Modelo Probabilístico en CLAM

Reconocimiento musical mediante Modelos Ocultos de Markov

Nir Lipovetzky Chervin
nlipovetzky@iua.upf.es

Director: Xavier Amatriain
Universitat Pompeu Fabra

16 de diciembre de 2004

Agradecimientos

En primer lugar quisiera agradecer a las personas del entorno de trabajo que me ayudaron a lo largo del desarrollo de este proyecto, Xavier Amatriain, Miguel Ramirez, Enric Guaus, al grupo de desarrollo de CLAM, a Marcos, compañero de fatigas y tantos otros que debería citar.

Cómo no, agradezco a mis padres y a mi hermana por infinitas razones, a esos amigos que forman parte de mi familia y a Julio Cortázar, J. L. Borges, Nietzsche, etc. que me han acompañado desde otra vertiente, a lo largo de mi vida.

Abstract

En este proyecto final de carrera se pretende conseguir un estudio sobre el modelo probabilístico. El enfoque será encontrar relaciones entre éste y el modelo de redes de Procesos (Process Network) conceptual de CLAM, librería en C++ orientada al desarrollo de aplicaciones relacionadas con el procesamiento de sonido. Se estudiará el Modelo Probabilístico, el de Redes de Procesos y se extraerá las conclusiónes de la proyección del primero sobre el segundo. Más concretamente, para poder servir de base experimental, se profundizará en la búsqueda de paralelismos entre los modelos ocultos de Markov (HMM)y el anteriormente citado de CLAM. Para ello se hará una implementación de HMM a partir de la cual se extrapolarán las conclusiones más concretas.

Índice general

Índice de figuras

Capítulo 1

Introducción

Antes de empezar a describir los principales rasgos del proyecto, contextualizaremos el entorno en el cuál se desarrolló y el marco en el cual se motivó el emprendimiento de su realización.

1.1. Motivación y Objetivos

Dentro del *MTG* (Music Technology Group) hace varios años se inició un proyecto relacionado con el procesado de señal estadístico, consecuencia del cual se construyó la librería *Amadeus* para Linux, optimizada para los casos de uso que se encontraron dentro del proyecto *Aida*, un sistema de reconocimiento de voz.

Paralelamente se está desarrollando *CLAM*, librería multiplataforma con la finalidad de ser un framework estándar que ayude a desarrollar aplicaciones relacionadas con el procesado de señal y donde a su vez, trata de unificar los puntos de vista sobre temas comunes existentes en distintos proyectos del MTG.

Actualmente hay varios proyectos que trabajan sobre la extracción de parámetros y el procesado de señal estadístico, cada uno con su propia abstracción, hecho que dificulta el intercambio de datos y conocimientos. A raíz de este problema se planteó la posibilidad de adaptar los conceptos existentes dentro de Amadeus (modelo probabilístico) a la librería CLAM (modelo de redes de procesos).

En este proyecto final de carrera se pretende conseguir un estudio sobre el modelo probabilístico. El enfoque será encontrar relaciones entre éste y el modelo de redes de Procesos (Process Network) conceptual de CLAM. Se estudiará el Modelo Probabilístico, el de Redes de Procesos y se extraerá las conclusiones de la proyección del primero sobre el segundo. Más concretamente, para poder servir de base experimental, se profundizará en la búsqueda de paralelismos entre los modelos ocultos de Markov (HMM) y el anteriormente citado de CLAM. Para ello se hará una implementación de HMM a partir de la cual se extrapolarán las conclusiones más concretas.

Una vez llegado a este punto nos planteamos una serie de preguntas:

- ¿Cómo se adapta el modelo probabilístico a una librería planteada según la base del modelo Process-Networks?
- ¿Qué Paralelismos uno puede extraer?
- ¿Es eficiente plantear una implementación?
- ¿Qué adecuación de conceptos existe entre ambos paradigmas?

El principal objetivo sobre el que se basará el siguiente trabajo, será estudiar la proyección del modelo probabilístico sobre las redes de procesos, teniendo en cuenta la correspondencia entre conceptos, el grado de adecuación, la eficiencia, etc.

Otro rasgo interesante que se desarrollará será la comparación con proyecciones a otros modelos existentes, hecho que nos ayudará a enriquecer la nuestra y nos dará puntos de vista diferentes para poder juzgar los resultados.

Más específicamente se explicará como se implementa los Modelos ocultos de Harkov sobre CLAM, estudiando los diagramas que se crean, la generalización necesaria para abarcar el mayor numero de casos de uso y planteando una generalización de AMADEUS.

Desde el punto de vista práctico se creará sobre el modelo de redes de procesos, un framework para utilizar el modelo probabilístico, más concretamente HMM, con un alto grado de generalización.

1.2. Introducción al Modelo Funcional del Sistema

Un primer objetivo del sistema va a ser dar respuesta en CLAM a tres funcionalidades clave en AMADEUS:

- Entrenamiento
- Generación de descriptores
- Reconocimiento de una señal

A la primera funcionalidad responderemos con el uso de un sistema llamado lnkNet. Tanto el Entrenamiento como la Generación de parámetros y el Reconocimiento, comparten la necesidad de una función clave que sirve como base de funcionamiento, una primera subetapa que se realiza en los tres apartados. Éste subapartado consiste en el análisis de la señal de entrada realizando una extracción de parámetros que nos servirá para caracterizar la señal, de una forma más comprimida y obteniendo datos de alto nivel.

Podemos observar en la figura 1.1, qué los tres componentes funcionales reciben como entrada una señal audio que se codifica con un algoritmo específico que realiza la extracción de parámetros (MFCC, etc.). También cabe destacar que el orden en el cual se ha presentado las distintas fases no es casual, primero se debe realizar la fase de entrenamiento, sin el cual no se puede pasar a realizar la siguiente consistente en la generación de descriptores, que a su vez es imprescindible para llegar a la última fase de reconocimiento de la señal.

En el primer componente, los parámetros extraídos pasan a ser utilizados por lnkNet, el cual será el encargado de generar los modelos de Markov. La razón por la cual la función de recibir los datos y entrenar los modelos no estará implementada en CLAM está regida por la limitación temporal para desarrollar el proyecto, encontrando el programa lnkNet suficientemente competente para realizar dicha tarea, y asumiendo nuestra imposibilidad de alcanzar dicho nivel en un tiempo razonable.

Los Modelos de Markov resultantes de esta primera etapa, una vez entrenados con un número mínimo de señales que nos garantice la eficiencia de la representación, tienen el objetivo de ser capaces de describir una señal de una forma única. Por lo tanto tienen que, a partir del numero de modelos que queremos utilizar (512, 1024, 8048, etc.), ser repartidos con una distancia que garantice la representación de las características de la parametrización del audio.

Para su más fácil comprensión les plantearé un paralelismo con el lenguaje: cada modelo es una letra y cada señal de entrenamiento parametrizada una palabra; esta primera fase, recibiendo palabras que codifican nuestra realidad, tiene que ser capaz de generar un abecedario que sea capaz de representar las palabras sin equívocos. Las palabras, parámetros, nos servirán para definir el abecedario, modelos de Markov.

A partir de esta primera aproximación conceptual, podemos entender las diferencias entre crear 512 modelos o 8048, donde lógicamente será más eficaz temporalmente interpretar un abecedario de 512 letras que uno de 8048, pero a su vez el de 8048 es más seguro de garantizar la unicidad del vocabulario. En este caso se debe buscar un compromiso, teniendo en cuenta que la segunda fase no tiene restricciones de eficiencia temporal, mientras que la fase de reconocimiento, en el caso que nos ocupa, sí es necesario que sea en tiempo real.

El entrenamiento se realizará con un juego entradas que sea representativo del mundo que queremos ser capaces de describir.

Una vez cumplido el objetivo de la primera fase en la cual hemos generado los modelos de Markov, entramos en la segunda etapa cuyo objetivo es la generación de los descriptores para cada señal de entrada.

El funcionamiento inicial sigue el mismo esquema que el apartado anterior, extrayendo los parámetros del audio entrante que queremos representar. Una vez parametrizado, haremos funcionar un algoritmo que recibirá como entradas dicha parametrización y los modelos generados en la fase anterior, generando como salida los descriptores de la señal

analizada.

Haciendo uso del paralelismo anteriormente planteado, en este caso tenemos la realidad, la señal parametrizada y un abecedario, los modelos de la etapa anterior; de tal forma que el algoritmo encontrará cómo se escribe la realidad usando el abecedario que tenemos, es decir, como a través de la concatenación de cada modelo de Markov, cada letra, podemos construir un macro-modelo, la palabra, que defina la realidad, la señal de entrada.

Generaremos descriptores para todas las señales que querremos reconocer en el siguiente apartado.

Llegados a este punto, hemos obtenido de las anteriores fases como resultado por una parte los modelos de markov, el abecedario, y por otra los descriptores, el vocabulario.

A partir de ahora podemos empezar la tercera fase consistente en el reconocimiento de una señal de entrada parametrizada. Haremos funcionar un algoritmo que recibirá como entrada la parametrización, los modelos de Markov y los descriptores, sacando como resultado qué descriptor se ajusta mejor a la señal de entrada, a grandes rasgos, el algoritmo evaluará la probabilidad de cada modelo de Markov a través de la señal entrante y luego evaluará los descriptores viendo cual de ellos obtiene mayor puntuación dependiendo de su listado de modelos.

Volviendo a nuestro ejemplo, tendremos como entrada la realidad, la señal, el vocabulario, los descriptores, y el abecedario, los modelos de Markov. En este caso, el algoritmo buscará dentro de nuestro vocabulario qué palabra representa mejor la realidad que queremos definir, intentará discriminar cual es el nombre que define mejor nuestro objeto. De esta forma si uno entra un fragmento de una señal ya analizada, el sistema será capaz de reconocerla, es decir, aprendido un concepto por el sistema en la fase anterior, en esta fase, dado un objeto concreto, será capaz de discernir qué concepto está ejemplificando.

La primera y la segunda etapa se realizarían offline, es decir no tenemos el compromiso de tiempo de ejecución, mientras en la tercera se plantea ésta restricción, hecho que nos condicionará el modo de resolución del diseño de esta tercera fase.

Una vez aclaradas las funciones que debemos responder respecto AMADEUS, detectaremos los objetivos exigidos a CLAM.

En primer lugar, tenemos como objetivo la implementación de las etapas anteriormente citadas sobre el framework de CLAM, con lo cual deberemos identificar como crear la red de procesos que implementarán los diferentes algoritmos necesarios para proyectar la funcionalidad de AMADEUS. Por lo tanto subdividiremos el algoritmo en varios procesos que tengan semántica por si solos, y crearemos a su vez el proceso que esté compuesto por una red de procesos que se capaz de representar el algoritmo.

Una vez cumplida una primera aproximación, el segundo objetivo será disminuir la restricción de los casos de uso de AMADEUS, estudiando y efectuando los cambios oportunos para generalizar el método anteriormente descrito.

Los objetos de datos que definiremos tendrán que ser serializables en tiempo de ejecución creando estructuras de datos para almacenar como una base de datos los modelos de Markov, los descriptores y los ficheros de configuración necesarios, a su vez se estudiará la implementación de una especie de piscina de datos, donde uno pueda depositar de forma consistente los diferentes descriptores usados en un momento determinado, dándole cohesión y significado a un grupo de datos.

Respecto a los modelos de Markov generados por el lnkNet, su formato de salida sigue el estándar de HTK [35], por lo que implementaremos un parser que convierta dichos ficheros al estándar XML [38] utilizado para la serialización de datos en CLAM.

El funcionamiento de la tercera etapa tendrá que ser en tiempo real.

A continuación definiremos el significado de algunos términos que se irán repitiendo a lo largo del análisis y diseño del sistema.

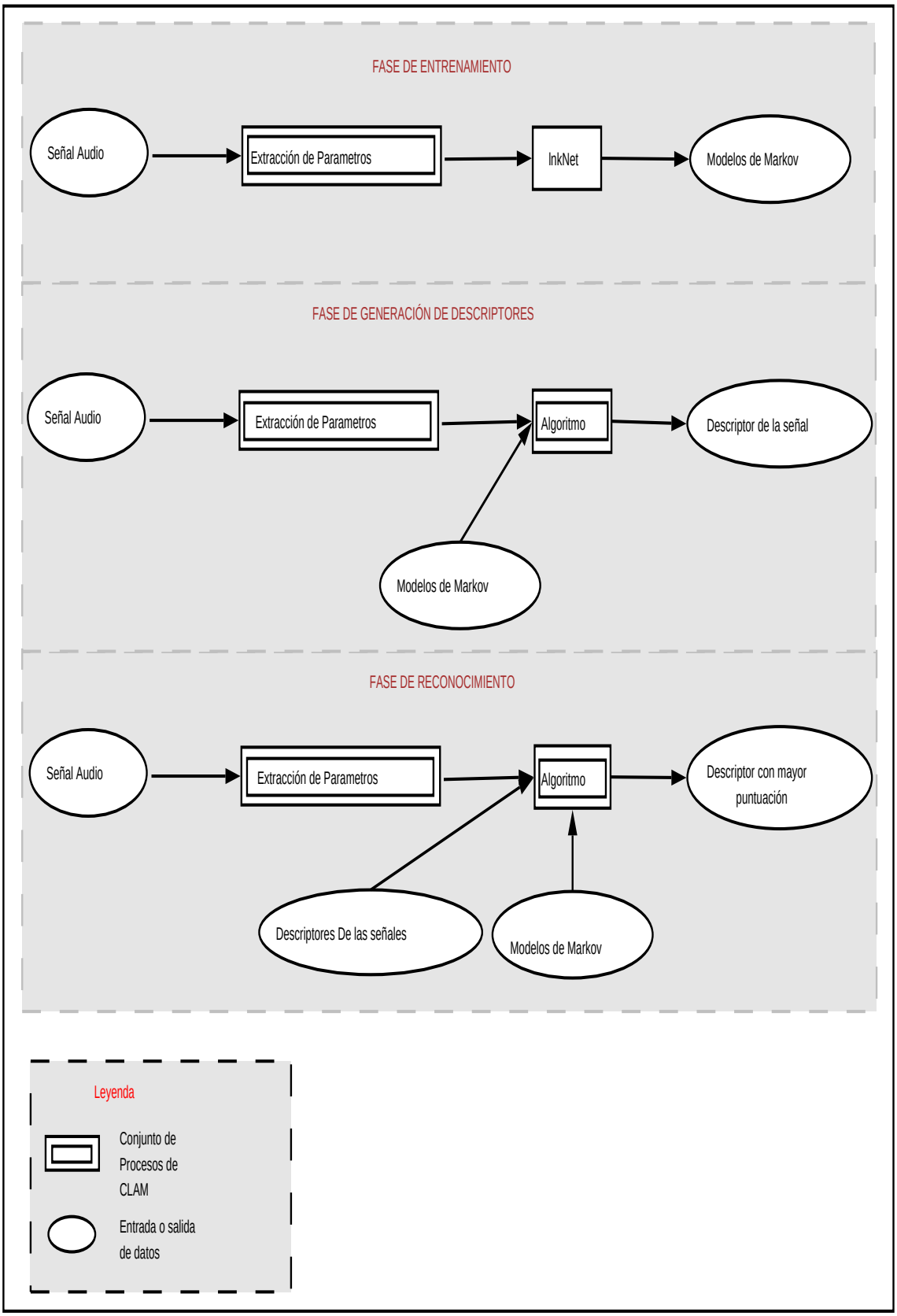

Figura 1.1: Visión General Funcional del sistema

Capítulo 2

Conceptos Introductorios

2.1. CLAM Process Network Model (DSPOOM)

El modelo conceptual, que usa a CLAM como demostración práctica, es la hipótesis de que cualquier sistema de procesado de señal puede ser modelado como un conjunto de objetos interrelacionados entre ellos. *Digital Signal Processing Object-Oriented Metamodel* (DSPOOM) es una abstracción de las ideas que se han ido encontrando mientras se desarrollaba CLAM.

El modelo de CLAM se puede entender como una infraestructura de objetos y canales a través de los cuales se interactúa. Podríamos clasificar la mayor parte de los objetos en tres clases: *Processing, ProcessingData* y *Network*. Estas tres clases son las que participan como pilares en la construcción de aplicaciones con CLAM, siendo modelos en CLAM de sistemas concretos. Son clases abstractas por los que no se las puede instanciar directamente, por lo que el modelo de CLAM se describe a través de instancias más específicas y el modo en el que interacción.

De esta forma podemos clasificar los objetos en cuatro categorías:

- Objetos que procesan (Processing).
- Objetos que contienen los datos necesarios para los procesos (ProcessingData).
- Objetos que conectan o sirven de interfaz (puertos, nodos o controles).
- Objetos concretos para la aplicación.

Las dos primeras categorías son las que cobran más importancia dentro del modelo mientras que las dos últimas son auxiliares pero a su vez imprescindibles para completar el modelo.

La principal idea consiste en separar objetos que encapsulen procesos (Processing) y objetos que sirvan a algún proceso encapsulando el tipo de los datos (ProcessingData).

El modelo DSPOOM podríamos verlo como un conjunto de objetos de procesado formando una red interconectada. Cada objeto de procesado podría recuperar símbolos de datos de procesado y modificarlos de acuerdo a algún algoritmo. El programador puede coger el control sobre el flujo de datos entre procesos o puede delegarlo a uno de los controladores automáticos de control de Flujo.

Cuando un conjunto de *Processing* forman un nuevo proceso, se puede observar el conjunto como una nueva abstracción formando un nuevo objeto de procesado. Así se puede considerar la modularidad y la escalabilidad. Diferentes objetos de procesado se pueden agrupar en tiempo de compilación como *ProcessingComposite* o en tiempo real como *Networks*.

DSPOOM es un metamodelo orientado a objetos, consecuencia de aplicar la practica orientada a objetos y los modelos computacionales gráficos relacionados con redes de procesos.

En la figura 2.1 se puede observar los distintos Elementos de DSPOOM. A continuación detallaremos más los conceptos de Processing, ProcessingData y las diferentes formas de componer los Processing, ya que estos son los conceptos principales que modelaran el diseño de los Modelos de Markov en CLAM.

2.1.1. Processing

Por definición, el acto de procesar se asocia con el de moldear a través de unos conceptos, ciertos objetos, en nuestro caso datos.

La clase *Processing* es una encapsulación abstracta de un proceso, siguiendo el paradigma orientado a objetos. A cada instancia de subclases de *Processing* la llamaremos objeto de procesado, al cual asociaremos una identidad, un comportamiento, una estructura y una secuencia de estados válidos.

El modelo del objeto de procesado contiene los principales bloques de la construcción de un sistema modelado según DSPOOM.

Mientras un objeto de procesado está funcionando, tiene dos tipos de salidas:

- Datos síncronos
- Controles asíncronos

En la figura 4.2 se ilustran los diferentes conceptos encapsulados dentro de la abstracción de *Processing*. Los componentes principales son la *configuración*, *los puertos* de entrada y salida de datos, *los controles* de entrada y salida, y el *algoritmo*.

A continuación profundizaremos en estos conceptos, pero antes quisiera resaltar, que como se puede observar en la figura existen dos tipos de flujos: de izquierda a derecha el *flujo* de datos y de arriba a abajo el flujo de controles. El flujo de datos es síncrono y se

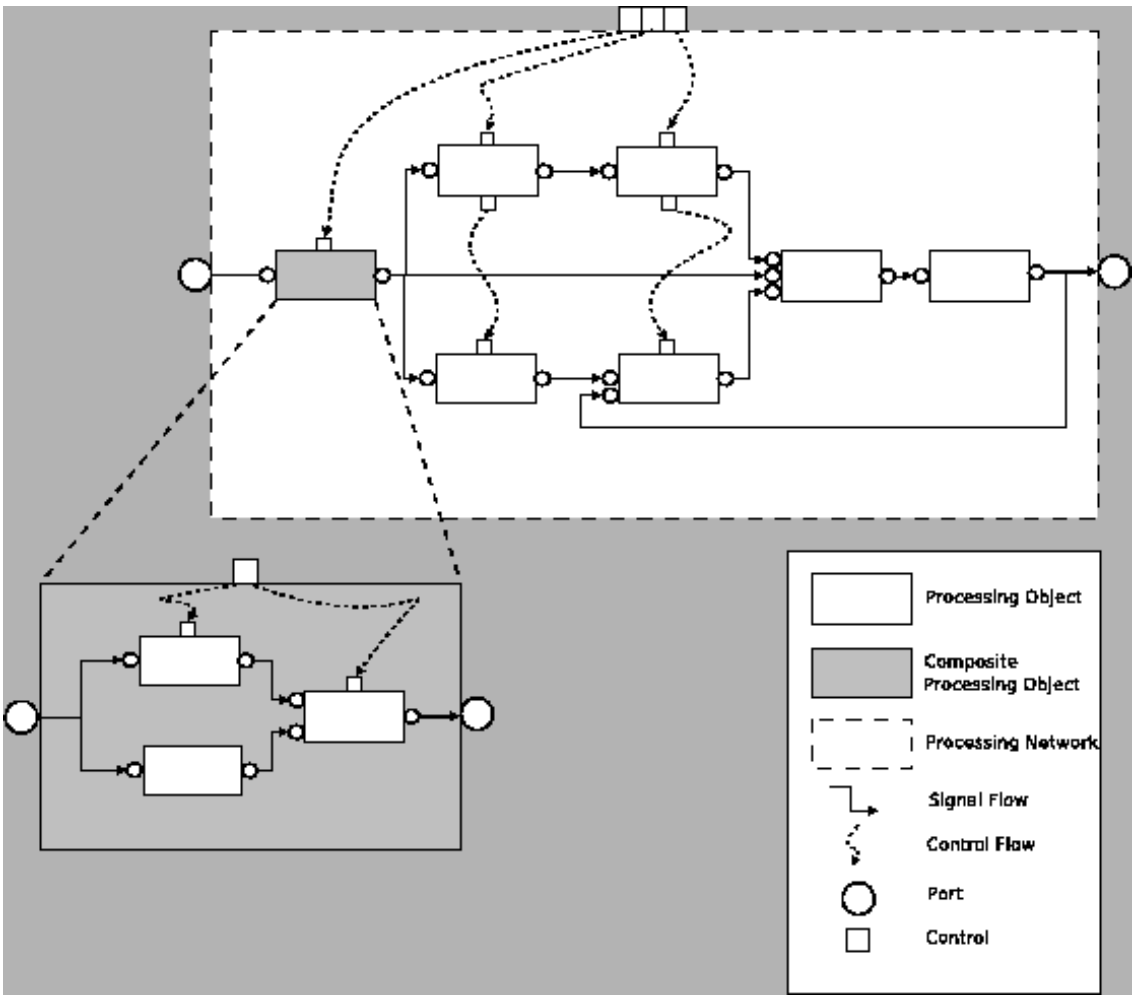

Figura 2.1: Elementos basicos de DSPOOM *(extraído de [3])*

controla a través de un reloj externo, mientras que el flujo de control es asíncrono y "even-driven". De esta forma el *flujo de datos* síncrono serán datos provenientes y dirigidos hacia los puertos, cada vez que un método Do sea llamado. Se consumirán los objetos de datos de los puertos de entrada y se producirán nuevos en los puertos de salida.

