armonización de la partitura y lo más probable es que si no coinciden con los detectados automáticamente producirá resultados erróneos.

Para modificar el comportamiento del módulo de completado de partituras se puede hacer uso de dos módulos de preferencias adicionales que refinan la salida para obtener un mejor resultado a cambio de requerir más tiempo de búsqueda de las soluciones. Estas preferencias tienen en cuenta las notas presentes en la entrada, pero no las modifican de ningún modo. El módulo de preferencias melódicas está orientado a ofrecer unas melodías más suavizadas y pulidas, principalmente minimiza la distancia de los saltos melódicos entre notas de la partitura y realiza un análisis de tendencia de las otras voces para imitar dicha tendencia. Este módulo puede ser utilizado con la opción `-M`. El módulo de enlaces de sextas está basado en la detección y enlazado de inversiones de cuarta y sexta de acordes, muy común en polifonía. Este módulo intenta detectar estos patrones y de hallarlos busca que las nuevas notas generadas los continúen. Es significativamente costoso computacionalmente, puede ser invocado mediante la opción `-6`.

Para configurar más aún el comportamiento de ambos módulos, la importancia de cada uno de los parámetros que se maximizan o minimizan durante la búsqueda de las mejores soluciones puede ser alterada haciendo uso de ficheros de configuración en formato ASP. En la carpeta **pref** se encuentra uno comentado con las opciones por defecto. De querer hacer uso de un archivo de configuración solo hay que indicarle la ruta del mismo mediante la opción **-c /ruta/fichero/config.lp**.

Cada parámetro de los ficheros de configuración cuenta con dos valores:

- **Peso:** Indicado mediante **nombre_weight** mide la importancia de cada uno de estos hechos en el resultado.
- **Prioridad:** Indicado mediante **nombrep** indica el orden en el que se ordenarán las preferencias, a más valor, mayor relevancia en el orden.

Para ajustar la cantidad de resultados puede usarse la opción **-N N**, aunque usarla limitará la búsqueda de resultados a N y no garantiza que se encuentre el mejor valor posible, de no usarse, siempre se explorará todo el espacio de búsqueda.

Para limitar la cantidad máxima de resultados óptimos mostrados por el módulo de completado de partituras puede usarse la opción **-O 0**, ya que al completar puede hallarse una gran cantidad de resultados. A diferencia de **-N** esta opción solo restringe la cantidad de óptimos mostrados, independientemente de haberse limitado el espacio de búsqueda.

Para controlar el tiempo de ejecución del módulo de completado, ya que este puede dispararse al querer completar grandes secciones puede usarse la opción **-t tiempo** para expresar cuantos segundos se quiere invertir en la búsqueda. La herramienta siempre usará la cantidad de tiempo especificada, por defecto es 5 segundos.

Existen dos modos de solicitar al módulo de completado de partituras que haga su trabajo:

- **Silencios Completables:** Se realiza mediante un marcado especial en el editor de partituras.
- **Inclusión de voces nuevas:** Se realiza desde la línea de comandos.

Los silencios completables son silencios especiales que el procesador interpreta como huecos en vez de como silencios. Para lograr esto se hace uso de una meta-propiedad de las figuras en MusicXML: Su visibilidad. Cualquier figura puede marcarse como invisible desde el panel de propiedades de la figura en el editor, con el fin de que esta no sea impresa en el momento de hacerlo. Esta herramienta entiende los silencios marcados como “no visibles” como huecos a rellenar y la figura utilizada para rellenar dicho espacio será la correspondiente al silencio “no visible” que aparecía originalmente en la partitura.