A su vez el *flujo de datos* asíncrono serán datos provenientes y dirigidos hacia los controles, cada vez que un evento de control suceda. Normalmente sirven para cambiar el estado o el modo de funcionamiento del algoritmo y no para transportar un gran peso de datos.

Cuando se dispara un proceso, se accede a los datos de entrada y usando un algoritmo se los transforma en datos de salida. El objeto de procesado puede acceder a datos externos a través de los conexiones entre puertos, por lo cual los puertos de entrada son el acceso

de entrada a los datos y los de salida sirven para enviar hacia fuera los datos.

Los puertos se pueden conectar entre ellos por parejas, aunque también se pueda conectar un puerto de entrada a más de uno de salida o viceversa, cumpliendo siempre la condición de que traten el mismo tipo de *ProcessingData*. Aún así existen los puertos polimórficos, pero no es recomendado su uso.

La ejecución de la funcionalidad de un objeto de procesado es lanzada por el envío del mensaje Do. Ésta es la única forma de acceder a la funcionalidad aunque los cambios dinámicos, cambios que no supongan variaciones en el estado interno del objeto, podrán ser realizados a través de los controles.

Ciclo de Vida

El estado de un objeto se define según los valores de sus atributos en un momento temporal determinado, siendo la secuencia de dichos estados el ciclo de vida de un objeto. No obstante, no todas las variaciones de los atributos producen un cambio de estado en el ciclo, por lo que tendremos que diferenciar entre los distintos niveles de importancia de los estados.

Definiremos el ciclo de vida de un objeto de procesado a través de los siguientes estados principales:

- *Unconfigured*, en donde el objeto estará esperando a ser configurado.
- *Ready*, en donde el objeto podrá ser reconfigurado o encendido, listo para la ejecución.
- *Running*, mientras el objeto haya sido configurado y encendido, podrá ejecutarse, para finalmente apagarse y poder reiniciar el ciclo de vida.

Como se ilustra en la figura 2.2, cuando el objeto está en el estado *Unconfigured*, y realizamos la tentativa de configurarlo, dicha acción, en el caso de que el valor de la variable *ConcreteConfigure* retornada sea *veradero*, transitaremos al estado *Ready* sabiendo que la configuración se ha podido realizar satisfactoriamente; en caso contrario permaneceremos en el estado *Unconfigured* de tal forma que si uno intenta realizar las acciones *Do, Start o Stop*, recibiremos un mensaje del controlador de errores para hacernos concientes de la situación excepcional del objeto. Las configuraciones concretas de las distintas implementaciones de objetos de procesado se realizaran a través de un método abstracto puro, obligatorio de implementar por las subclases de *Processing*, llamado *ConcreteConfigure*.

El control del flujo de los estados está ya implementado en la clase padre *Processing*, por lo que el programador tendrá que cuidar los aspectos concretos de las acciones relativas a cada objeto de procesado distinto.

Estando el objeto en *Ready*, tendremos la opción de reconfigurarlo ateniéndonos a los posibles cambios mencionados anteriormente según la respuesta de dicha acción, y

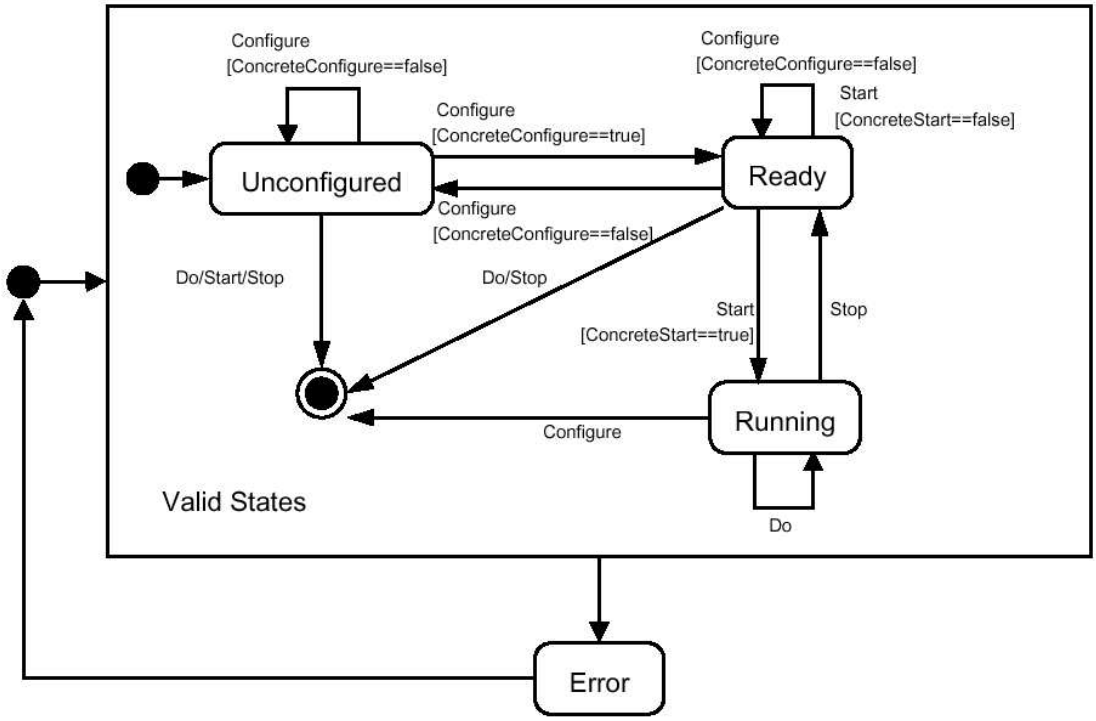

Figura 2.2: Diagrama de estados de Processing *(extraído de [3])*

podremos también realizar el encendido, que también vendrá a ser controlado por el valor de una variable que nos servirá para saber si se ha podido realizar correctamente o no, pasando al estado *Running* o quedándonos en el mismo estado *Ready*. Las acciones *Do o Stop*, nos llevarían a un error.

Una vez el objeto está en *Running*, Se puede llamar a Do para ejecutar la funcionalidad del proceso tantas veces como uno quiera. Cuando uno considera que no va a utilizarlo más, hace la acción de apagar, lo que nos traslada al estado *Ready*, teniendo que encenderlo de nuevo si quisiéramos usar de nuevo su funcionalidad. Si estando en el estado *Running* intentamos configurar, nos llevará a un error. En este caso si queremos realizar cambios sobre el objeto de procesado se usarán los *controles*.

Configuración (*ProcessingConfig*)

ProcessingConfig es la clase que define la abstracción de cómo debe ser un objeto de configuración, conteniendo los valores necesarios para las variables no ejecutables (todas aquellas que no varían en el estado *running*) de los objetos de procesado.

Normalmente las variables que configura suelen estar relacionadas con la estructura

del algoritmo, como pueden ser elección de estrategia, inicialización de tablas, tamaño de los datos, etc.

De esta forma nos serviremos de *ProcessingConfig* para realizar las configuraciones iniciales, usando los controles para la configuración de parámetros en estado de ejecución.

Estereotipos de *Processing*

Se pueden clasificar los distintos *Processing* dentro de tres estereotipos:

- Generadores
- Sumideros
- Transformadores

Los objetos generadores son aquellos que generan datos y no consumen, es decir, todos aquellos que no tienen puertos de entrada. A la llamada de ejecución generarán los datos según su estado interno, la configuración de variables. En estado de ejecución cobran importancia los controles, puesto que son los únicos capaces de cambiar los datos que se generen, dependiendo de la topología del objeto, puesto que algoritmos como generadores de valores aleatorios no repetirán su resultado.

Los sumideros en cambio son aquellos objetos que consumen datos sin generar ninguno, es decir, tienen puertos de entrada pero no de salida. Los principales usos que se dan a éste estereotipo es el de traducir los datos a un formato para poder comunicarse con otro sistema o con usuarios, como pueden ser los pasivadores, visualizadores, etc.

Los transformadores son aquellos que tienen puertos de entrada y salida, en donde el tipo de datos que entra es diferente al que sale. Por ejemplo un banco de filtros, recibe un espectro y devuelve un vector.

En la figura 2.3 podemos observar un ejemplo generadores, sumideros y transformadores.

2.1.2. Processing Data

Los objetos de procesado (*Processing*) que anteriormente hemos descrito, tan solo pueden consumir *Processing Data*.

La abstracción del objeto de datos tiene que ofrecer los servicios de:

- *Introspección*, capacidad para describirse a sí mismo, es decir, poder decir cuantos atributos posee, los tipos, etc.

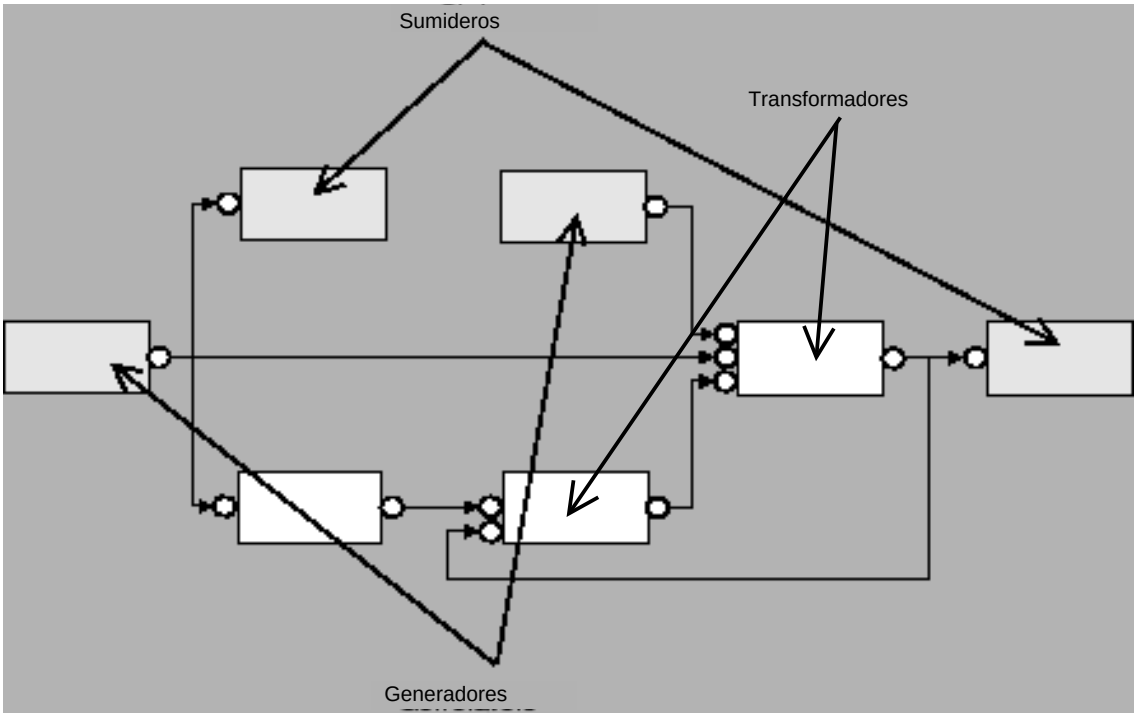

Figura 2.3: Generadores, Sumideros y Transformadores *(extraído de [3])*

- *Una Interfaz Homogénea*, para facilitar el acceso transparente a los datos sin necesidad de tener conocimiento sobre las subclases de los datos.
- *Encapsulación*, donde los datos sean protegidos y solamente accedidos a través de "getters" y "setters".
- *Persistencia*, teniendo un sistema automático de construcción hacia un formato apropiado.
- *Visualizadores*, facilitando la observación de los datos en tiempo de debug.
- *Composición*, permitiendo que atributos de un objeto de datos sean otros *Processing*

Data.

Los tipos de atributos se dividen en:

- Atributos de datos, que actúan como container de datos.
- Atributos de valor, actúan como información auxiliar relacionada con los atributos de datos.

A su vez los atributos de valor se pueden dividir en:

- Informativos, valores que sirven para interpretar el contenido del *Processing Data* y cuya modificación no implica cambios en los datos.
- Estrucutrales, valores que también son informativos pero modifican también la estructura interna del objeto. Así sucede por ejemplo con el atributo "size" de la FFT, que debe ser ajustado coherentemente con los distintos buffers internos.

Otro aspecto interesante son los puntos de vista *dinámico* y *estático* que se pueden tener sobre el flujo de datos.

De tal forma podemos observar los objetos de datos como entidades que viajan, entrando por los puertos de los objetos de procesado, siendo transformados y saliendo por los puertos de salida. Esta visión *dinámica* sirve para observar la latencia de los sistemas o aspectos en tiempo real.

Por otra parte está la visión *estática*, que observa los objetos de datos como nodos fijos a donde se conectan los objetos de procesado. Los objetos de datos están siempre en la misma posición de memoria y los puertos apuntan a ella, por lo que podríamos interpretar los puertos como objetos de procesado estático, siempre en la misma posición, que van cambiando constantemente de valor. Es interesante éste punto de vista para estudiar los problemas de accesos simultáneos, etc.

Respecto a otras formas de encapsular datos como puede ser los controles, es importante saber que podemos modelarlo solo si son tipos básicos, no podemos modelar como controles estructuras complejas.

También debemos resaltar los *Descriptores* como un caso particular de *Processing Data*. Son obtenidos a través de procesos de extracción a partir de otros objetos de datos u otros descriptores. Suelen ser metadatos describiendo aspectos de los datos originales.

Para poder distinguir cuándo los datos se pueden describir como un *Descriptor*, diremos que para ser un descriptor tiene que sufrir procesos estadísticos para transformar datos, y que no sea de fácil regresión a los originales, como puede pasar con el espectro. Los objetos *Extractores* realizan un proceso estadístico que producen descriptores.

Volveremos a tratar éste tema en el capítulo de diseño, puesto que Amadeus tiene un componente importante de extractores de parámetros y descriptores resultantes.

2.1.3. Composición de Redes de Procesos (*Process Network*)

Dentro del metamodelo existen diversas formas de componer los objetos de procesado.

Los dos arquetipos para componer los objetos de procesado son :

- *Network*, una composición dinámica que puede ser modificada en tiempo real añadiendo nuevos procesos o modificando las conexiones.
- *ProessingComposite*, una composición estática construida en tiempo de compilación sin poder ser modificada posteriormente.

Ambos métodos aportan una mayor complejidad y ocultamiento, sin embargo la posibilidad de construir un control de flujo automático es propio del modelo *Network*, mientras que *ProcessingComposite* es más eficiente y optimizado.

ProcessingComposite

Modelo de composición estática, donde los ProcessingComposite son implementados manualmente por el desarrollador con la posibilidad de ser altamente ajustables para buscar la eficiencia en casos de uso específicos. En tiempo real no pueden ser añadidos más objetos de procesado ni variar sus conexiones.

Como se puede observar en la figura 2.4,un ProcessingComposite tiene un padre y un número de hijos que son sus componentes. Pueden haber diferentes niveles de composición, objetos que sean hijos y padres a la vez, es decir, que pertenezcan a un objeto padre Composite y que a su vez también sean objetos Composite, subsitemas.

Realizar una operación sobre un objeto Composite, implica realizar la misma operación a lo largo de todos sus objetos de procesado recursivamente hasta el último subsitema. De tal forma, crear un ProcessingComposite implica crear todos los objetos de procesado hijos, así como configurar, encender, apagar y ejecutar.

Los objetos de procesado hijos se tratarán como atributos del objeto padre, siendo construidos en el constructor como cualquier otro atributo. Serán añadidos a la lista de hijos para así saber cuál es el padre en cada momento.

En lo que respecta a la configuración, el padre tendrá que contener todos los parámetros necesarios para configurar todos sus hijos, de tal forma que las dependencias y el diseño de decisiones no podrán ser automatizados. El desarrollador deberá elegir qué parámetros de los hijos afectarán al padre o a otros hijos. Por ejemplo, cuando configuremos la frecuencia de muestreo del padre, tendremos que preocuparnos de mantener la coherencia y que todos los hijos sean también configurados.

El encendido y apagado del ProcessingComposite está automatizado sin implicar ninguna complejidad de cara al desarrollador.

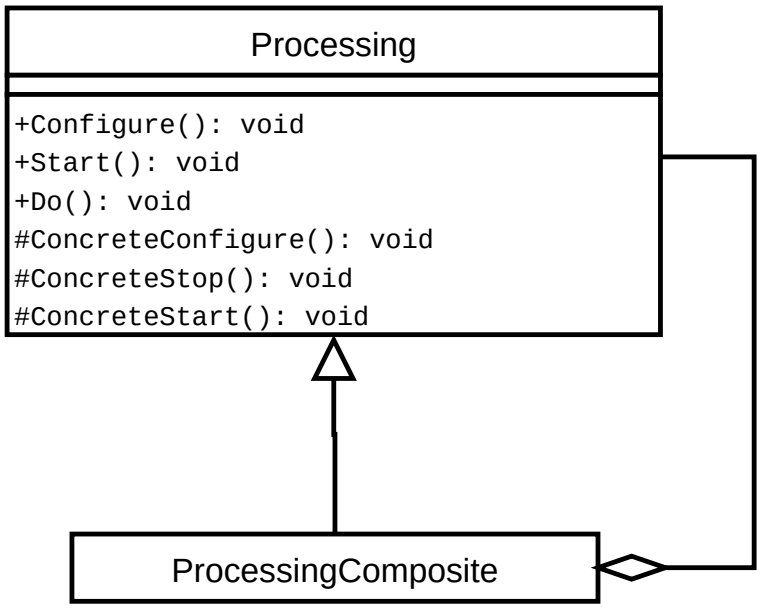

Figura 2.4: Diagrama de clases de ProcessingComposite

Respecto a la ejecución, el diseñador es el que decide el orden de ejecución y los datos entrantes y salientes de cada objeto puesto que no existe ningún proceso de automatización. En este aspecto es donde reside la capacidad de eficiencia, puesto que uno puede optimizar el funcionamiento según el caso de uso.

Network

Modelo de composición dinámica, definido por un conjunto de objetos de procesado interconectados entre ellos con un mismo objetivo, tolerando modificaciones en tiempo real. La figura 2.5, ilustra el Network entendido como un conjunto de processings con sus correspondientes puertos y controles.

Podríamos definir Network como una composición dinámica de objetos de procesado. El soporte de la funcionalidad en tiempo real la proporciona la clase Network, a través de dos atributos claves:

- Un Mapa de objetos de procesado. El Mapa es un tipo estándar ofrecido por la stl [15], conteniendo un vector de punteros para instanciar los objetos junto a un nombre que actúa como clave única.
- Un Flujo de Control asociado, ya que la red puede tomar diferentes ejecuciones según la política elegida. Dichas políticas están encapsuladas dentro de la clase *FlowCon-*

trol.

Para añadir nuevos objetos de procesado, éstos se crean a través de un porceso *Factory*, respetando el patrón de diseño *factory method* [12], el cual a su vez asigna la clave única para luego poder acceder a través de ella a los distintos objetos.

Los puertos y controles también poseen una clave única (string) para poder distinguirlos. Esta clave se forma juntando la clave única del objeto de procesado al que pertenece más el nombre del puerto. La cantidad de datos que consume o produce un puerto en una ejecución vendrá fijada por la variable *region o window size*.

La interfaz de uso que ofrece Network es similar al uso de Processing Composite, Start, Stop, etc. Tan solo varía el Do, que en este caso la llamada sería DoProcessings, que ejecutará los distintos Do respetando la política aplicada.

Las distintas políticas, son conjuntos de decisiones sobre el modo de actuar en cada disparo de ejecución.

Network es más flexible que ProcessingComposite, ya que las decisiones no se toman en tiempo de compilación, siendo ésta una de las razones de su menor eficiencia.

De esta forma, igual que podríamos observar un ProcessingComposite como un solo objeto de procesado, también ocurre lo mismo con una Network, pudiendo clasificarla también como Proceso generador, sumidero o transformador.

Nodo de Datos

Como se puede observar en la figura 2.6, un nodo de datos se define donde haya agrupación de arcos para crear un solo puerto en común, con política FIFO (first in, first out). De esta forma se evita tener réplicas innecesarias adjuntando un poco más de complejidad en el control de flujos, de tal forma que garantice la consistencia de los datos. Aún asi la interfaz será transparente y sin necesidad de preocuparse por los detalles de bajo nivel.

En un nodo de datos puede haber sólo un objeto *productor* de datos y el resto de los conectados *consumidores*, es decir, un puerto de entrada y varios de salida.

A demás de ser un nodo, actúa como contenedor de datos implementado como una cola FIFO. La cola es circular, con diversos punteros o regiones de lectura y uno solo de escritura. Para garantizar la consistencia, siempre el área de lectura será "legible" sino se están escribiendo datos; y el área de escritura avanzará las zonas que están siendo leídas sin modificarlas y sobrescribirá sobre las zonas donde no se estén leyendo los datos.

2.1.4. Modelo Computacional Gráfico

Al aplicar técnicas de modelado orientado a objetos, para el procesado de señal, es fácil que nos resulte como consecuencia un modelo computacional gráfico.

Antes de definir el de CLAM, introduciré los conceptos básicos relacionados sobre qué es un modelo gráfico computacional.

Un modelo gráfico computacional, que para abreviar lo llamaremos (*MoC*), es una abstracción de una familia de sistemas basados en la computación. La visión abstracta sirve para modelar diferentes sistemas, ayudando a comprender el diseño y sus propiedades. La elección del tipo de MoC viene dada por el dominio de la aplicación. Por ejemplo, las aplicaciones de procesado de señal digital (DSP) generalmente aprovechan los modelos llamados *Dataflow*.

Los distintos paradigmas de programación pueden representarse a través de los modelos computacionales.

Cuando hablamos de MoC gráficos, entendemos por gráfico que el sistema puede ser modelado a través de un grafo, formado por nodos y arcos 2.7. Las diferencias entre diversos MoC se deben a la asignación de una semántica concreta a arcos y nodos; y a las restricciones sobre la estructura del grafo.

Los principales modelos computacionales gráficos son:

- *Queueing Models*, donde los nodos representan complejos operadores, como puede ser Poisson queues, o nodos de decisión y los arcos representan eventos, símbolos o peticiones. Se usa para aplicaciones que realicen estimaciones.

- *Finite State Machines*, se usan especialmente en tareas de control intensivo y protocolos. Está compuesto por entradas, salidas, estados, estado inicial, siguiente estado y producciones. No es un modelo de Turing completo.

- *State Chart*, tiene estados, eventos, condiciones y acciones. Los eventos y las condiciones causan transiciones, que son composiciones de estados.

- *Petri Nets*, sirve para estudiar sistemas con concurrencia, asíncronos, distribuidos, no determinísticos y con características de sistemas en paralelo. Contiene *Lugares*, *transiciones* y arcos que los conectan entre ellos. Es ejecutado por las reglas de disparo que transmiten los símbolos de un lugar a otro y donde cada disparo es permitido cuando cada lugar de entrada tiene un símbolo dentro.

- *Process Networks or Kahn Process Network*, es un modelo de computación concurrente de subconjuntos de modelos Dataflow, es decir, es un grafo dirigido donde cada arco representa una cola FIFO para comunicarse y cada nodo representa un proceso independiente y concurrente.

- *Dataflow Networks*, es un caso especial de Process Network donde los nodos son actores que responden a reglas de disparo de ejecución.

A continuación haremos una proyección de los conceptos del modelo de CLAM y los modelos gráficos computacionales.

En primer lugar, en el modelo gráfico computacional, los nodos principales del grafo corresponderían a los objetos de procesado. Los arcos se definirían conectando los puertos entre los distintos objetos de procesado, interpretando dichos arcos como colas FIFO sin límite donde los datos se escriben y leen.

Los objetos de procesado producen y consumen de las colas de forma síncrona. El número de datos que procesan no tiene un ritmo constante, la intensidad de actividad varía a lo largo del tiempo. Con la conexión de los controles definimos arcos secundarios que funcionan con el mecanismo "event-driven" y al no tener una cola si los datos no han sido leídos se pierden. En las reglas de disparo de *Dataflow Networks* se suele especificar la cantidad de datos a consumir por disparo, hecho que también se realiza en DSPOOM configurando el *window size* del puerto 2.1.3. Al poder variarse dicho tamaño en tiempo real, podríamos decir que el metamodelo se asimila a *Dynamic Dataflow Networks*, o en caso de ser estático dicho tamaño se asimila a *Synchronous Dataflow Networks*.

Sin embargo el aspecto asíncrono que agregan los controles, es parecido a la extensión de *Khan Process Networks*, llamada *Context-Aware Process Network*, que agrega el concepto de comunicación asíncrona de informatición. *Dataflow Networks* tambíen és una variación de *Khan Process Networks*, por lo que podríamos concluir que DSPOOM se relaciona con el MoC *Context-Aware (Dynamic o no) Dataflow Network*.

Aún así el concepto de nodo de datos 2.1.3, nos podría acercar al modelo *Petri Net*, donde los nodos de datos podrían ser los *lugares* y los objetos de procesado las *transiciones*. Otra interpretación sería pensar en los nodos de datos como Nodos secundarios dentro de lo definido anteriormente.

2.2. Modelo Probabilísitco

Cada vez que realizamos un cálculo matemático para resolver un problema, lo que estamos haciendo es aplicar un modelo matemático a un fenómeno de la realidad.

Al enfrentar un problema de ingeniería o de algún otro tipo, estamos analizando e investigando una parte o aspecto de la realidad material que nos rodea. Para resolver el problema, necesitamos modelar esa realidad, es decir, construir una representación de cómo ocurren los hechos, utilizando la matemática para calcular los efectos de los mismos.

Un modelo es sólo una representación de la realidad, utilizado para estudiar y analizar dicha realidad.

Los modelos matemáticos que después de efectuar los cálculos nos dan un resultado numérico preciso se denominan Determinísticos. Sin embargo hay fenómenos que necesitan otro tipo de modelos matemáticos, que se denominan no determinísticos, probabilísticos o estocásticos.

Antes de embarcar el estudio de los Modelos Ocultos de Markov, modelo probabilístico en el cual nos hemos centrado en la parte práctica y teórica de éste proyecto, me gustaría

realizar una introducción sobre las motivaciones del estudio del modelo probabilístico.

Primero quisiera destacar la capacidad universal de la teoría probabilística para representar realidades físicas, mientras que otros modelos sirven para casos muy concretos. También cabe observar la indispensabilidad de los conceptos probabilísticos en diferentes escenarios normales de la ingeniería.

En las líneas generales de la física moderna que estudia el conocimiento sobre la naturaleza de la realidad, se está haciendo insostenible el punto de vista de una naturaleza determinista y exacta. En el mejor de los casos existe una representación aproximada de dicha naturaleza y aquí es donde la teoría probabilística cobra importancia gracias a su capacidad para describir situaciones de incertidumbre. El mundo real se suele razonar con información incierta e imprecisa.

A principios de 1980, Judea Peral [21] retorna la importancia del modelo probabilístico creando las redes bayesianas. Es un modelo probabilístico inspirado en la causalidad, asociándole un modelo gráfico, cuyos nodos representan variables y cuyos arcos representan mecanismos causales. Estos conceptos intervienen en los modelos markovianos.

2.3. Modelos ocultos de Markov

Los Modelos de Markov (HMM) nacieron como una herramienta estadística para modelar series temporales discretas, encontrándole aplicaciones muy potentes en las áreas de biología molecular y reconocimiento de señal.

Un HMM es un modelo probabilístico temporal en donde cada estado del proceso se describe por una o más variables aleatorias discretas. Los posibles valores de las variables son los posibles estados del mundo que desea representar. Cada instante discreto de tiempo, el proceso asume estar en un estado y al recibir las observaciones el proceso cambia de estado según su matriz probabilística de transición.

Un HMM, se define a través de los siguientes parámetros (π, A, B):

- π es el vector de distribución del estado inicial, siendo π_i la probabilidad del estado i en tiempo $= 0$.
- A es la matriz de transición de estados, donde $A = [a_{ij}]$, siendo a_{ij} la probabilidad de transición del estado actual i al estado j.
- B es la matriz de distribución de salida, donde $B = [b_{jk}]$, siendo b_{jk} la probabilidad de observar el símbolo k dado el estado actual j.

Una representación gráfica posible de un modelo sería por ejemplo la propuesta en la figura 2.8.

2.3.1. Primera Aproximación

Para aclarar los conceptos, empecemos por proponer un proceso de Markov discreto compuesto por tres estados que representen el estado de ánimo. Para simplificar, pondremos la convención de que existen tres estados de ánimo:

- Estado 1: Triste
- Estado 2: Normal
- Estado 3 : alegre

Proponemos que el estado de ánimo en un momento temporal t se caracteriza por uno de los tres estados y que su matriz A de probabilidad de transición de estados es:

$$\mathbf{A} = \begin{pmatrix} 0{,}4 & 0{,}4 & 0{,}2 \\ 0{,}2 & 0{,}6 & 0{,}2 \\ 0{,}2 & 0{,}4 & 0{,}4 \end{pmatrix}$$

Dado que en el dia 1 ($t = 1$) el estado de ánimo es alegre (estado 3), podemos preguntarnos cuál será la probabilidad, de acuerdo al modelo propuesto, de que el estado de ánimo de los siguientes cinco días sea:

- "Alegre-alegre-normal-alegre-triste-triste"

Más formalmente, podemos definir la secuencia de observación O como $O = \{S_3, S_3, S_2, S_3, S_1, S_1\}$ correspondiente a $t = 1, 2 \dots 6$, teniendo que determinar la probabilidad de O dado el modelo anterior. Esta probabilidad se podría evaluar de la siguiente manera:

$$\begin{aligned} P(O|Modelo) &= P[S_3, S_3, S_2, S_3, S_1, S_1|Modelo] \\ &= P[S_3] * P[S_3|S_3] * P[S_2|S_3] * P[S_3|S_2] * P[S_1|S_3] * P[S_1|S_1] \\ &= \pi_3 * a_{33} * a_{32} * a_{23} * a_{31} * a_{11} \\ &= 1 * (0{,}4) * (0{,}4) * (0{,}2) * (0{,}2) * (0{,}4) \\ &= 2,56 * 10^{-3} \end{aligned}$$

Otra pregunta interesante que se podría hacer al modelo sería: dado que el modelo está en un estado conocido, ¿cuál es la probabilidad de que se quede exactamente d días en el mismo estado? Esta probabilidad se podría evaluar a través de la siguiente secuencia de observaciones:

$$O = \{S_1, S2, \dots Sd, Sd + 1 \neq Sd\} \qquad \text{donde } S_i = S_d,\ i = 1 \dots d \tag{2.1}$$

Dado un modelo la probabilidad sería:

$$P(O|Model) = (a_{ii})^{d-1} * (1 - a_{ii}) = P_i(d) \tag{2.2}$$

El valor $P_i(d)$ es la función probabilística de densidad (discreta) de duración d en el estado i.

El desarrollo matemático, evaluando con el modelo dado anteriormente, nos resulta que estaría triste 1.67 días seguidos, 2.5 días normal y 1.67 alegre.

2.3.2. Observaciones ocultas

En el anterior ejemplo hemos asociado a cada estado un evento observable, lo cual restringe mucho la aplicación de dichos modelos para una gran cantidad de problemas, ya que en la realidad pocas veces tenemos los eventos observables.

Vamos a tratar lo que sucede cuando los eventos son ocultos (*hidden*) por lo que deberemos representar la observación como una función probabilística de estado.

De esta forma un modelo resultante que tiene un doble proceso estocástico, uno incrustado dentro del otro, por lo que uno de ellos será subyacente, estará debajo del otro y no será observable, será oculto. Solo será observable a través de otro conjunto de procesos estocásticos capaz de producir observaciones.

Dado un escenario, donde los datos no sean observables, el problema interesante consistiría en cómo construiríamos el HMM para modelar las observaciones ocultas.

El primer rasgo a considerar sería decidir qué representan los estados del modelo y *decidir cuántos* estados tendría nuestro modelo.

El diseño del modelo es muy importante para el funcionamiento del sistema, por ejemplo, si el escenario que queremos modelar consiste en una persona situada en otra habitación no visible para nosotros que tira una moneda x veces, y queremos construir un modelo que represente la secuencia de caras y cruces podríamos proponer:

- un modelo de dos estados donde cada estado representa cada lado de la moneda, uno cara y el otro cruz.
- Un modelo de dos estados donde cada estado estaría caracterizado por una función probabilística de densidad.
- Un modelo de tres estados con una función que caracterice cada estado a través de algún evento probabilístico.

El primer modelo propuesto sería correcto en el caso de que las observaciones no sean ocultas teniendo tan solo un parámetro desconocido pero observable. En el segundo y el tercero el número de parámetros a determinar sería de cuatro y nueve respectivamente.

Cuánto más grado de libertad haya, es decir, más parámetros a determinar, el grado de libertad será mayor y el HMM será más capaz de modelar un escenario.

Sin embargo en la práctica, no ocurre así y existen grandes limitaciones respecto al tamaño de los modelos. En el ejemplo concreto propuesto, el tercer modelo sería inapropiado ya que no corresponde con los eventos físicos a modelar.

También se debe considerar otros factores importantes relacionados con el tamaño del modelo como pueden ser el costo computacional, la necesidad de realizar cálculos en tiempo real, etc.

2.3.3. Elementos de un HMM

Por lo tanto, un HMM va a estar caracterizado por los siguientes parámetros:

- N, que será el número de estados del modelo. Aunque los estados sean ocultos, para muchas aplicaciones prácticas hay un significado físico asociado a los estados o conjuntos de estados. Dichos estados están conectados entre ellos, habiendo así diversos modos de interconexión. Los estados se denotan individualmente como $S = \{S_1, S_2 \dots S_N\}$, mientras que el estado en el tiempo t se denota como q_t.
- M, que será el número de símbolos de observación distintos por estado. El símbolo de observación corresponde a la salida física del sistema modelado, por ejemplo en el escenario de las monedas sería cara o cruz. Se denota como $V = \{V_1, V_2, \dots, V_M\}$.
- La distribución de transición de estados $A = \{a_{ij}\}$ donde

$$A_{ij} = P[q_{t+1} = S_j | q_t = S_i], \qquad 1 \leq i,\ j \leq N \tag{2.3}$$

  Para los modelos en que desde cada estado se pueda ir a todos los demás estados, el valor de a será $a_{ij} > 0\ \forall i, j$, mientras que en otras tipologías de HMMs tendremos algunos $a_{ij} = 0$.

- La distribución de probabilidad de los símbolos de observación en un estado j, $B = \{b_j(k)\}$ donde

$$B_j(k) = P[v_k \text{ en el momento } t | q_t = S_j], \quad 1 \leq j \leq N \tag{2.4}$$
$$1 \leq k \leq M \tag{2.5}$$

- La distribución del estado inicial $P_i = \{\pi_i\}$, donde

$$\pi_i = P[q_1 = S_i], \qquad 1 \leq i \leq N \tag{2.6}$$

Dando los valores apropiados a N, M, A, B y π, el HMM se podrá usar como generador para dar una secuencia de obsrvaciones $O = O_1, O_2, \ldots O_t$

Donde cada observación O_t es un de los símbolos de V, siendo T el número de observaciones en esta secuencia. El procedimiento sería el siguiente:

1. Elegir el estado inicial $q_1 = S_i$ de acuerdo a la distribución del estado inicial π.
2. Poner $t = 1$.
3. Adjudica $O_t = v_k$ de acuerdo a la distribución de probabilidad de los símbolos de observación en el estado S_i, es decir $b_i(k)$.
4. Realizar la transición al nuevo estado $q_{t+1} = S_j$ de acuerdo a la distribución de transición de estados en el estado S_i, es decir a_{ij}.
5. Poner $t = t + 1$, y volver al paso *3)* si $t < T$; sino terminar el proceso.

Por lo tanto podemos observar que para la especificación de un HMM es necesario especificar los parámetros M y N, y las medidas probabilísticas A, B y π. Usaremos

$$\lambda = (A, B, \pi) \tag{2.7}$$

Para indicar los parámetros que configuran el modelo.

2.3.4. Propiedades

Un proceso estocástico $(X_n, n = 0, 1, 2, \ldots)$ se denomina una cadena de markov en tiempo discreto si cumple con las propiedades markovianas y de estacionareidad.

Markoviana

Sabemos que para conocer la distribución de X_{n+1} se necesita conocer X_n. Razonando, se ve que X_n depende a su vez de X_{n-1}, X_{n-1} a su vez depende de X_{n-2}, etc. Luego, se concluye que X_{n+1} depende no solo de X_n, sino que indirectamente depende también de X_{n-1}, X_{n-2},...,X_1,X_0.

El proceso estocástico $(X_n, n = 0, 1, 2, \ldots)$ está conformado por una familia de variables aleatorias independientes entre sí. Sin embargo, la dependencia entre estas variables aleatorias es tal que, si yo conozco el valor de X_n, los valores de $X_0, X_1, X_2, \ldots, X_{n-1}$ son irrelevantes para estudiar el valor de X_{n+1}. Podemos observar que esto no quiere decir que X_{n+1} sea independiente de $X_0, X_1, \ldots, X_{n-1}$, sino que lo que ocurre es que estas variables influyen en X_{n+1} solo a través de X_n.

Más exactamente, si llamamos al periodo $n+1$ como el futuro, al periodo n como el presente, y los periodos $0, 1, 2, ..., n-1$ como el pasado, se podrá afirmar entonces que "el pasado influye sobre el futuro sólo a través del presente". Esta propiedad es la denominada markoviana, su definición formal queda expresada en el teorema 1.

Teorema 1 *Sea Xn una variable aleatoria que toma valores enteros, Entonces diremos que el proceso (Xn, n=0,1,2,...) cumple con la propiedad markoviana si* $P_t[X_{n+1} = j | X_{n=i}, X_0 = i_0, X_1 = i_1 \ldots X_{n-1} = i_{n-1}] = P_t[X_{n+1} = j | X_{n=i}]$

Estacionareidad

Por otra parte el proceso $(X_n, n = 0, 1, 2, \ldots)$ cumple con la propiedad de estacionareidad si la propiedad $P_r(X_{n+1} = j | X_n = i)$ depende sólo de i y de j, pero no de n. En este caso, definimos como $P_{i,j}$ a esta probabilidad, y se denomina probabilidad de transición en una etapa.

La propiedad de *estacionareidad* establece que los mecanismos probabilísticos que definen la evolución del proceso no cambian con el tiempo. Las probabilidades $P_{i,j}$ pueden agruparse en una matriz P, denominada matriz de probabilidades de transición en una etapa, que tendrá tantas filas y columnas como estados tenga el proceso.

Transiente, reccurrente e irreductible

En las cadenas de markov en tiempo discreto se pueden estudiar los procesos a largo plazo, dentro del cual, lo que se puede analizar es si estos tienen distribución límite o no.

En la medida que esto ocurra, podemos concluir que, pasado mucho tiempo, la distribución de probabilidades del proceso no cambiará de una etapa a otra.

En un largo plazo los estados del proceso se pueden clasificar de diversas maneras, de tal forma que estas clasificaciones formalicen las existencias de las distribuciones límites.

Un estado se denominará *transiente* si $F(i, i) < 1$, es decir, si un estado parte en i, la probabilidad de que regrese al mismo estado es menor que uno. También los estados se denominarán recurrentes si $F(i, i) = 1$, y si su valor esperado de tiempo es menor a infinito $(E(T(i, i)) < \infty)$ se denominará *recurrente positivo*, en caso contrario, se denominará *recurrente nulo*. En el caso de un estado recurrente nulo, existe la seguridad de que, si el proceso parte en el estado i, volverá a ese estado alguna vez, sin embargo, el tiempo promedio de retorno será infinito.

La comunicación es un ámbito importante en el estudio de los estados a largo plazo, debido a que los determina a estos según clases. Por convención se dice que un estado siempre se comunica consigo mismo, ya que:

$$P_{i,j}^{(0)} = P_r[X_0 = i | X_0 = i] = 1, \qquad \text{es decir, se comunica a través de un camino de largo 0} \tag{2.8}$$

Si las clases de estados que se comunican son disjuntas entre sí y cubren todo el conjunto de estados, generan una partición del conjunto. Si una cadena de markov contiene una sola clase de estados (todos los estados se comunican entre sí), se dirá que es *irreducible*.

Periodicidad

La existencia de una distribución límite depende esencialmente del comportamiento a largo plazo (n) de las probabilidades $P_{i,j}(n)$. Para analizar estas probabilidades necesitamos el concepto de estado periódico: Un estado es periódico si, partiendo de este estado, sólo es posible volver a él en un número de etapas que sea múltiplo de un cierto número entero mayor que uno. Es dicir, el estado j es periódico si existe un número entero K, tal que $P_{i,j}(n) > 0$ sólo para valores de n en el conjunto $(K, 2K, 3K, 4K, \ldots)$.

Cabe mencionar que un periodo se puede definir de la siguiente forma: El período d del estado j corresponde al máximo común divisor de los n estados para los que $P_{i,j}(n) > 0$. Si $d > 1$, diremos que el estado es periódico con periodo d.

Si $d = 1$ se dirá que el estado es *aperiódico*. Además se puede decir que el estado es aperiódico si no es posible volver a él (en este caso, el estado es aperiódico por la imposibilidad de chequear la condición de periodicidad).

2.3.5. Topologías Markovianas

Las cadenas de Markov presentan una amplia variedad de comportanmientos, por lo que a lo largo del tiempo se ha ido realizando algunas clasificaciones. Podríamos tomar tres de ellas como las más representativas:

- Las Cadenas Ergódicas
- Las Cadenas Absorventes
- Las Cadenas cíclicas

Ergódica

Diremos que una cadena de Markov X es ergódica Cuando es:

- Irreducible 2.3.4
- Recurrente no-nula (o recurrente positiva) 2.3.4

- Aperiódica 2.3.4

Teorema 2 *Si P es la matriz de transición de una cadena ergódica de s estados, entonces existe un vector tal que $\pi = [\pi_1, \pi_2, \ldots, \pi_s]$*

$$\lim_{\mathbf{n}\to\infty} \mathbf{P^n} = \begin{pmatrix} \pi_1 & \pi_2 & \ldots & \pi_s \\ \pi_1 & \pi_2 & \ldots & \pi_s \\ \vdots & \vdots & \ddots & \vdots \\ \pi_1 & \pi_2 & \ldots & \pi_s \end{pmatrix}$$

es decir, $\lim_{n\to\infty} P_{ij}(n) = \pi_j$

A π se le llama distribución de estado estable o de equilibrio para la cadena de Markov

La mayor parte de sistemas se modelan con modelos markoviano ergódicos ya que los estados adquieren un comportamiento estable

Un Ejemplo gráfico de un modelo ergódico sería el de la figura 2.9.