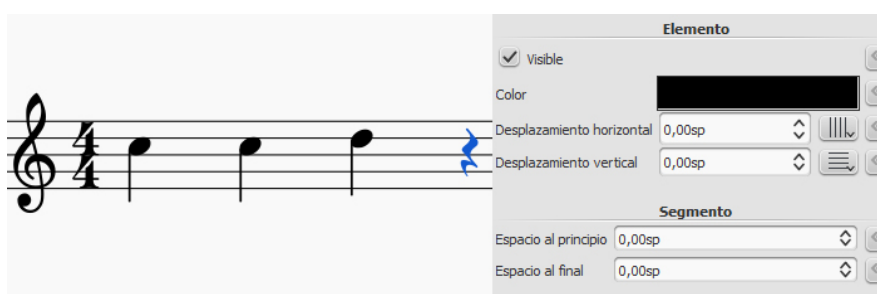


Figure E.3: Seleccionando una figura podemos ver sus propiedades, entre otras la casilla de visibilidad

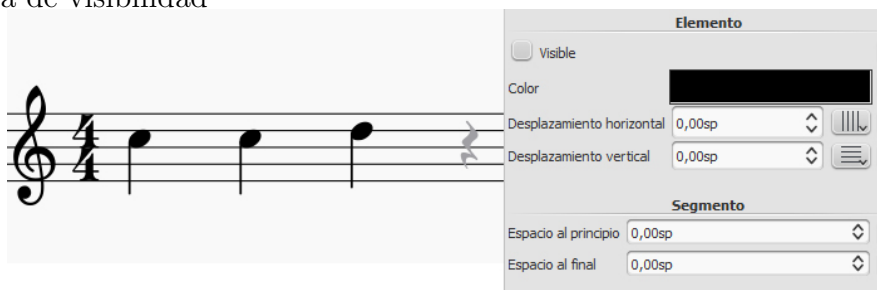


Figure E.4: Desmarcando la casilla de visibilidad, la figura pasa a un color más claro, indicando que no es visible

Las voces nuevas son líneas melódicas completamente vacías originalmente que el módulo de completado se encarga de rellenar respetando la armonía y la tesitura de la voz. Para hacer esto solo hay que usar la opción `-v voz1 [voz2] ...` que toma tantas voces como se quiera añadir como parámetro. Pueden ser especificadas mediante rango de numérico de notas en la forma `min,max`, usándose 0 si quiere dejarse uno de los extremos abiertos o mediante nombre si la tesitura (o rango de notas) está definida en el fichero `asp/include/voice_types.lp`. En éste figuran las tesituras corales más comunes, pero cualquier nuevo instrumento puede ser definido a gusto del usuario.

```

Answer 1:
Chords: [iv, vim, v, iv, iiim, vim, v7, vim, iim, iiim, v, vim, iiim, i, v, vim, iv]
Errors: Voice: 3, 169 // Voice: 3, 121
Voice 1 - violin: [(6/8), R[8], R[2], 64[2], 67[4], 69[2], 71[3], 72[1], 71[2], 69[4], 66[2], 62[3],
64[1], 66[2], 67[4], 64[2], 64[3], 62[1], 64[2], 66[4], 62[2], 59[4], 64[2], 67[4], 69[2], 71[3], 72[
1], 71[2], 69[4], 66[2], 62[3], 64[1], 66[2], 67[3], 66[1], 64[2], 63[3], 60[1], 63[2], 64[6], 64[4],
R[2], 74[6], 74[3], 73[1], 71[2], 69[4], 66[2], 62[3], 64[1], 66[2], 67[4], 64[2], 64[3], 62[1], 64[
2], 66[4], 62[2], 59[6], 74[6], 74[3], 73[1], 71[2], 69[4], 66[2], 62[3], 64[1], 66[2], 67[3], 66[1],
64[2], 63[3], 60[1], 63[2], 64[6], 64[4], R[2]]
Voice 3 - violonchelo: [(6/8), R[12], 52[2], 47[2], 54[2], 55[2], 50[2], 55[2], 54[2], 50[2], 57[2],
54[2], 50[2], 57[2], 52[2], 50[2], 48[2], 55[2], 57[2], 55[2], 50[2], 52[2], 54[2], 57[2], 55[4], 52[
2], 59[2], 60[2], 62[2], 64[2], 62[2], 60[2], 62[2], 57[2], 54[2], 55[2], 57[2], 52[2], 47[4], 47[2],
45[2], 47[2], 48[2], 50[2], 54[2], 52[2], 55[2], 57[2], 59[2], 57[2], 55[2], 54[2], 52[2], 50[2], 49
[2], 47[2], 45[2], 47[2], 49[2], 50[2], 52[2], 54[2], 55[2], 57[2], 59[2], 61[2], 62[2], 61[2], 59[2]
, 57[2], 55[2], 52[2], 59[2], 57[2], 55[2], 54[2], 52[2], 50[2], 49[2], 47[2], 45[2], 47[2], 49[2], 5
0[2], 52[2], 50[2], 48[2], 47[2], 45[2], 47[2], 43[2], 45[2], 42[2], 40[4], R[2]]
OPT: [2, 112, 14]
Select a solution to output (1..1) [1]:

```

Figure E.5: Se pide al usuario que seleccione la solución que más le guste entre varias posibles

En caso de requerir completar la partitura de algún modo, tras la armonización, la herramienta solicitará al usuario que escoja la solución que más le guste, mostrando los errores, acordes y notas de cada una de las mejores soluciones halladas.

Si se quiere obtener el fichero en algún formato de salida diferente a MusicXML este puede ser especificado mediante **-f formato**, siempre y cuando se dispongan de herramientas para realizar la conversión instaladas en el sistema. Por ejemplo, para exportar a Lilypond usando **-f ly** precisaríamos de tener instalado el propio entorno de Lilypond en el sistema. Así mismo se puede cambiar el nombre del fichero de salida con la habitual opción **-o /ruta/de/salida** (La ruta debe existir).

Por último, si no existiese interés en almacenar el fichero de salida y solo se deseara previsualizar el resultado, puede usarse la opción **-S** para ello. Esto generará un fichero temporal y se ejecutará el software relacionado con la extensión seleccionada para previsualizarlo.

Bibliography

- [1] D. Cope, *Experiments in Musical Intelligence*, 2nd ed. A-R Editions, 1996.
- [2] G. Boenn, M. Brain, M. De Vos, and J. Ffitch, “Automatic Music Composition using Answer Set Programming,” *ArXiv e-prints*, Jun. 2010.
- [3] G. H. Knud Jeppesen, *Counterpoint: the polyphonic vocal style of the sixteenth century*, 1st ed. Englewood Cliffs, N. J., Prentice-Hall, 1990.
- [4] G. Brewka, T. Eiter, and M. Truszczyński, “Answer set programming at a glance,” *Commun. ACM*, vol. 54, no. 12, pp. 92–103, Dec. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2043174.2043195>
- [5] M. Gelfond and V. Lifschitz, “The stable model semantics for logic programming.” MIT Press, 1988, pp. 1070–1080.
- [6] 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.
- [7] “Potassco - the potsdam answer set solving collection,” accessed: 2016-01-08. [Online]. Available: <http://potassco.sourceforge.net/>
- [8] “music21 a toolkit for computer-aided musicology,” accessed: 2015-11-05. [Online]. Available: <http://web.mit.edu/music21/>
- [9] E. Herrera, *Teoría Musical Y Armonía Moderna - Volumen 1*, 1st ed. Antoni Bosch, 2015.
- [10] MIDI Manufacturers Association, “Midi 1.0 detailed specification.”
- [11] 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.

- [12] V. Tran, “Music composition using artificial intelligence,” 2009.
 - [13] 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.
 - [14] 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
 - [15] 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: <http://dx.doi.org/10.1023/A:3A1018930122475>
 - [16] “LilyPond: Musical notation for everyone,” accessed: 2015-11-05. [Online]. Available: <http://www.lilypond.org>
-