Absorvente

Diremos que una cadena de Markov es absorvente cuando llega al estado k y $P_{kk} = 1$, de manera que una vez que la cadena llega al estado k, permanece ahí para siempre.

Si k es un estado absorvente y el proceso comienza en el estado i , la probabilidad de llegar en algún momento a k se llama probabilidad de absorción, dado que el sistema empezó en i, se denota f_{ik}.

Un posible Modelo absorvente sería el de la figura 2.10.

Cíclica

Teorema 3 *Diremos que una cadena de Markov es cíclica si existe un número $s \in \mathbb{N}$ tal que $P_k = P_{k+s} \forall k \in \mathbb{N}$.*

La interpretación de esta definición es clara. Si una cadena es cíclica la distribución en cierto futuro coincide con la distribución de los estados en un futuro posterior, cualquiera sea la distribución inicial. El número s se denomina *período* de la cadena.

2.3.6. Modelo de Reconocimiento de Sonido

El modelo de Markov que vamos a construir para el reconocimiento de Música, tendrá como símbolos los parámetros que extraigamos de las señales acústicas, de tal forma que el HMM represente los cambios que ocurren a lo largo del tiempo en la señal.

Aprovechando las características generadoras de HMM, cuando creemos un descriptor para cada señal analizada, lo que hacemos es crear una cadena de Markov capaz de generar dicha señal, más bien, capaz de generar representativamente los cambios que irá sufriendo la señal parametrizada a lo largo del tiempo.

De tal forma, el acto de reconocer va a ser el proceso de buscar qué modelo tiene mayor probabilidad, es decir, parametrizada una señal a reconocer, lo que se mirará para saber a qué señal original pertenece, será ver qué modelo es capaz de generarla. El que tenga la mayor probabilidad será el que se parezca más. Solo habrá probabilidad uno, si la señal a reconocer es idéntica a la señal con la que se entrenó y generó el descriptor.

Los modelos de Markov serán de un solo estado, y los descriptores serán concatenaciones de dichos modelos.

El algoritmo de inferencia que utilizaremos será el de Viterbi, computando la probabilidad de un modelo como la probabilidad de su estado, de tal forma que los cálculos se simplifiquen.

A continuación entraremos en detalle en los requisitos sin perder de vista la visión general ofrecida hasta ahora.

2.3.7. Modelo gráfico

Los Modelos Ocultos de Markov es un paradigma tratable como modelo gráfico. Como podemos ver en la figura 2.8, el HMM se puede visualizar como una generalización dinámica en donde hay ejes adicionales que conectan los nodos ocultos. Cada nodo puede estar en uno de los M estados, parametrizando la matriz de transición en $M\ x\ M$ ejes. El problema de la inferencia reside en cómo calcular las probabilidades de los nodos ocultos dados una secuencia de nodos observables. Este problema se resuelve con algoritmos recursivos (algoritmos alpha - beta) que recorren hacia adelante y hacia atrás el grafo. En capitulos posteriores volveremos a tratar este tema relacionando el modelo gráfico de HMM y el DSPOOM.

2.3.8. Inferencia sobre modelos probabilísticos

En el contexto de los HMM, el problema más común de inferencia es el cálculo del likelihood de las observaciones dado un modelo, la probabilidad de las observaciones en dicho modelo.

Las principales tareas que se deben resolver a través de la inferencia son:

- *Filtrado o Monitereo.* Es la tarea de computar el estado de creencia, es decir, la distribución posterior a través del estado actual, dadas todas las evidencias hasta la fecha. Hay que computar $P(X_t|e_{1:t})$
- *Predicción.* Es la tarea de computar la distribución posterior a través de un estado

futuro, dadas todas las evidencias hasta la fecha. Hay que computar $P(X_{t+k}|e_{1:t})$ para alguna $k > 0$.

- *Suavizado o Retrospectiva.* Es la tarea de computar la distribución posterior a través de un estado pasado, dadas todas las evidencias hasta el presente. Hay que computar $P(X_k|e_{1:t})$ para alguna $0 \leq k < t$.
- *Estimación más probable.* Dada una secuencia de observaciones, encontraremos la secuencia de estados más probable de haber generado dichas observaciones.

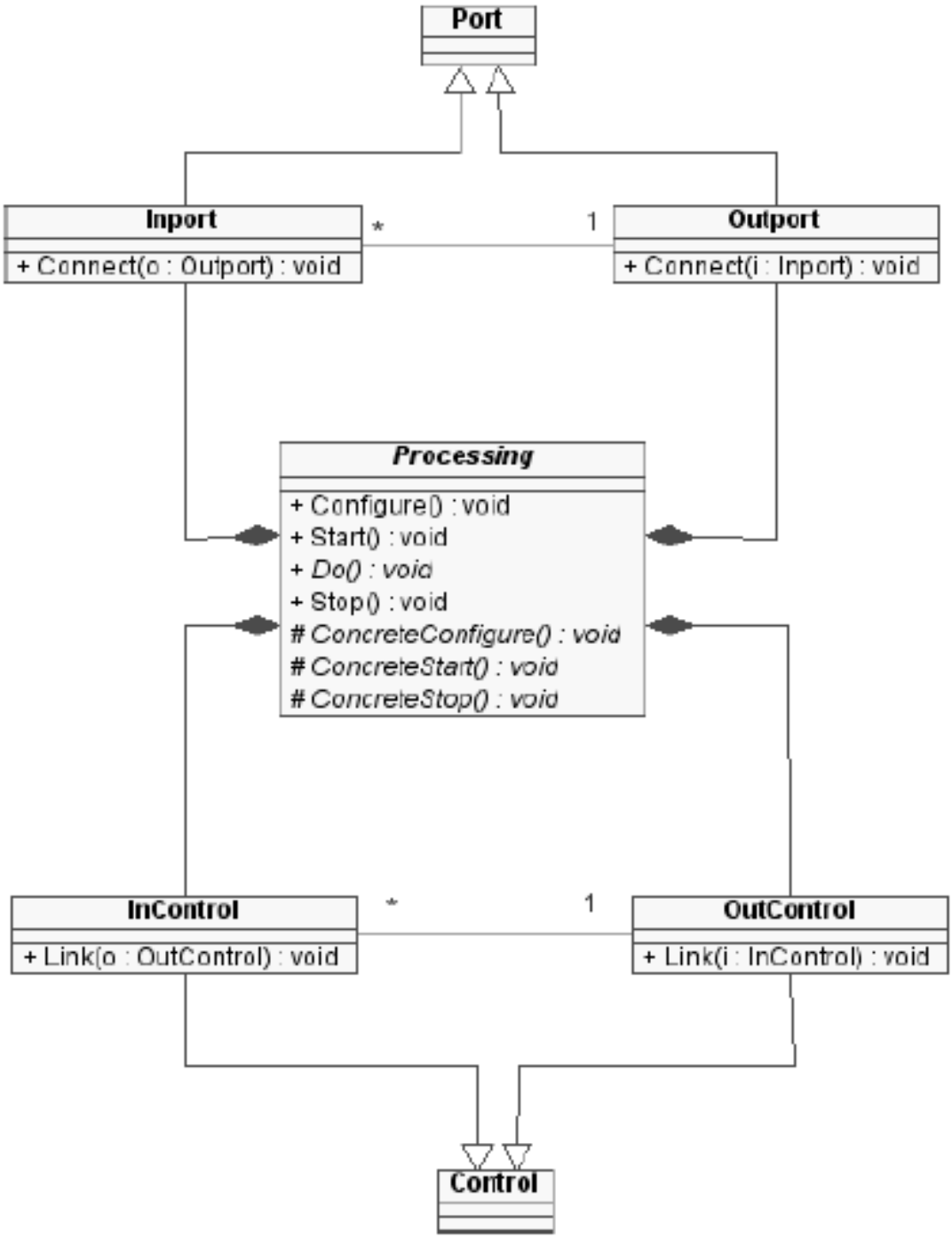

Figura 2.5: Diagrma de Clases de Network

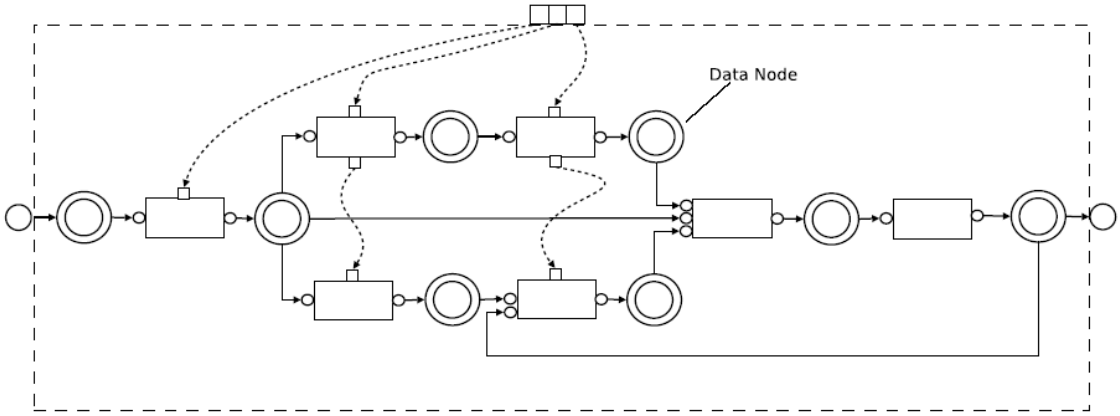

Figura 2.6: Network y Nodos de Datos

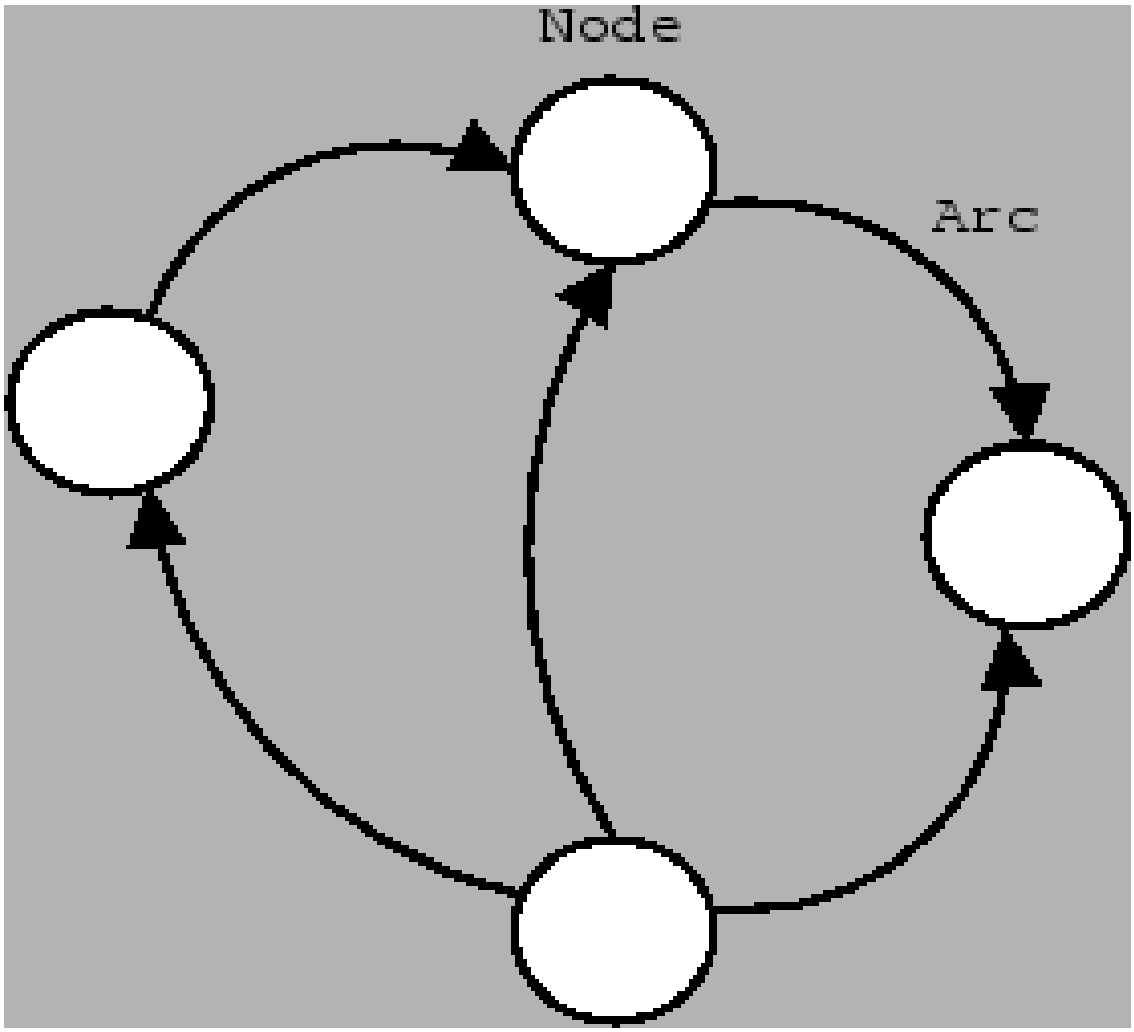

Figura 2.7: Grafo Ejemplo

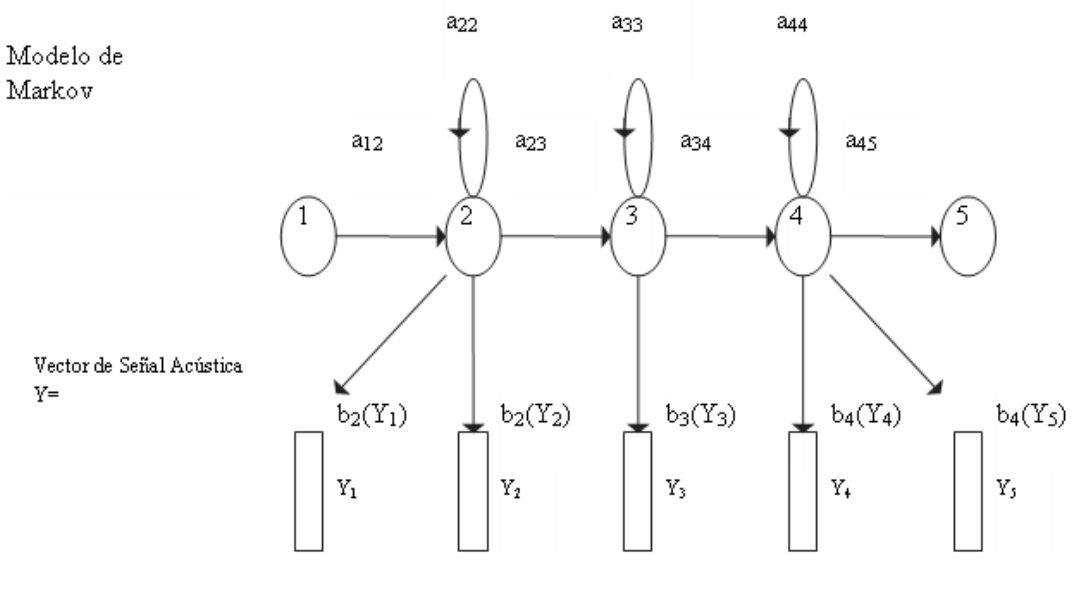

Figura 2.8: Ejemplo gráfico de un HMM

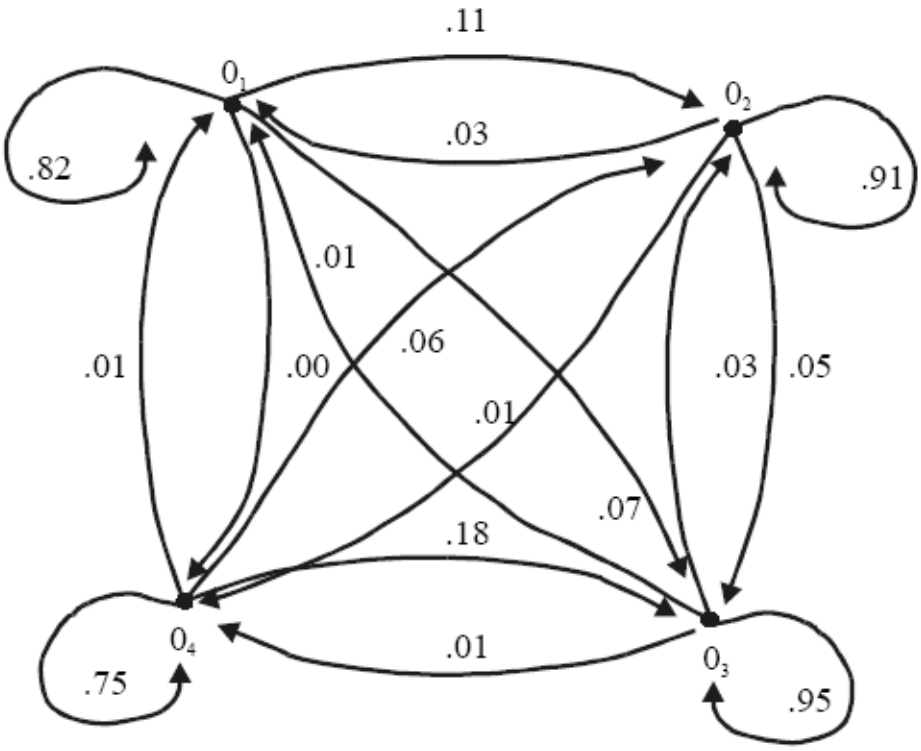

Figura 2.9: HMM Ergódico 4-estados

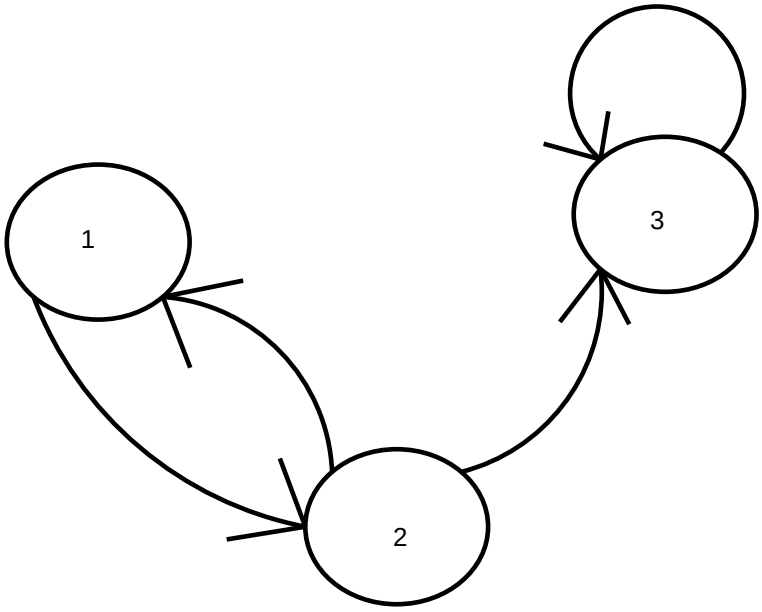

Figura 2.10: HMM Absorvente 3-estados

Capítulo 3

Análisis de requisitos

A continuación detallaremos los requisitos necesarios para ajustar el sistema, de forma que no nos alejemos de sus prioridades y nos sirva como guía o marcas que nos eviten errores de planteamiento, delimitando qué funciones queremos abarcar.

Definiremos los requisitos del sistema, en los cuales profundizaremos a nivel de detalle aclarando que es lo que tiene que realizar nuestro sistema y que no deberá hacer.

En la ingeniería del Software se suelen dividir los requisitos en tres categorías:

- requisitos funcionales
- requisitos no funcionales
- requisitos de dominio

En el primer apartado se definen las funciones que deberá ser capaz de realizar nuestro sistema y a qué necesidades tiene que ser capaz de responder.

En el segundo caso, son los requisitos provenientes de factores que no se relacionen con la función, por ejemplo la eficiencia, la robustez, el tiempo de reacción, etc. Suelen ser aspectos o exigencias restrictivas y que modelan el cómo desarrollar las respuestas del sistema.

Por último, los requisitos de dominio, como se puede inferir del nombre, se referirán a cómo nos afectará el dominio a nuestro proyecto, cómo el mundo en donde introduciremos el proyecto, modelará la forma en la que debe ser concebido.

La división viene a ser concretada con el fin de tener en primer lugar los objetivos de las funciones requeridas al sistema, a continuación las restricciones que deberá sostener dicho sistema y por ultimo cómo el contexto, el entorno, lo modela.

En este caso concreto, al tener dos modelos definidos, el probabilístico ejemplificado en AMADEUS y el de redes de procesos ejemplificado en CLAM, realizaremos una última subdivisión en donde analizaremos cada modelo por separado encontrando los puntos

comunes donde los dos coexisten. Es decir, nos fijaremos en el análisis subyacente a un caso concreto del modelo probabilístico como es el caso de AMADEUS, modelando Modelos de Markov y optimizado para un tipo restringido de funciones, para luego abstraer y llegar a una generalización de los requisitos. Por lo tanto primero plantearemos los requisitos necesarios para proyectar AMADEUS en CLAM y luego plantearemos los requisitos necesarios para realizar la generalización, pudiendo servirnos de ejemplos el uso de otros proyectos sobre los modelos probabilísticos.

3.1. Requisitos funcionales

Separaremos la descripción de los requisitos en los tres componentes destacables del sistema: Entrenamiento, generación de descriptores y reconocimiento.

3.1.1. Entrenamiento

Como podemos observar en la figura 3.1 Funcionalmente tendrá que ser capaz de permitir que un usuario extraiga una serie de parámetros especificados dado un audio de entrada.

El sistema ya implementa un módulo para extraer los parámetros. El usuario puede usarlo con la configuración por defecto o configurarlo a su medida. El módulo está compuesto por módulos más específicos (filtros, FFT, etc.), por lo que el usuario podrá experimentar componiendo la extracción de diferentes modos o agregando otros módulos no implementados en el sistema.

A su vez se podrá crear la lista de modelos de Markov usando la herramienta lnkNet, al cual se le deberá configurar los parámetros de creación de modelos. La salida de lnkNet es un archivo con dichos modelos representados según el estándar HTK.

Como extensión de uso, los modelos se podrán entrenar con los parámetros extraídos anteriormente.

Para poder leer los modelos en CLAM, se tendrá que poder convertir la representación en HTK a otra en XML ajustándose a nuestras estructuras de datos.

3.1.2. Generación de descriptores

En la figura 3.2 se observa que se tendrá que diseñar e implementar la estructura de datos necesaria en CLAM para cargar en memoria los modelos de Markov representados en XML ya que tendremos que utilizar los modelos creados y entrenados anteriormente.

La extracción de parámetros de un audio de entrada, en este caso servirá para generar los descriptores del mismo, que como resultado fabricará una concatenación de modelos de Markov capaz de representar los parámetros extraídos del audio. Esa lista de modelos

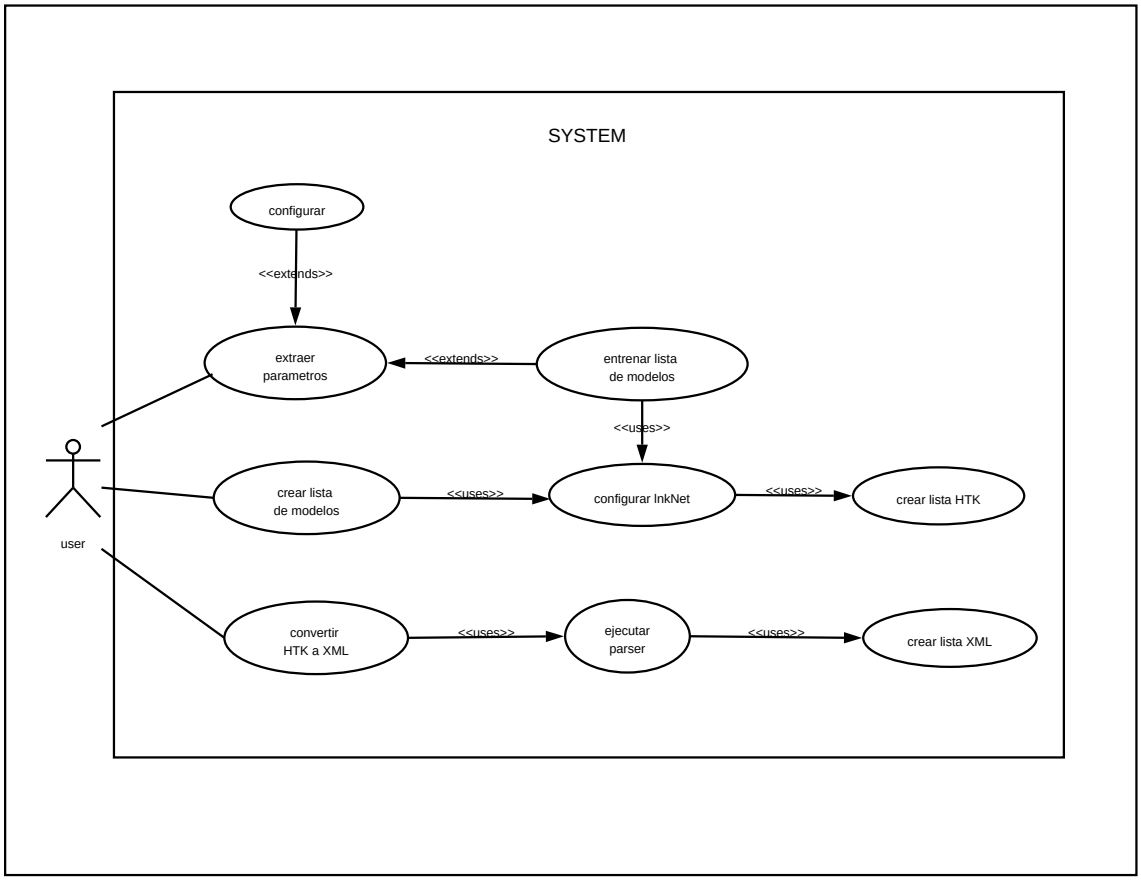

Figura 3.1: Casos de uso del Entrenamiento

que corresponde a la descripción de una entrada audio, será posible serializarla, siempre respetando las estructuras de datos y el estándar XML.

En el momento de querer crear los descriptores de un número considerable de audios, para evitar la inserción de uno por uno, crearemos un fichero que contenga todos los audios para analizar y construiremos un script en CLAM para leer la lista y así agilizar el trabajo del usuario.

3.1.3. Reconocimiento de Sonido

Diseñaremos e implementaremos la estructura de datos necesaria en CLAM para cargar en memoria los Descriptores representados en XML, a los cuales llamaremos con el nombre de VHMM (vector of Hidden Markov Models). También necesitaremos poder cargar los Modelos de Markov creados en el entrenamiento.

Para reconocer una señal de entrada, es necesario que esta señal haya sido analizada

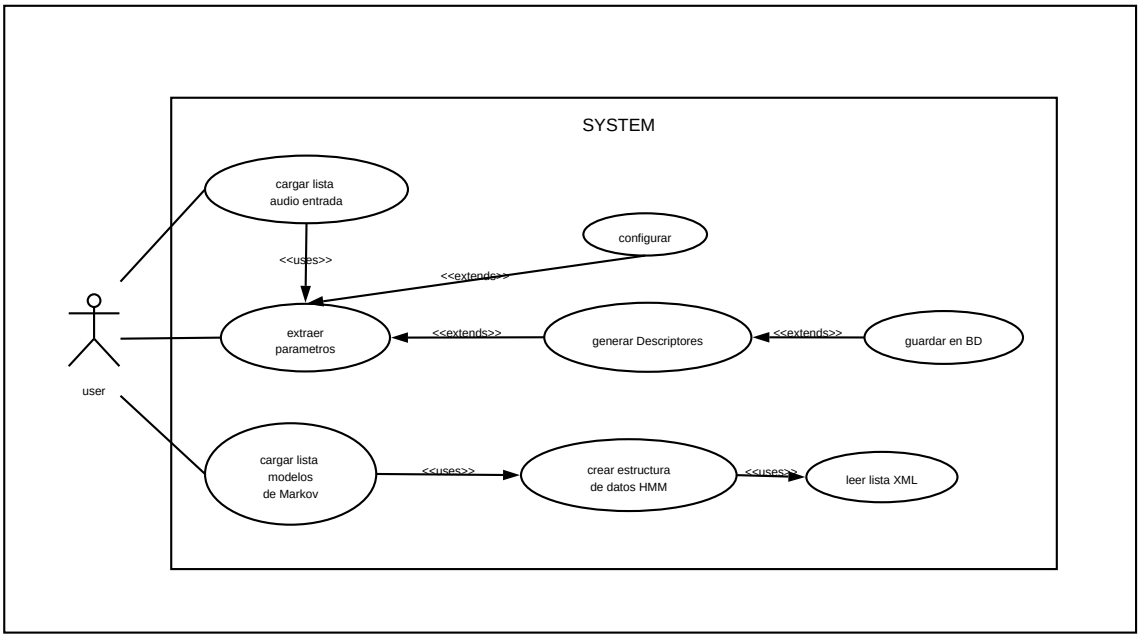

Figura 3.2: Casos de uso de la Generación de Descriptores

y creado su descriptor, a partir de allí la señal que analizaremos para reconocer no es necesario que sea completa, es decir que basta con un fragmento para poder encontrar qué descriptor es capaz de generar esta señal. Por lo tanto la extracción de parámetros se realizará con la posibilidad de dejar de extraer parámetros desde el momento que el sistema haya generado un resultado positivo respecto al reconocimiento de la entrada.

Una vez que extraemos los parámetros se los pasamos al algoritmo utilizado para reconocer la señal, el cual es configurable según distintos parámetros (tamaño de frame, intervalo de análisis del frame, etc.) que ya poseen un valor por defecto.

El algoritmo necesita la lista de modelos de Markov y los Descriptores para realizar sus cálculos, puesto que si el usuario no los ha cargado explícitamente, en ese caso antes de ejecutar realizaremos la carga en memoria necesaria para su funcionamiento.

También tendremos que considerar que en este caso a parte de un actor humano puede haber un actor que sea un sistema externo que nos envíe la señal para reconocerla. En ese caso el sistema tendrá que realizar las configuraciones y usos necesarios para reconocer la señal que reciba.

Se puede observar estos detalles en el diagrama de casos de uso 3.3

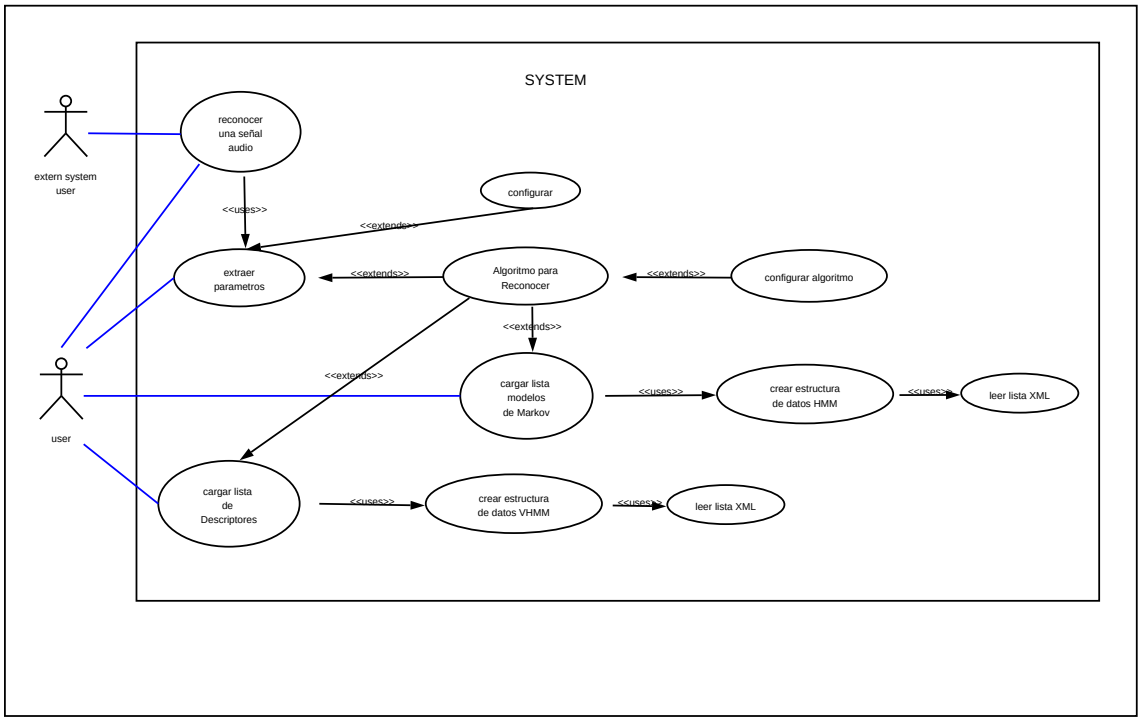

Figura 3.3: Casos de uso de la Generación de Descriptores

3.1.4. Extracción de parámetros

La extracción de parámetros tendrá que dar la funcionalidad de diversos filtros necesarios para dicho proceso donde cada componente dará una funcionalidad de forma independiente pudiendo así uno aprovecharlo para otros dominios:

En primer lugar es necesario realizar una *Extracción de la Energía* de la señal.

También necesitaremos poder realizar un filtro que realice un *Preenfasis* sobre la señal, controlado por un factor de configuración.

Otra importante funcionalidad es la de un filtro que sea un banco de Frecuencias, representando un espectro de x frecuencias en otro de y frecuencias, siendo y mucho menor que x de forma que comprima la información y retenga solo lo imprescindible. Este filtro nos devolverá en escala Mel en vez de Hz, cuya información le llamaremos MelCepstrum.

Será necesario poder usar un Filtro que realice la *Transformada Discreta de coseno* para poder así compactar la energía en el caso de que los datos estén correlacionados.

La generación de la *ventana sinusoidal* no se contemplaba en CLAM por lo que tendremos que añadir ésta funcionalidad al generador de ventanas.

El Filtro *Deconvolución homomórfica* el cual deshace los filtros lineales que se le haya

aplicado a la señal también es importante poder usarlo.

Por último necesitaremos Filtros que extraigan la *derivada*, la *aceleración* y otro que realice la *media* de un conjunto de datos.

Por otra parte CLAM tendrá que dar funcionalidad a representar los distintos arquetipos existentes de Modelos de Markov.

3.2. Requisitos no nuncionales

Uno de los puntos más importantes es detectar en qué casos es necesaria la eficiencia temporal del tiempo de ejecución. Después de haber analizado los casos de uso del sistema, podemos observar que las etapas primera y segunda serán *offline* mientras que la tercera será *online*, en tiempo real. Esto conlleva a que todos los procesos relacionados con la etapa de reconocimiento van a tener éste requisito. La extracción de parámetros que se utiliza también en las fases anteriores sufrirá de esta restricción, por lo que las etapas anteriores también se verán afectadas en parte.

Otro punto interesante que debemos destacar es la independencia de los distintos filtros, de tal forma que exista un proceso que junte todos los filtros según una lógica estudiada, pero que a su vez permita que todos sus subprocesos puedan ser utilizados en otros campos sin perder la coherencia de datos al interrelacionarlos.

Como rasgo principal al modo de implementación, cabe mencionar que el proyecto va a estar enmarcado dentro de la librería CLAM, por lo que obliga a usar el lenguaje de programación C++ y preservar en la implementación la filosofía interna del FrameWork, respetando:

- Jerarquías de clases.
- Funcionalidad en diversas plataformas.
- Interfaces.
- Contextos y tipos de datos.
- Uso de XML para la serialización.
- Claridad y flexibilidad en el diseño de las clases.

Este último aspecto viene marcado por la importante participación de los usuarios de CLAM, que entendiendo el código cooperan de forma activa como testers.

3.3. Requisitos de usuario

El uso del sistema está enfocado hacia usuarios con conocimientos de C++ y modelos de Markov. No será necesario tener grandes conocimientos de música para su uso ya que se dará la posibilidad de una fácil integración de las tres etapas principales, dejando la posibilidad de modificar para usuarios más avanzados los algoritmos de extracción de parámetros, generación de descriptores y reconocimiento.

Por esta razón habrá un modo de uso más automatizado y otro más abierto a ser personalizado por el usuario. También se permitirá la conexión de otros sistemas que lo usen como usuario. Pero ante todo, el usuario siempre actuará como usuario de una librería que puede utilizar, modificar o agregar funcionalidades, por lo que tendremos que comprometernos con la construcción de una API *transparente y usable*.

3.4. Requisitos de dominio

Vamos a tratar ciertos aspectos relacionados con el entorno del audio.

Cuando se envía una señal de entrada, todos los procesos del sistema se tendrán que configurar en función de ella para su correcto funcionamiento, configuración que creará una latencia considerable en el reconocimiento de la señal si se debiera realizar para cada audio de entrada.

Considerando este aspecto, y el de la redundancia de datos al utilizar frecuencias de muestreo altas para describir y reconocer un audio, realizaremos una estandarización del *Sample Rate* de todas las señales del sistema, hecho que nos llevará a comprobar siempre el sample rate y convertirlo en el deseado. De esta forma bastará con configurar una sola vez de forma inicial todo el sistema.

También convertiremos los audios de dos canales en audios de uno solo, siempre con el mismo criterio de mantener solo la información principal.

En lo que respecta a la carga en memoria de un fichero audio, para no colapsar la RAM haremos funcionar todo el sistema enventanando la señal con el tamaño mínimo que garantice el correcto funcionamiento de los cálculos de extracción de parámetros, de tal forma que tendremos en memoria el audio parametrizado, hecho que no influye por su bajo peso en densidad de datos (Kb).

El uso del *Delay* como en el caso de los filtros de Aceleración, Derivada, Viterbi, etc. nos obliga a mantener una información temporal, hecho que siempre habrá que observar para no perder consistencia en los datos.

3.5. Generalización del Sistema

Por último cabe destacar la importancia de ser capaces de generar un sistema flexible, dentro del marco de flexibilidad permitido en el framework de CLAM.

Analizando otros sistemas, como puede ser Amadeus, se observa el alto grado de especificación, dirigido e optimizado para casos de uso muy determinados.

La ventaja reside en la eficiencia, puesto que uno debe considerar menos aspectos simplificando el posterior diseño. No obstante CLAM ha conseguido ser una librería de propósitos generales, dentro del marco del procesado de audio, conservando un alto grado de eficiencia; hecho que nos impone ser capaces de generar un sistema con dichas características.

Capítulo 4

Herramientas usadas

A continuación haremos un recorrido descriptivo y orientativo de las herramientas que nos hemos servido para afrontar los requisitos del sistema.

4.1. Introducción

Si algún fruto dio la inteligencia, desde milenios atrás, el más privilegiado fue el invento de herramientas que nos facilite la solución de problemas determinados. Así hemos ido creando éste gran abanico de ayudas sin el cual el ser humano no sabría sobrevivir, así en todos los campos, y así ocurre en la informática, en donde existe esa necesidad de uso; no tendría sentido ni seríamos capaces en un corto plazo de tiempo de responder a nuestras necesidades.

Sin embargo, más importante que la existencia de dichas herramientas es la capacidad de elección, la capacidad de abstraer ventajas de un grupo de herramientas, de proyectar un análisis de la situación y ver qué herramienta da un puntaje mayor, que por una serie de parámetros, el coste, el tiempo de aprendizaje, etc. nuestro entendimiento consigue evaluar como la solución óptima.

Más concretamente, en el dominio de la informática nos han formado, como ingenieros, una base de conocimientos para ser capaces de realizar el proceso de análisis de una situación y parámetros a tener en cuenta. La correcta elección de las herramientas que vamos a usar es clave para el desarrollo del proyecto, También cabe decir que existen requisitos de sistema que ya imponen implícitamente una herramienta de uso, por lo tanto elegimos sólo en los aspectos que impliquen un grado de libertad sobre las posibles herramientas.

Hemos separado las herramientas usadas en dos apartados, según el ámbito donde las hemos usado:

1. Las enfocadas al análisis y diseño:

- Paradigma de programación
- Métodos de ingeniería de software
- Diagramas UML

2. Las enfocadas a la implementación:

- Lenguaje de programación C++
- CLAM C++ Library for Audio and Music
- Construcción de un parser
- Desarrollo en grupo (CVS)
- Tests Unitarios
- Estándar HTK
- LNKNet
- Compilador
- Editor código fuente

4.2. Paradigmas de Programación

El concepto "Paradigma" se refiere a realizaciones científicas universalmente conocidas, que durante un cierto periodo de tiempo nos ofrece además de los problemas, las soluciones modelos para la comunidad que esta involucrada en la ciencia. Por extensión, "Paradigma de programación" se referirá más concretamente a la colección de conceptos que guiarán el proceso de construcción de una aplicación, determinando su estructura. Estos conceptos controlaran la forma en que pensaremos y formularemos los programas.

Los paradigmas de programación son los siguientes:

- Procedural
- Orientado a Objetos
- Funcional
- Lógico

4.2.1. Orientado a Objetos

La orientación a objetos se acopla a la simulación de situaciones del mundo real. Las entidades centrales son los objetos que son tipos de datos que se encapsulan con el mismo nombre, las estructuras de datos y las operaciones que manipulan estos datos. El atractivo intuitivo de la orientación a objetos es que proporciona conceptos y herramientas

con las cuales se modela y representa el mundo real tan fielmente como sea posible. La programación orientada a objetos según Grady Booch es "Un método de implantación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de clase u objeto, y cuyas clases son todas miembro de una jerarquía de clases unidas mediante relaciones de herencia". El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

- *Modularidad*, esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.
- *Ocultamiento de la información*, es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependan de ello.

Un lenguaje orientado a objetos debe soportar el polimorfismo, lo que significa que clases diferentes pueden tener comportamientos diferentes para el mismo método. La herencia nos permite crear una relación jerárquica entre las clases usando subclases. Una subclase hereda atributos y métodos de su superclase.

Con la herencia, puedes crear y agregar funcionalidades a las clases existentes. De una simple clase base, puedes derivar clases más complejas y especializadas como sea necesario. Esto hace el código más reusable, que es una de las grandes ventajas de la POO. Los lenguajes más conocidos que usan este paradigma son entre ellos JAVA, C++, etc.

4.2.2. Elección

Los aspectos que condicionaron la elección del paradigma orientado a objetos fueron los siguientes:

- Los mecanismos de *encapsulación* soportan un alto grado de reutilización de código, que se incrementa por sus mecanismos de herencia.
- Referente a las bases de datos, se adjunta bien a los modelos semánticos de datos.
- La alta *modularidad* del paradigma, que permite subdividir un programa en módulos con una independencia notable.
- La *jerarquización* que nos va a ayudar a realizar una ordenación de las abstracciones.
- El *polimorfismo* de la orientación a objetos nos permitirá ser más independientes a los tipos de las entradas de datos.

Otros aspectos que nos agradaron del paradigma fueron propiedades tales como *la concurrencia, la persistencia de datos y el manejo de excepciones.*

4.3. Métodos de Ingenieria del Software

Para disminuir el fracaso de los proyectos de Software se han desarrollado diversas metodologías para el análisis y diseño, dando diversas soluciones a aspectos conflictivos en la gestión de proyectos como el análisis del tiempo, de los recursos, de costos, etc. El modelo de ciclo de vida que elegí más idóneo a las características de mi proyecto fue el incremental [30], que combina elementos del modelo lineal secuencial con la filosofía interactiva de construcción de prototipos donde cada secuencia lineal produce un "incremento" del software, ya que desarrollaré pequeñas partes, realizando iteraciones muy cortas de tal forma que el sistema se irá creando paso a paso a sabiendas de que los módulos anteriores cumplen ya un ciclo, estando garantizado su correcto funcionamiento. A su vez, a lo largo de todo el proyecto, se utilizó la técnica de "Refactoring" [19], consistente en un proceso de cambio del sistema a través de un camino que no altere las prestaciones y significados externos pero cambiando la estructura interna del código. Es una forma disciplinada de reciclar código sin introducir errores.

La paradoja del *Refactoring* es que uno diseña después de que el código haya sido escrito, mientras que normalmente uno diseña y luego escribe. Por lo tanto uno va encontrando el diseño a medida que rescribe, pero en nuestro caso ya teníamos unos patrones de diseño a seguir dentro de CLAM. Se podría decir que mientras uno construía el sistema, el diseño va naciendo.

4.4. Diagramas UML

UML es una especificación de notación orientada a objetos. Divide cada proyecto en un número de diagramas que representan las diferentes vistas del proyecto. Estos diagramas juntos son los que representa su arquitectura, por lo que nos sirve como una herramienta de análisis y diseño.

Se da mucha importancia al diagrama de clases ya que éste representa una parte importante del sistema, pero sólo representa una vista estática, es decir muestra al sistema parado. Sabemos su estructura pero no sabemos qué le sucede a sus diferentes partes cuando el sistema empieza a funcionar.

UML introduce nuevos diagramas que representa una visión dinámica del sistema. Es decir, gracias al diseño de la parte dinámica del sistema, podemos darnos cuenta en la fase de diseño, de problemas en la estructura, ya que se observará la propagación de errores, las partes que necesiten ser sincronizadas, así como el estado de cada una de las instancias en cada momento.

El diagrama de clases continua siendo muy importante, pero se debe tener en cuenta que su representación es limitada, y que ayuda a diseñar un sistema robusto con partes reutilizables, pero no a solucionar problemas de propagación de mensajes ni de sincronización o recuperación ante estados de error.

4.5. Lenguaje de Programación C++

Existen varios lenguajes que permiten escribir un programa orientado a objetos; de los cuales elegí C++. ¿Por qué C++? [32] Porque posee características superiores a otros lenguajes. Las más importantes son: portabilidad, brevedad, programación modular, compatibilidad con C y velocidad.

Además, se trata de un lenguaje de programación estandarizado (ISO/IEC 14882:1998), ampliamente difundido, y con una biblioteca estándar C++ que lo ha convertido en un lenguaje universal, de propósito general, y ampliamente utilizado tanto en el ámbito profesional como en el educativo.

Respecto a su antecesor (C), se ha procurando mantener una alta compatibilidad hacia atrás por dos razones: poder reutilizar la enorme cantidad de código C existente, y facilitar una transición lo más fluida posible a los programadores de C clásico, de forma que pudieran pasar sus programas a C++ e ir modificándolos.

Por lo general puede compilarse un programa C bajo C++, pero no a la inversa si el programa utiliza alguna de las características especiales de C++. Algunas situaciones requieren especial cuidado.

C++ no es un lenguaje orientado a objetos puro (en el sentido en que puede serlo Java por ejemplo), además no nació como un ejercicio académico de diseño. Se trata simplemente del sucesor de un lenguaje de programación hecho por programadores (de alto nivel) para programadores, lo que se traduce en un diseño pragmático al que se le han ido añadiendo todos los elementos que la práctica aconsejaba como necesarios.

De hecho, en el diseño de la Librería Estándar C++ [15] se ha usado ampliamente esta dualidad (ser mezcla de un lenguaje tradicional con elementos de POO), lo que ha permitido un modelo muy avanzado de programación extraordinariamente flexible.

Muchos sistemas operativos compiladores e intérpretes han sido escritos en C++ (el propio Windows y Java). Una de las razones de su éxito es ser un lenguaje de propósito general que se adapta a múltiples situaciones.

4.6. CLAM C++ Library for Audio and Music

CLAM (C++ Library for Audio and Music), una librería multiplataforma con la finalidad de ser un framework estándar que ayude a desarrollar aplicaciones relacionadas con el procesado de señal. Actualmente está siendo desarrollada por el *MTG* (Music Technology

Group) del Instituto Universitario del Audiovisual (*IUA*) de la Universidad Pompeu Fabra (*UPF*), habiéndolo iniciado en el año 2000.

A su vez, trata de unificar los puntos de vista sobre temas comunes existentes en distintos proyectos del *MTG*, obligándole a ser muy escalable y con un grado alto de modularidad. Podríamos destacar conceptualmente la existencia de un modulo de serialización, otro de visualización, control de flujo, Entrada - Salida y un núcleo constituido por los datos de procesado y sus respectivos algoritmos.

Respecto al marco legal, tiene una licencia *GPL* (General Public License) y está enmarcado en el proyecto europeo *AGNULA*.

4.6.1. Objetivos

Por lo tanto las líneas generales con las que nació fueron de:

- Crear un framework común para el desarrollo interno de proyectos dentro del MTG.
- Facilitar el intercambio de conocimientos creando una estandarización provocada por el uso de CLAM
- Ser capaz de proporcionar un correcto funcionamiento sobre diversas plataformas, ofreciendo una interfaz de programación (API) transparente, despreocupando así al usuario de las particularidades de cada sistema.

Respecto a aspectos que requerían otros proyectos, se propuso como punto básico dar soluciones a:

- La gestión de aspectos de entrada y salida hardware de audio sobre las distintas plataformas.
- El control de las instrucciones MIDI.
- El control de los aspectos del multithreading existente en las aplicaciones.
- La pasivación y activación de los objetos de datos de una aplicación.

A medida que el proyecto ha ido creciendo, se han sumado varios objetivos más particulares relacionados con distintos proyectos, que no forman la base funcional del framework pero que a su vez lo enriquecen.

4.6.2. Estructura Interna

En la documentación de CLAM [8] se explica con profundidad de detalle su estructura interna; a continuación intentaremos usarnos del diagrama 4.1 para dar las mínimas pinceladas necesarias para visualizar su arquitectura modular.

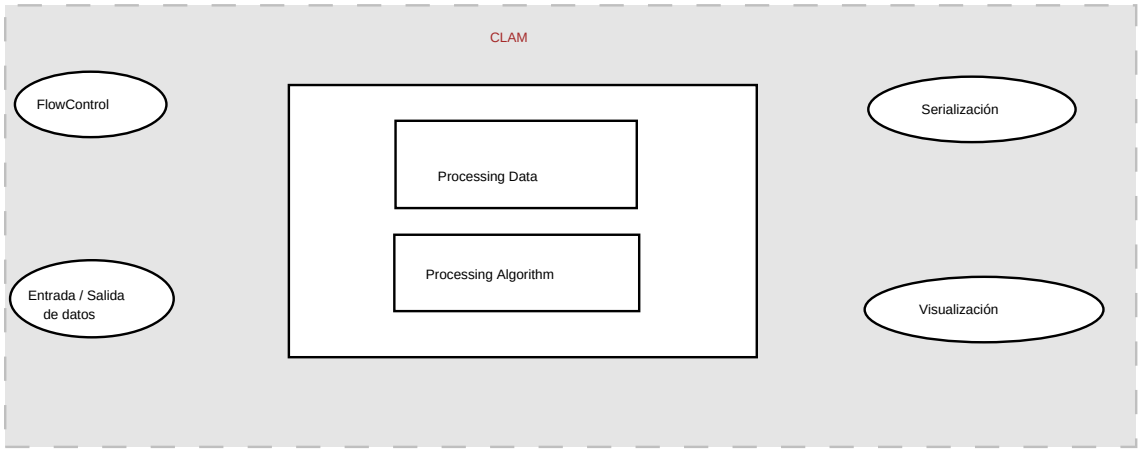

Figura 4.1: Módulos de CLAM

Al querer ser modular de tal forma de poder prescindir de aquella parte que uno no necesite para desarrollar una aplicación, se ha propuesto una forma jerarquizada de clases para modelar distintos conceptos relacionados entre si por sus dominios. La base de todo el árbol es una clase llamada Component, hecho que nos permite tener puntos en comunes en todo los niveles inferiores, ya sea restrictivos como funcionales.

4.6.3. Modelo de Procesado

El modelo que plantea para hacer frente al procesado de señal se podría dividir en cuatro subapartados:

- Algoritmos de procesado (*Processing o ProcessingComposite*).
- Modelación de los datos que manipulan dichos algoritmos (*ProcessindData*).
- Pasivación y representación transparente y manejable de los datos (*XML*).
- Flujo de los datos de procesado a través de los diferentes Processing (*FlowControl*).

Algoritmos de procesado (*Processing o ProcessingComposite*)

Al modelar un algoritmo dentro del paradigma del modo de funcionamiento de *Processing*, el usuario tendrá conciencia de ciertos detalles mientras otros le permanecerán ocultos con el fin de facilitarle el uso y no tener que enfrentarse al código *oscuro* del algoritmo. De tal forma se le facilitará la información relativa a los datos de entrada y salida, consiguiendo de esa manera una despreocupación por parte del usuario de cómo se

implementa el algoritmo y facilitándole la tarea de concentrarse en buscar qué algoritmos le son más útiles para los requisitos del problema que intenta resolver.

El único detalle que tendrá que observar respecto al funcionamiento del algoritmo será la configuración, ya que muchos de ellos necesitan que se les ajuste ciertos parámetros, pudiendo hacerlo en tiempo de compilación con el uso de objetos modelados en CLAM con el nombre de *ProcessingConfig* o en tiempo de ejecución usando los *Controles*, cuya utilidad es la de mandar señales al algoritmo en caso de cambios en tiempo de ejecución, de forma síncrona o asincrónica.

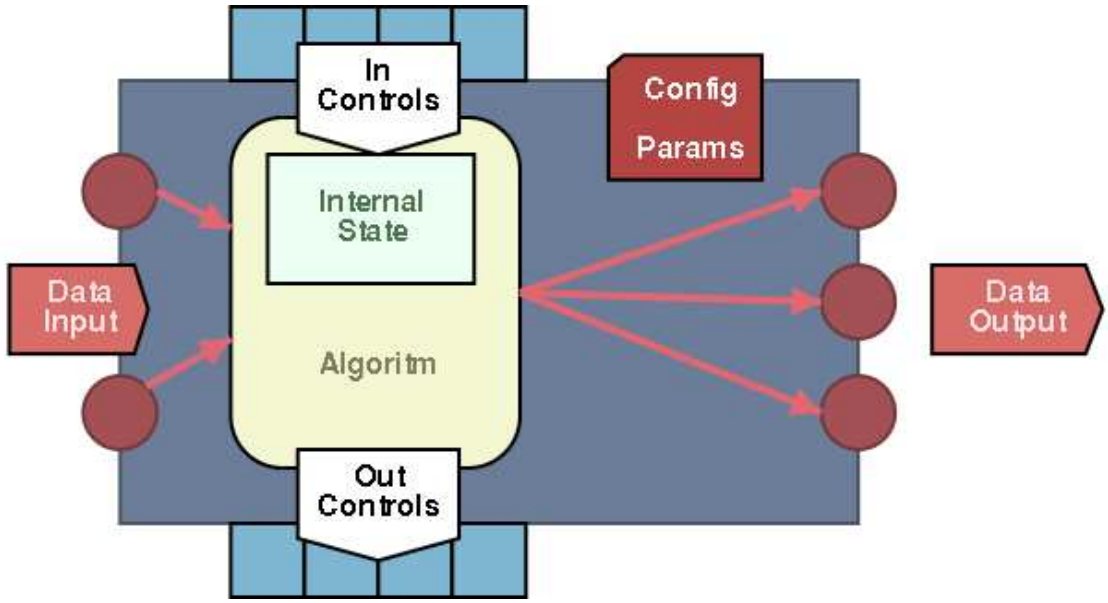

Figura 4.2: Arquitectura de un Processing

Como podemos observar en la figura 4.2, el *processing* tiene dos buffers de *controles*, uno para ir procesando los controles que le van llegando y otro para mantener los que se deben ir enviando.

Una peculiaridad del *processing* es su estado interno, puesto que normalmente uno se refiere a el como el valor que hayan tomado sus atributos, sin embargo podemos dividir el estado de un objeto *processing* según sus atributos de configuración y según los atributos relativos a la ejecución.

A veces, por la complejidad de un *processing* que uno desea implementar, ocurre que internamente use otros *processing* implementados en CLAM. Ese es el Caso que llamamos *ProcessingComposite*, puesto que estará compuesto por uno o más *processing* internamente. Los aspectos de configuración de los *processing* internos se tienen que controlar.

Modelación de datos (*ProcessigData*)

Los datos que se manejan a través de los distintos algoritmos vienen encapsulados por el modelo de *ProcessingData*, el cuál consigue la diversidad de sus tipos a través de las estructuras y semánticas adherentes a cada dominio.

Desde el punto de vista del significado de los datos podríamos realizar la siguiente separación:

- Datos del dominio frecuencial.
- Datos del dominio temporal.
- Datos que representan características abstractas de una señal, es decir, diferentes tipos posibles de parametrización.

Unos datos que por vez primera se representan de una forma dada, pueden ir cambiando de semántica a través de ser manipulados por los *Processing*, como también pueden variar su estructura. Cabe destacar dos modos de estructurar los datos:

- buffers (listas, vectores, etc.), tipos básicos (strings, integer, flota, etc.), etc. Que sirven para estructurar los datos en un bajo nivel.
- Estructuras más complejas que a su vez aportaran semántica a la estructura, como puede ser el que un *ProcessingData* contenga como miembro a otro, formando un árbol semántico y estructural.

La buena conjunción de estructura y significado ayudará a la claridad del sistema ya que por ejemplo, a nivel estructural se pueden guardar en un buffer de flotantes una componente frecuencial de una señal o una gama de amplitudes por unidad de tiempo, los dos conceptualmente muy diversos pero estructuralmente idénticos. A su vez, los mismos conceptos, dependiendo de su entorno o algoritmos que los utilicen, pueden obligar a que datos idénticos a nivel conceptual se tengan que representar de diversas formas estructurales; hecho que aumenta la complejidad del sistema y provoca en algunos casos replicaciones de datos.

Hasta la actualidad no se había abordado el problema de la representación de los descriptores, que en su mayoría son parámetros estadísticos que no requieren una complejidad estructural elevada, por lo que se está planteando una posible aproximación que consista en una "piscina de datos" en donde se contextualice y se agrupe según la semántica de los datos un conjunto de descriptores relacionados entre si. Más a delante discutiremos con más detalle esta cuestión ya que hubo una disertación entre posibles soluciones propuestas, una de ellas provenientes de la implementación de la parte práctica de éste proyecto.

Algo que siempre prevalece en cualquier planteamiento de modelación es la obligación de la transparencia y claridad del diseño y la implementación.

Con este fin se desarrolló el diseño y la implementación de los *Dynamic Types*, macros desarrolladas en C que sirven como una extensión del lenguaje C++. Su uso permite que los atributos de los objetos de datos sean creados, destruidos, pasivados o activados en tiempo de ejecución a través de una interfaz de uso muy transparente. Cuando el usuario las usa, dichas macros se encargaran de expandir el código necesario. De hecho todos los *ProcessingData* Son *Dynamic Types*.

Pasivación de los Datos (*XML*)

Éste apartado podríamos decir que es el que destaca a CLAM más como framework que como librería ya que ofrece los mecanismos necesarios para la pasivación o activación de los datos del sistema, parcial o totalmente, pudiendo salvar su estado en cualquier punto de la ejecución. A su vez se puede recuperar esos estados en cualquier punto de tiempo de ejecución facilitando así la interacción con otros sistemas.

Esto se consigue a través del uso del estándar *XML*, al cual se le ha ido desarrollando una interfaz de uso dentro de CLAM ya que no existe en C++ esta interfaz como puede ocurrir con otros lenguajes que ya definen una API para usarlo como es el caso de JAVA.

Flujo de los datos (*FlowControl*)

El flujo de datos de un sistema, corresponde a las conexiones entre diferentes procesos por donde pasarán los datos y el orden con el que se ejecutarán dichos procesos. Cumpliría la aproximación de un grafo dirigido, donde los nodos corresponderían a los Processing donde a través de las aristas viajarían los ProcessingData.

Por lo tanto el *FlowControl* corresponde a la tarea de controlar el correcto funcionamiento del flujo de datos, es decir, la estructuración del orden de ejecución y la adjudicación de "carreteras" por donde viajarán los datos.

De esta forma CLAM propone dos grados de responsabilidad del control del flujo de datos:

- Se hace responsable el usuario (Modo No supervisado).
- Se hace responsable CLAM (Modo Supervisado).

4.7. Construcción de un Parser

Hubo la necesidad de construir un parser que transforme documentos de modelos de Markov estructurados según el estándar HTK a una representación más conveniente para las estructuras de datos implementadas en CLAM y respetando la representación de XML.

Para ello nos servimos del FLEX y BISON, un analizador léxico y otro sintáctico, que permiten generar programas en lenguaje C, con algunas restricciones (debe ser un lenguaje regular para Flex y uno independiente de contexto para Bison).

4.7.1. FLEX

El reconocimiento de patrones léxicos -conjuntos catalogados de palabras- es una tarea muy importante con aplicaciones en una infinidad de rangos, el diseño e implementación de un rastreador o scanner para dichos patrones léxicos varía de acuerdo al tamaño del proyecto y a la ambición del que lo implemente, de manera que se hace evidente, como en todas las áreas de desarrollo de software que es necesario por todos los medios posibles desarrollar técnicas, métodos, teorías y herramientas que nos permitan realizar este proceso de análisis e implementación en el menor tiempo posible y al menor costo.

Flex [11] es un programa que realiza esta tarea, permite generar scanners en una forma y con una sintaxis bastante sencilla que se corresponden directamente con la de las expresiones regulares.

Flex lee los archivos de entrada de dados, o la entrada estándar si no se le ha indicado ningún nombre de archivo, con la descripción del scanner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominados reglas. Flex genera como salida un archivo fuente en C, 'yylex()'. Este archivo se compila y se enlaza con la librería '-lfl'para producir un ejecutable. Cuando se arranca el archivo ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

4.7.2. BISON

Bison [1] es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto LALR en un programa en C que analice esta gramática. Bison puede ser utilizado para desarrollar un amplio rango de analizadores de lenguajes, desde aquellos usados en simples calculadoras de escritorio hasta los complejos lenguajes de programación.

Bison es compatible hacia arriba con Yacc: todas las gramáticas escritas apropiadamente para Yacc deberían funcionar en Bison sin ningún cambio.

Bison aplica el concepto de traducción dirigida por la sintaxis, es decir, a cada entrada en el programa fuente le asigna una salida, la cual puede ser cualquier acción, la cual se ejecuta al reconocerse la entrada válida. En Bison, una regla gramatical tiene una acción compuesta de sentencias de C asociadas.

El propósito de una acción es generalmente computar el valor semántico de la construcción completa a partir de los valores semánticos de sus partes. Por ejemplo si se tiene una regla que afirma que una expresión puede ser la suma de dos expresiones. Cuando el

analizador reconozca tal suma, cada una de las subexpresiones posee un valor semántico que describe cómo fueron elaboradas.

Cuando ejecuta Bison, uno le da un archivo de gramática de Bison como entrada. La salida es un programa fuente en C que analiza el lenguaje descrito por la gramática. Este archivo se denomina un analizador de Bison. Hay que tener en cuenta que la utilidad Bison y el analizador de Bison son dos programas distintos: la utilidad Bison es un programa cuya salida es el analizador de Bison que forma parte de su programa. El trabajo del analizador de Bison es juntar tokens en agrupaciones de acuerdo a las reglas gramaticales por ejemplo, construir expresiones con identificadores y operadores. A medida que lo hace, éste ejecuta las acciones de las reglas gramaticales que utiliza.

Los tokens provienen de una función llamada del analizador léxico que uno provee de alguna manera (en nuestro caso desde el programa generado por FLEX). El analizador de Bison llama al analizador léxico cada vez que quiera un nuevo token. Éste no sabe qué hay "dentro" de los tokens (aunque sus valores semánticos podrían reflejarlo). Típicamente el analizador léxico construye los tokens analizando los caracteres del texto, pero Bison no depende de ello.

4.8. Desarrollo en grupo

La utilización de una herramienta para coordinar el desarrollo de CLAM en grupo, es fundamental para la cooperación entre los distintos desarrolladores. De esta forma se evita incoerencias y se facilita la comunicación interactiva del personal.

4.8.1. CVS Repositorio de Códigos Fuentes

CVS, Concurrent Versions System, es una aplicación cliente/servidor que se encarga de mantener un repositorio de software centralizado que es actualizado y distribuido, desde y hacia las copias locales de los desarrolladores. Es una utilidad clave para el desarrollo de sistemas en grupo, que implique una colaboración entre desarrolladores (aunque solamente haya uno).

Sin embargo no se utiliza solamente para desarrollo de software y manejo de código fuente. CVS está siendo utilizado para mantener sitios web, o para mantener documentación y para grandes proyectos como los ports. Incluso se puede utilizar para mantener copias de trabajo en la máquina de casa, en la del trabajo, en el notebook, etc. El funcionamiento es simple, el desarrollador se conecta con el servidor CVS y le pide la última versión disponible del proyecto, en este paso el desarrollador puede ver qué cambios se han realizado respecto a su versión local y los conflictos que pudiera ocasionar el código que el desarrollador ha realizado en su copia local con el código que ya está disponible en el servidor.

En caso de que el código no sea problemático se modifican los ficheros locales respe-

tando los cambios del desarrollador.

El *servidor CVS* se encarga de manejar "una historia" de lo que ocurre, mantener un registro de los cambios realizados a cada fichero y de servirlos según las necesidades del desarrollador. Además gestiona diversas utilidades para controlar en qué ficheros se está trabajando y notificar a los autores de los ficheros de los cambios.

El *cliente CVS* del desarrollador se encarga de obtener las últimas versiones disponibles (o las que necesite), de confrontarlas con las copias locales y de crear una copia local de los ficheros del proyecto que sean editables por el desarrollador. Y por supuesto de añadir el código del desarrollador al proyecto.

4.9. Tests unitarios

Para asegurar que las funcionalidades que se iban rediseñando conservaban el correcto funcionamiento, se seguía el método de los tests unitarios o también llamado metodología test-driven development [4], cuyo fin es probar que la funcionalidad cumple los requisitos definidos.

Se necesita tiempo para implementar los tests, pero a la larga ahorra tiempo en detectar dónde residían los errores. A su vez cuando uno se introduce en un sistema con gran cantidad de clases resulta muy complicado realizar un trazado para encontrar errores, pero más grave resulta si el error reside en una clase del sistema implementada por otro programador, puesto que invertiremos mucho tiempo en comprender las estructuras antes de encontrar dicho error.

Otra razón interesante por la cual merita implementar los tests, reside en la escalabilidad, cuando la clase está terminada y queremos introducir cambios futuros, el comprobar que todo sigue funcionando es simplemente seguir cumpliendo sus correspondientes tests.

Esto se consigue a través de CPPUnit [9] que es una herramienta para realizar dichos tests en C++.

4.10. Estándar HTK

La aplicación LNKNet usa el estándar HTK [35] para representar los datos. HTK (Hidden Markov Model Toolkit) es un toolkit para construir y manipular modelos de Markov. Principalmente se usa en los dominios de investigación de reconocimiento de señal aunque también se ha usado en otros campos como la síntesis de señales, el reconocimiento de cadenas de caracteres, etc.

HTK es una librería que contiene módulos y herramientas escritas en el lenguaje C. Algunas de sus herramientas más destacables son el análisis de señal, el entreno de los modelos de Markov, el testeo y el análisis de los resultados. Soporta los modelos que

usan gausianas en dominio continuo y discreto, pudiéndose usar para construir complejos sistemas de HMM.

4.11. LNKNet

LNKNet es un software desarrollado con el objetivo de simplificar la aplicación de los más importantes patrones de *clasificación estadísticos*, de *redes neuronales* y de *maching learning*.

Implementado en *C*, también proporciona generación de código en *C* y en lenguajes de *scripting*, lo que permite que, tras entrenar un sistema, se pueda integrar el algoritmo ya ajustado, preparado para clasificar datos, dentro de cualquier sistema externo.

Es una aplicación desarrollada por el *MIT Lincoln Laboratory* con código público disponible en la página web de Lnknet [28].

Mediante un Script con parámetros de configuración uno ajusta el modo de generar y entrenar los Modelos Ocultos de Markov.

4.12. Compilador

El compilador que usaba Amadeus era g++/gcc (GNU C & C++ Compiler), mientras que CLAM se puede compilar con diversos compiladores. Opté por usar g++/gcc ya que de esta forma usaba el mismo compilador para ambos sistemas, aunque siempre teniendo en cuenta que el código debería funcionar correctamente bajo otros compiladores.

4.13. Editor Código Fuente

El editor que usaremos será Emacs o GNU Emacs, editor de texto altamente extensible y configurable creado por *Richard Stallman*, distribuido bajo la licencia libre GPL. En la actualidad es mantenido por la Free Software Foundation. Forma parte del proyecto GNU. Emacs es un editor potentísimo muy adecuado tanto para escribir texto plano como para programar o escribir scripts. Es extensible mediante el lenguaje elisp (Emacs Lisp), un dialecto interpretado de Lisp. A su vez incorpora una serie de aplicaciones que actúan como plug-in permitiéndonos realizar tantas acciones como si de una IDE se tratara. Hay plug-in para compilar con GDB, navegar por el árbol de directorios, etc.

Capítulo 5

Análisis

El paso previo que debemos realizar antes de empezar el diseño del sistema en CLAM, será detectar los servicios que queremos proporcionar y los problemas a los que nos podemos enfrentar y debemos ser capaces de solucionar.

Uno puede tener una idea aproximada sobre esos aspectos de análisis, pero sin embargo hay que dilucidar todo tipo de ambigüedad posible para conseguir un diseño completo, robusto y eficiente. Cuanto más completo sea el análisis, menos sobresaltos inesperados tendremos en el diseño. Nuestra única limitación es el tiempo global del proyecto, que nos hará repartir las diferentes etapas buscando el mayor equilibrio.

Debido a la naturaleza iterativa de la metodología de ingeniería de software que hemos elegido para el desarrollo del proyecto, los errores de una etapa se irán solucionando en iteraciones posteriores.

Los principales objetivos de esta etapa de análisis será:

- Observar los diagramas de casos de uso obtenidos en la etapa de análisis de requisitos y así definir cuales son los actores del sistema.
- Reconocer la conducta y responsabilidad de cada actor.
- Encontrar las relaciones que se van estableciendo entre los distintos actores.
- Definir una estrategia para abordar la fase de diseño.

Por otra parte, la técnica del *Refactoring* 4.3 desarrollada a lo largo de todo el proyecto, nos obliga a realizar un análisis del modelo de Amadeus, del cual queremos partir como base. Amadeus es una herramienta para reconocer música a través de los Modelos ocultos de Markov.

5.1. Observaciones

Observando el capitulo de análisis de requisitos 3 podemos inferir que nuestro sistema será usado como un módulo de la librería CLAM, aportando una funcionalidad más al desarrollo de aplicaciones para solucionar problemas del dominio del procesado de señal.

El usuario de CLAM accederá a la funcionalidad de la misma forma que accede al resto de la librería, a través lenguaje C++.

Analizando los diagramas de casos de uso se puede extraer que los principales elementos del sistema serán:

- Los objetos de procesado que conforman los diferentes algoritmos de extracción de parámetros y métodos específicos, como puede ser el de inferencia (Viterbi), etc.
- Los objetos de datos, que conforman la estructura de los Modelos ocultos de Markov dentro de CLAM, y otras estructuras complejas que han debido ser modeladas cuidadosamente, como puede ser los vectores de HMM (vhmm), etc.
- El sistema LNKnet usado para la generación de los modelos ocultos de Markov.
- El sistema de conversión de formato HTK a XML a través de la construcción de un parser.

El peso estructural y funcional más grande lo proporcionan los componentes de procesamiento de CLAM, Processing y ProcessingData, acorde al modelo de redes de procesos de CLAM llamado DSPOOM 2.1.

El principal actor que tendremos será el usuario de la librería, y en otros casos serán sistemas externos que usen las funcionalidades implementadas por los desarrolladores. Por esta razón una de las responsabilidades que tenemos que hacernos cargo es el de ofrecer medios para conectar una aplicación de CLAM con sistemas externos, a través de pasivadores o señales de sistema, aún siendo el usuario de la librería quien confeccionará el modo de comunicación con otros sistemas.

5.2. Análisis del modelo de AMADEUS

Amadeus es una librería creada con el fin de dar soporte a diversos proyectos como framework de trabajo sobre tecnologías que usen los modelos ocultos de Markov, en C++. El servicio que da esta librería es el de ofrecer las funcionalidades necesarias para reconocer música, para el entrenamiento de modelos y para clasificación (*Finger Printing*).

Vamos a analizar en detalle su estructura, sus módulos, y sus diagramas de clase subyacentes, de tal forma que nos sirva de base para "refractar" 4.3 su código.

Los módulos que nos van a interesar más son:

- Libbase, donde se define la estructura base de la librería.
- Libepar, donde se define la extracción de parámetros.
- Libhmm, donde se definen los modelos ocultos de Markov y los algoritmos asociados a ellos.
- Libprocess, donde se define la estructura base de los objetos que procesan datos.

Por otra parte existen otros módulos como libio, que gestiona la entrada y salida de datos, plots, etc. Pero son funcionalidades básicas que CLAM ya ofrece, por lo que vamos a centrarnos en el estudio de los cuatro módulos citados anteriormente para así entender el modelo de AMADEUS.

5.2.1. LibBase

Es el módulo que define la estructura base y la filosofía de encapsulación de datos y polimorfismo del framework.

La clase base del módulo y a su vez de toda la librería es *BufferBox*.

De ésta clase heredan todas las otras clases implementadas en la librería. Contiene dos atributos:

- *_buffer*, un puntero a números flotantes
- *_size*, un entero que informa del tamaño del *_buffer*.

Como se puede observar en la figura 5.1, hay tres clases fundamentales que derivan de BufferBox, donde *Capacitor* y *Multibufferbox* pertenecen al módulo libbase mientras que Process pertenece al módulo libprocess.

Al tener todas las clases derivando de BufferBox podremos aprovechar los aspectos de polimorfismo ya que toda clase podrá ser entendida del tipo BufferBox. Éste fenómeno de polimorfismo sucede también en CLAM, por ejemplo a través de la clase ProcessingData de la cual derivan todos los objetos de datos.

La clase *Capacitor* está diseñada para almacenar las muestras contenidas en un fichero audio. A parte de los atributos de BufferBox, contiene los atributos:

- *_length*, que informa del número de muestras que falta para llegar al final del fichero.
- *_step*, que informa cuántas muestras debemos desplazarnos para iniciar la siguiente lectura del fichero.

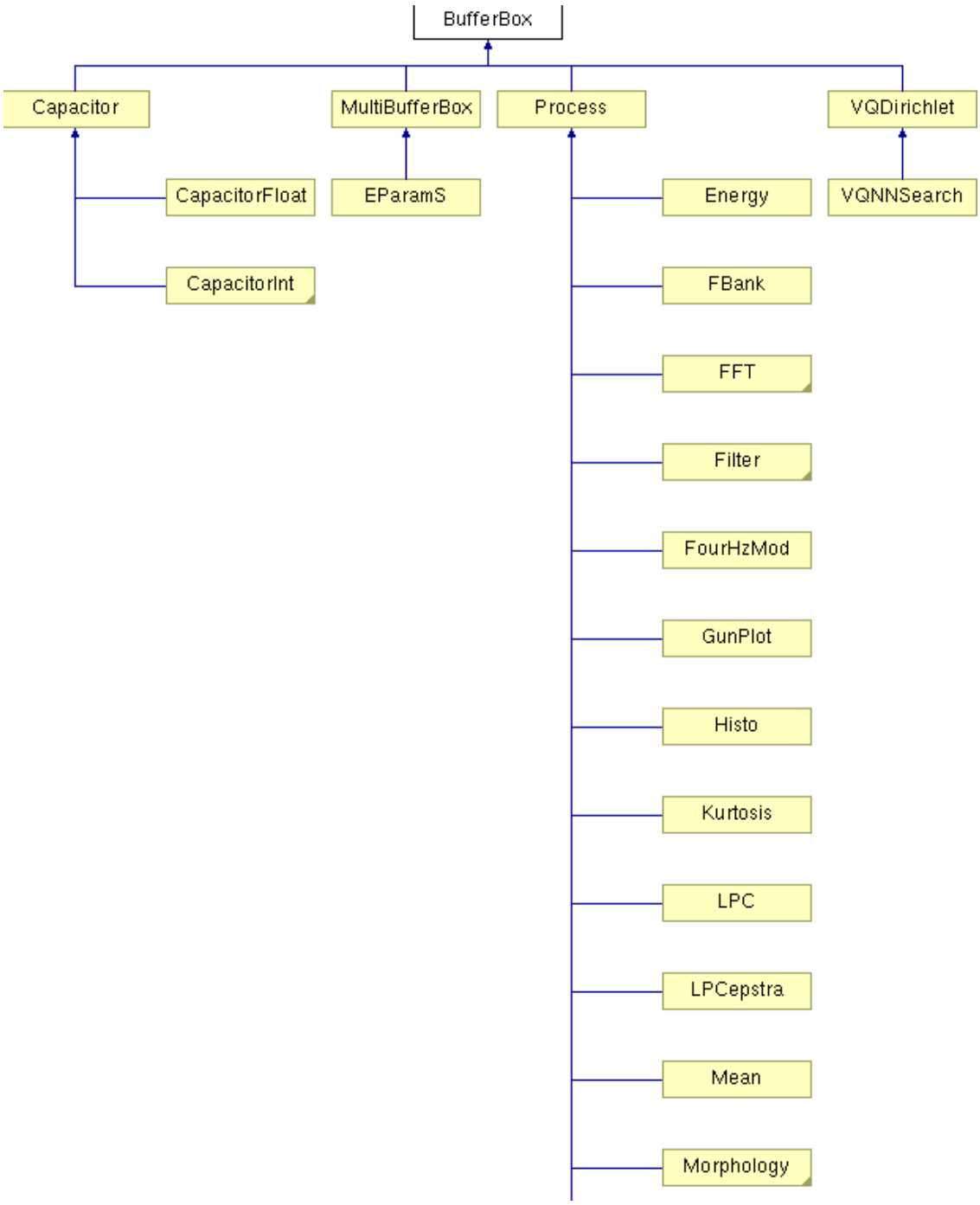

Figura 5.1: Diagrama de Clases de Amadeus que heredan de Bufferbox

De esta forma se va cargando en memoria los datos, se va desplazando el puntero de lectura _step muestras, se resta _step a la variable _length, y se sigue leyendo hasta el final del fichero.

En el módulo libio se encuentra la implementación de la lectura del header de los ficheros wav, propiedades tales como la frecuencia de muestreo, etc. Sin embargo una de las limitaciones más palpables de Amadeus es la única posibilidad de leer ficheros de audio codifcados con 16 bits por muestra. Fue realizado de esta forma por razones de optimización puesto que no era necesario una codificación más compleja para las funciones que debía soportar el framework, ya que codificar la señal con más bits aportaba información redundante.

Otra clase importante dentro de la estructura de Amadeus es el MultiBufferBox, que como se puede inferir de su nombre, su principal característica es que contiene un vector de BufferBox. De esta forma ofrece la posibilidad de conservar el polimorfismo también para estructuras más complejas que necesiten de formas matriciales. Es lo que ocurre con clases como EparamS, que para almacenar los datos que genera necesita una matriz, puesto que genera un BufferBox por cada frame que analiza. EparamS es la principal clase del módulo libepar que más a delante detallaremos.

Por último, la otra clase trascendental en la estructura del framework es Process, de la cual derivan todo tipo de clases que procesan datos.

5.2.2. LibProcess

La clase principal de este módulo es *Process*. Todas las clases que deriven de Process deberán implementar el método *DoIt(BufferBox & pBuffer)*, al cual se le puede pasar cualquier tipo de datos, realizando en caso necesario un "cast" si nos interesa solucionar casos de polimorfismo.

Otro método importante es el *PreDoIt()* que sirve para preparar los datos de tal forma que el *DoIt()* pueda realizar los cálculos de forma coherente.

Por lo tanto todos los objetos que procesen datos deberían heredar de esta clase, sin embargo dentro de la librería no se realiza una separación exhaustiva entre objetos que poseen datos y objetos que los procesan. De esta forma tienden a ser más oscuras conceptualmente algunas clases que contienen las estructuras para almacenar datos y a su vez contienen algoritmos para hacer los cálculos sobre ellos mismos. Esto sucede en clases como Vhmm, mix, etc.

5.2.3. LibEpar

Aquí es donde está modelado e implementado todo el proceso de extracción de parámetros. Dentro del módulo están implementados todos los filtros y algoritmos necesarios, FFT, banco de filtros, MFCC, etc. Todas las clases son derivados de Process, ya que todos

son algoritmos que procesan datos.

El núcleo del módulo es la clase Eparams, responsable de organizar la extracción de parámetros. Dentro contiene los filtros necesarios, y las variables de configuración:

- *SampleFreq*, un entero que representa la frecuencia de muestreo.
- *SizeParams*, define el número de parámetros con el que queremos representar un frame.
- *DecimateCounter*, define cada cuántos frames generar un resultado, haciendo la media de los últimos "DecimateCounter" frames, según el valor que le hayamos asignado.

Estos atributos se definen en el constructor y no se pueden variar en el ciclo de vida del objeto. Es poco flexible y en el caso de querer posibilitar la opción de variar configuraciones dentro del ciclo de vida del objeto, tendremos que implementar un exhaustivo control de coherencia de datos dado la complejidad de la clase.

5.2.4. LibHmm

Éste es el módulo donde se implementan todas las estructuras y algoritmos relacionados con los modelos ocultos de Markov necesarios para el reconocimiento de musica. Dichas estructuras están optimizadas para la topología de modelos que usa Amadeus, modelos de un solo estado, con una única función gausiana, y sólo con posibilidad de transición hacia la derecha. No soporta otro tipo de topología.

Implementa las gausianas en la clase *mix*, los estados en la clase *sshmm*, que a su vez es el modelo completo, puesto que como comentamos anteriormente, los modelos ocultos de Markov en Amadeus tienen sólo un estado.

Los modelos ocultos de Markov generados en la fase de entrenamiento, son cargados a través de la clase *ghmm*, que corresponde a un vector de modelos.

Para guardar la generación de descriptores se implementa la clase *vhmm*, que equivale a un vector de identificadores, que al fin y al cabo son estados concatenados. Se almacena un identificador de cada modelo, en el orden adecuado. Cuando tenemos una base de datos de descriptores, usamos la clase *gvhmm*, que equivale a un vector de *vhmm*, como si fuese una matriz donde cada fila es un descriptor asociado a una señal específica.

Todas estas estructuras derivan de Process, cuando semánticamente sería más idóneo hacerlos derivar de BufferBox ya que son estructuras para guardar datos. Sin embargo algo que ofusca el modelo es que dentro de cada clase se implementan métodos que procesan sus propios datos, por lo que dificulta el entendimiento de algoritmos que usan estos cálculos incrustados. La lectura de los algoritmos resulta laberíntica, puesto que los cálculos están encapsulados dentro de clases que uno intuye contengan tan sólo datos, haciendo algunos objetos los cálculos sobre sus propios datos.

Es lo que sucede con los algoritmos que implementan el método de inferencia propuesto por Viterbi. Éstos algoritmos, tanto el libre de gramática como el sujeto a una gramática son muy oscuros, puesto que no se sabe de donde sacan los cálculos si uno realiza una navegación exhaustiva por todo el módulo. Los algoritmos se implementan en las clases *ViteGram* y *ViteFree*, ambos heredando de la clase abstracta *Viterbi*.

5.2.5. Libio

Es el módulo de entrada y salida de audio. Solo soporta el formato wav de 16 bits. Es una limitación debida a que el diseño se ha realizado para casos de uso muy específicos. Una codificación con más bits por muestra no aportaría información trascendente para los objetivos descriptivos que se pueden tener con el framework, aportando mayor peso a la representación de las señales y disminuyendo la eficiencia del sistema.

5.3. Conclusiones

Habiendo observado con detenimiento los diversos aspectos que debemos tener en cuenta, previos al diseño del sistema, podemos destacar una serie de líneas importantes de desarrollo para la siguiente etapa.

El diseño se dividirá en cuatro apartados funcionales independientes pero a su vez relacionados. Independientes en el sentido de que siguiendo un orden específico de desarrollo se los puede tratar por separado, y relacionados a causa de que sin el desarrollo de algún apartado funcional, no se puede empezar el siguiente.

A su vez como unos influyen sobre otros por el uso que hacen de su funcionalidad, al ser iterativo el desarrollo, se irá adaptando acorde a las necesidades de cada componente, prestando atención al análisis de requisitos funcionales.

Recordemos por ejemplo que la componente de *extracción de parámetros* es usada por las componentes de *Entrenamiento*, *Generación de descriptores* y *Reconocimiento*. De tal forma que el primer diseño que desarrollaremos será el de ésta componente funcional.

Luego seguiremos con el componente de *Entrenamiento*, puesto que sus resultados son usados para la *Generación de descriptores* y el *Reconocimiento*. Acto seguido iniciaremos el diseño de la Generación de descriptores para finalizar con el *Reconocimiento*, que hace uso de todos los componentes funcionales anteriormente citados.

En lo que respecta a la técnica del *refactoring*, tras analizar Amadeus concluimos que es una librería diseñada y optimizada para casos de uso muy específicos, razón por la cual faltan muchos aspectos de control de flujo, diversidad de tipos de datos, etc. Es un framework diseñado e implementado para casos de uso relacionados únicamente con los Modelos ocultos de Markov, razón por la cual consigue un funcionamiento eficiente en razón de velocidad de ejecución.

Es un reto realizar con CLAM un diseño, respetando la filosofía del modelo DSPOOM 2.1, posibilitando un uso más general y a su vez manteniendo la eficiencia, resultando muy interesante para valorar el trabajo realizado, comparar la eficiencia de ambos sistemas.

Capítulo 6

Diseño e Implementación

En el diseño se recoge todo lo estudiado anteriormente para aquí iniciar la construcción del sistema, concluyéndolo más a delante con la implementación. En este capítulo colocaremos las "vigas" principales del sistema, así como podemos pensar que lo hecho anteriormente consistía en allanar el terreno y poner los cimientos, para finalizar la construcción del sistema completo en la fase de implementación

Esta etapa de desarrollo de creación del diseño, respetará los requisitos no funcionales, los objetivos y hará uso de las herramientas descritas anteriormente enmarcados en el dominio que hemos analizado.

A su vez, en la mayoría de desarrollos de sistemas, el diseño y la implementación están fuertemente correlacionadas por lo que se suele tratar como una sola unidad.

Como hicimos anteriormente en la fase de análisis de requisitos, dividiremos el diseño en cuatro componentes funcionales: *Extracción de Parámetros, Entrenamiento, Generación de Descriptores y Reconocimiento.*

Dado su naturaleza iterativa el diseño y la implementación que vamos a exponer es la que corresponde a la última iteración del ciclo de vida del proyecto. En otro apartado describiremos los principales cambios que ha ido sufriendo el sistema.

6.1. Reglas de Diseño

A pesar de la división que hacemos en el desarrollo del diseño, hay una serie de partones o líneas filosóficas que regirán sobre todas las clases implementadas.

Una de éstas líneas será la implementación de los dos modos de uso que ofrece el modelo de CLAM, el supervisado y el no supervisado. De esta manera se habilitará un uso válido tanto para la llamada al Do con parámetros que al de sin parámetros. En este caso, el de usar la llamada del Do sin parámetros, hemos tomado la decisión de pensar que está siendo usado por una red de procesos dinámica (*Network*) y dado que la elección del

uso de este tipo implica un proceso semiautomatizado, hemos implementado un control más exhaustivo sobre las configuraciones en algunos casos donde era necesario. Por esta causa todos los *Processings* que se diseñen tendrán habilitados los puertos en caso de que sean necesarios.

Otro punto importante fue el de ser conciente que tanto el diseño como la implementación tenían que ser lo más comprensibles posible cumpliendo la "estética" del modo de desarrollo del equipo de CLAM. Por esa razón se ha cuidado que el código sea legible y comprensible, añadiendo comentarios y estructurando de forma clara el código.

No hay que olvidarse que el principal actor hacia quien está dirigido el proyecto es hacia las personas, más concretamente programadores, que agradecerán un código "vistoso".

Para asegurar que cada nueva clase funciona correctamente, a cada una se le creará un Test que compruebe su validez 4.9. De este modo cuando el sistema crezca bastará con pasar el juego de Tests para encontrar un posible error evitando arrastrar situaciones equívocas.

6.2. Extracción de Parámetros

Resulta trascendental para el desarrollo de todo el sistema crear un diseño e una implementación adecuada de este componente funcional.

El componente va a estar dividida en diversos Processing que realicen cálculos específicos y un ProcessingComposite que los agrupe, tal que si uno quiere, pueda usar dichos objetos de procesado de forma independiente. La repartición de los cálculos en cada Processing va a ser buscando crear los núcleos de cálculos mínimos con semántica, es decir, que sus resultados tengan un mínimo significado.

Por otra parte el ProcessingComposite va a ser el encargado de crear la red de procesos optimizada para los casos de uso analizados, con el objetivo de facilitar su uso al usuario. De esta forma los usuarios inexpertos en el campo de la extracción de parámetros o que no deseen conformarse su propio módulo, podrán usar de una forma transparente y sencilla la extracción de parámetros que proponemos. El ProcessingComposite se preocupará de mantener la coherencia de los datos, conformará el orden de ejecución y creará su propio control de flujo de los datos.

Dicho concepto de ProcessingComposite, viene diseñado e implementado en la clase llamada EparamS.

Antes de profundizar en esta clase, definimos el diseño de cada objeto de procesado que formaría parte de esta red estáticaza que sería imposible confeccionar la clase EparamS sin tener previamente todos sus subcomponentes.

De esta forma iremos describiendo los detalles de diseño específicos que hemos tenido en cuenta clase por clase, menos en algunas que no entrañan más complejidad de decisión que los rasgos generales definidos.

De esta forma la clase Energia no tiene ningún punto de decisión crítico.

Sin embargo el concepto Filtro ha sido modelado analizando los distintos filtros que necesitaba el sistema, hallando los puntos en común de todos ellos para poder conformar una primera abstracción hábil para representar rasgos que comparten. En nuestro caso hemos analizado los filtros de aceleración, derivada y preémfasis, encontrando una estructura común de ellos y de la arquitectura de los filtros en general, para así posibilitar la creación de nuevos con un método sencillo, asegurando la escalabilidad del componente.

Otro concepto importante que deriva del anterior es la composición de multifiltros, una cadena de filtros que compartan un comportamiento en común actuando sobre un mismo fichero audio.

El diseño que proponemos para solucionar estos conceptos es el ilustrado en la figura 6.1.

De tal forma la clase *Multifilter* será polimórfica en tiempo de compilación, templatizada por la clase *Filter*. El atributo que contiene el vector de filtros está templatizado, pudiendo tomar la definición de Filter o de cualquier clase que herede de ella. En nuestro sistema hay implementadas las clases *Accel*, *PreEmphasis* y *Derivate*, como extensiones de *Filter*. Por lo tanto después de templatizar el Multifilter, tan sólo deberemos configurar el tamaño, cuántos filtros lo compondrán. Hay que tener en cuenta de que al estar compuesto por varios *Processing*, Multifilter se convierte en un *ProcessingComposite* controlando el funcionamiento y la configuración de sus subprocesos.

Tal como es el diseño de Filter, las clases que quieran heredar de ella e implementar filtros diferentes, tendrán que:

- Implementar el método virtual *DoPost()* que es llamado por el método *Do* ejecutor de la funcionalidad de las clases. A su vez La clase Filtro tiene declarado unos cálculos que realizan todos los filtros, de tal forma que se realizan estos cálculos previamente a los que se implementen en el *DoPost()*. De esta forma es como se implementa la clase *PreEmphasis*.

- Cambiar las configuraciones de los parámetros del filtro. Al tener implementados los cálculos báscios comunes al concepto filtro, tan solo cambiando sus configuraciones uno puede generar diferentes filtros. Es el caso de cómo hemos implementado los filtros *Accel* y *Derivate*.

La *FFT* que usamos es la implementación *FFT_ooura* ya que corresponde al algoritmo usado por el sistema de Amadeus. Preferimos usarlo para poder comparar el funcionamiento del mismo algoritmo bajo los dos sistemas, el de CLAM y el de Amadeus. El banco de filtros se modela dentro de la clase *Fbank*, teniendo que configurar el número de filtros que uno desea tener (lo cual condicionará con cuantos filtros uno representará el espectro), el número de bins del espectro entrante y la frecuencia de muestreo. El banco de flitros descompone la señal en sub-bandas de frecuencias, tantas como indiquemos en

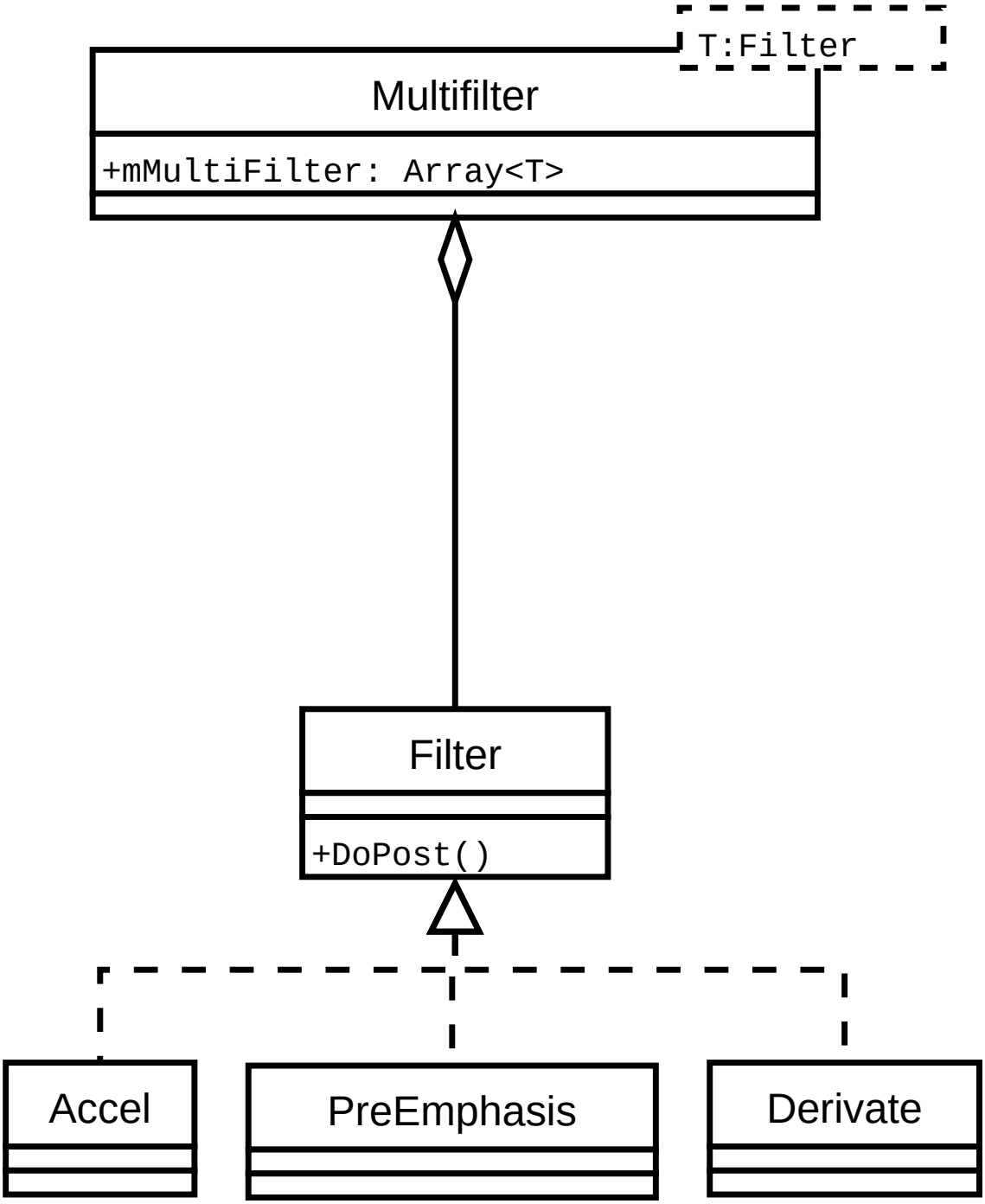

Figura 6.1: Diagrama de Clases del concepto *Filtro*

la configuración de la clase, teniendo en cuenta distribuciones específicas de energía de la señal en el dominio frecuencial 6.2.

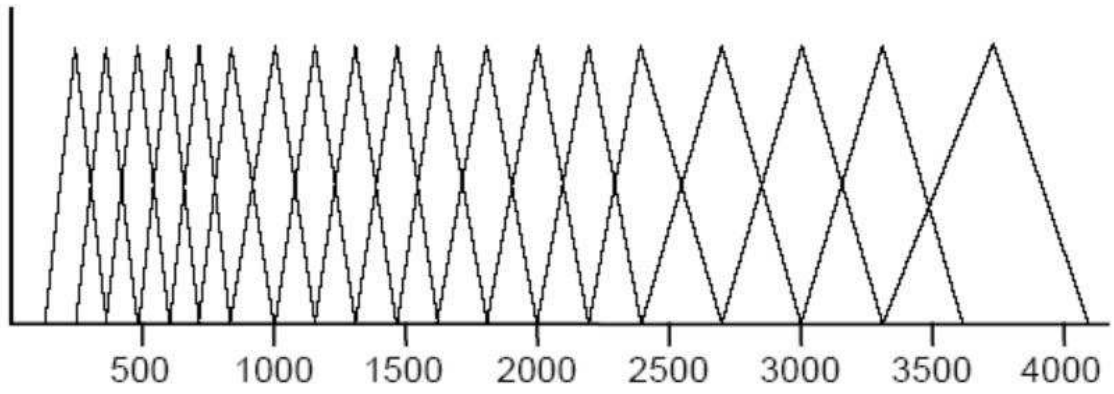

Figura 6.2: Diagrama de un Banco de Filtros

La *Transformada Discreta de Coseno* se modela en la clase *DCT* que a su vez hereda de la clase *Project* 6.3. Dentro de la clase *Project* hemos implementado los cálculos realizados por las transformadas, de tal forma que si quisiésemos crear distintas transformaciones, lo único que debemos hacer es implementar el método *createSpace()* creando una tabla que caracterice los cálculos. *Project* es una clase abstracta, forzando así a las clases hijas a implementar dicha tabla. Se necesita configurar la longitud de los datos de entrada y la longitud de los datos que deseamos generar.

Para realizar la media de un conjunto de datos hemos diseñado una clase *Mean* que guarda la media de los datos que ha ido calculando de tal forma que si uno le hace calcular la media de otros datos, si no se ha reseteado su estado, hará la media con todos los datos que se le han ido pasando hasta ese momento, es un filtro con memoria de estado. En el momento de querer hacer medias de conjuntos entre conjuntos, es decir, hacer media de vectores, se usará un conjunto de clases *Mean*, de tal forma que haya tantos *Mean* como posiciones tenga cada vector; calculando con el primer *Mean* la media de todos los valores de los vectores en la posición 0, luego con otro *Mean* calcular la media de la posición 1, etc. Así hasta llegar a la última posición. Ante esta necesidad se decidió componer la clase *MultiMean*, un ProcessingComposite de subprocesos *Mean* . De esta forma *MultiMean* gestionará todos sus subprocesos *Mean*, habiéndole de configurar la cantidad de subclases *Mean* que se desea tener 6.4.

Volviendo a la clase *EparamS*, los parámetros claves para su configuración y que afectan a su vez a las configuraciones de sus subprocesos son:

- *prSampleRate*, que configura el *SampleRate* del audio que se analiza, de tal forma que si cambia, EparamS se encarga de ajustar todos los procesos hijos que dependen

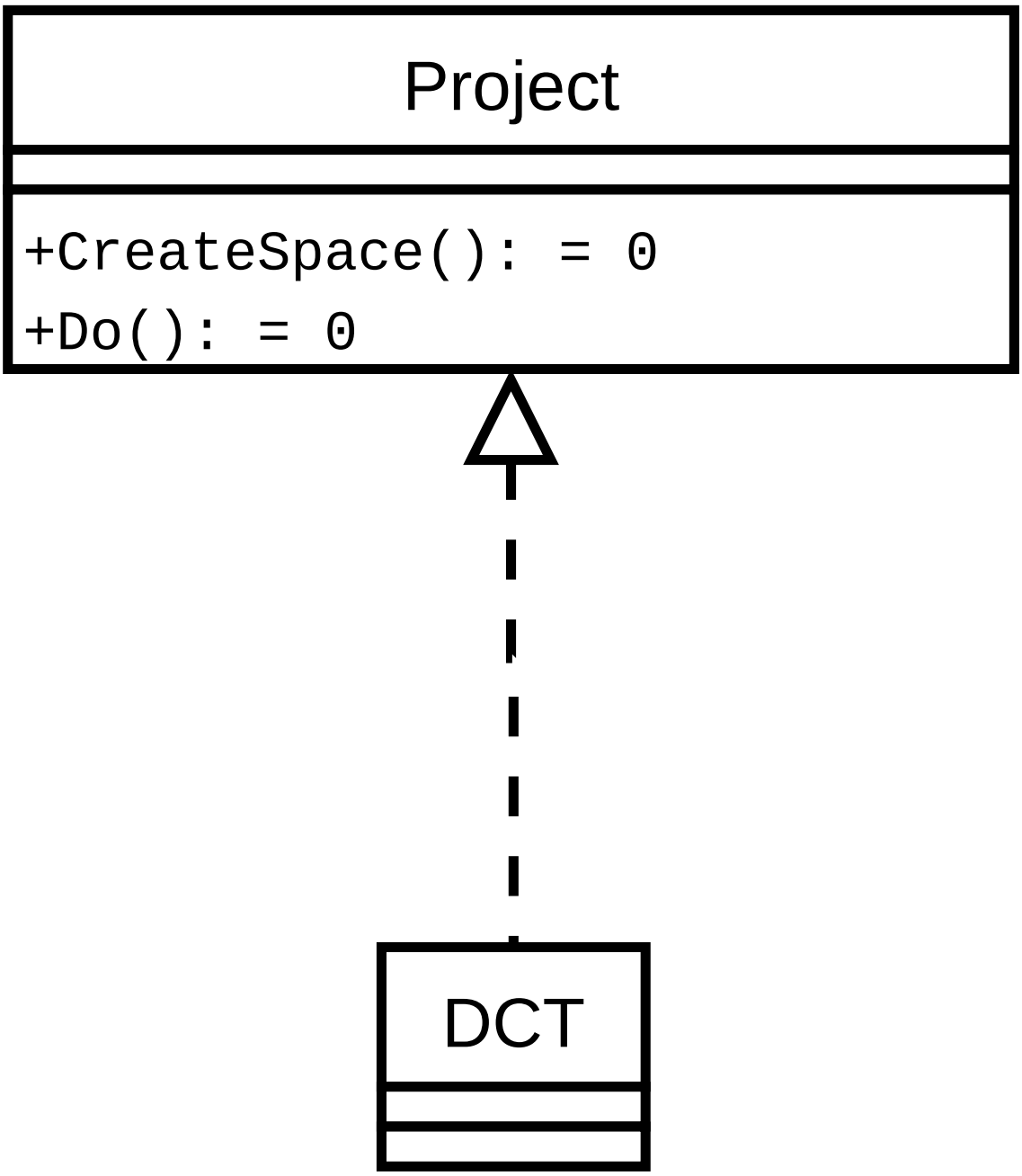

Figura 6.3: Diagrama de clases de la Transformada Discreta de Coseno

de este dato. También controla la coherencia de las configuraciones cuyos valores dependen del *SampleRate* como pueden ser el tamaño de ventana, etc.

- *Step*, que configura el número de milisegundos que avanzaremos el puntero de inicio

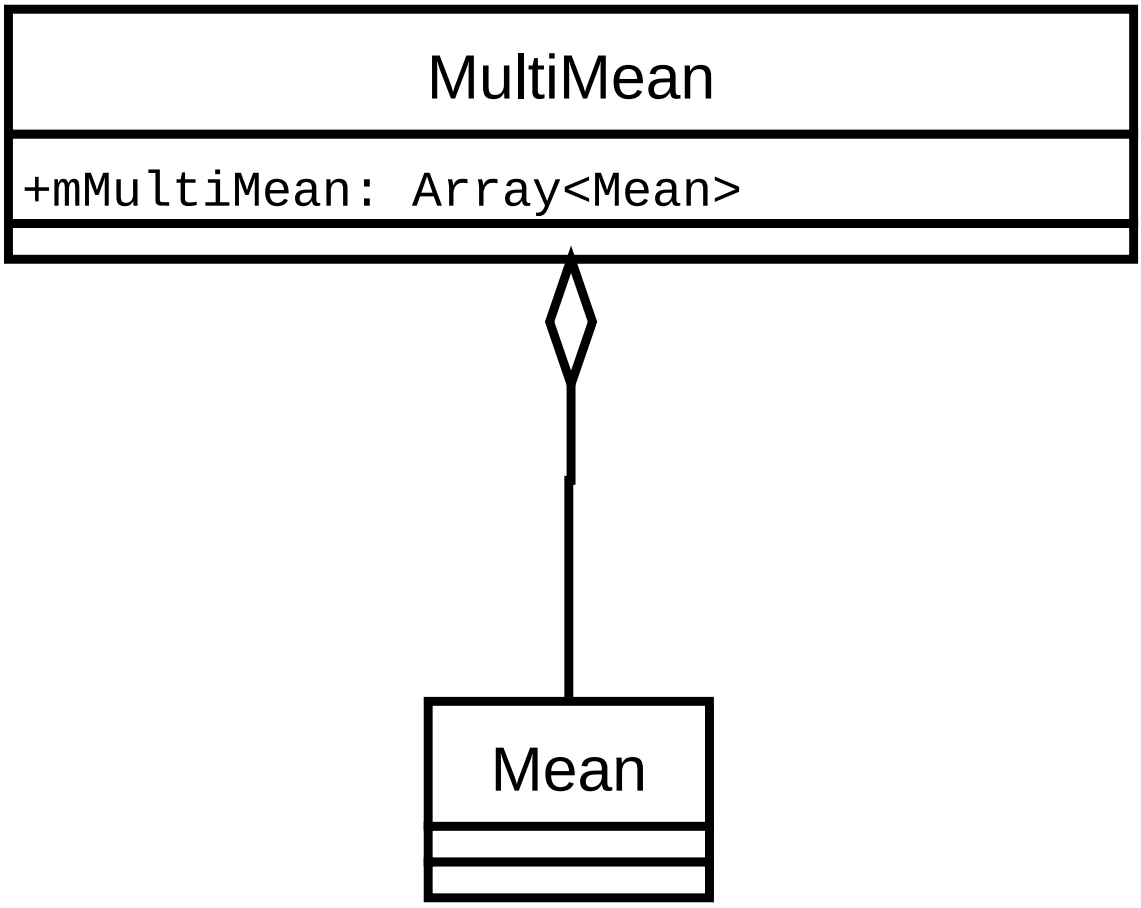

Figura 6.4: Diagrama de clases de MultiMean

de cada frame de la señal.

- *prWindowLength*, que configura el tamaño en milisegundos de cada frame en el que subdividiremos la señal, avanzando la ventana step milisegundos.

- *prSizeParams*, que define el número de parámetros con el que se desea representar cada frame.

- *NDecimate*, que define cada cuántos frames generar un resultado, haciendo la media de los últimos "NDecimate" frames, según el valor que le hayamos asignado.

El aspecto poco flexible que presentaba Amadeus para los posibles cambios en tiempo de ejecución del objeto de extracción de parámetros queda solucionado en CLAM con el uso de los *Controles* y el control sobre las dependencias de las configuraciones.

A través del diagrama 6.5, se puede observar el funcionamiento interno del algoritmo de extracción de parámetros implementado en la clase *EparamS*.

6.3. Estructura de Datos de HMM

Antes de abordar cualquier otro componente funcional diseñando los algoritmos necesarios, debemos solucionar el diseño de las estructuras de datos que modelarán los modelos ocultos de Markov, ya que todos los otros componentes se sirven de ellas.

La estructura la dividiremos jerárquicamente, de tal forma que respete los conceptos de que un modelo contiene una lista de estados, que a su vez contiene una lista de canales y que a su vez contiene una lista de funciones probabilísticas gausianas.

Dichas relaciones de pertenencia se modelarán como atributos dinámicos. Por ejemplo en la clase que represente un modelo oculto de Markov habrá un atributo dinámico que será una lista de estados. Es importante implementar dicha lista como " DYN_ATTRIBUTE" de CLAM, ya que genera automáticamente los accesores y los pasivadores en formato XML 4.6.3. Así definiremos todos los datos que consideremos relevantes en la pasivación y recuperación de los modelos de Markov.

La idea general puede ser observada en el siguiente diagrama 6.6.

Concretamente la clase *HMM*, tendrá como atributos dinámicos el *nombre*, el *número de estados*, la *lista de estados* y la *matriz de transición*. Además tendrá otros atributos irrelevantes para la representación de los modelos pero que sin embargo sirven de indexadores o de accesores, reduciendo el tiempo de búsqueda de cada estado. Para conseguirlo se declarará un mapa de punteros a la lista de estados, aumentando así la velocidad de acceso.

La matriz de transición vendrá modelada por una matriz dinámica que también nos facilitará la pasivación de sus datos.

Los estados modelados dentro de la clase *HMMState*, tendrán como atributos dinámicos un *índice* (que sirve para el mapa de punteros de la clase HMM), una *lista de canales*, un *vector con las dimensiones* de cada canal, otro *vector con los pesos* y otro con las *duraciones* de cada uno.

Los canales vienen modelados en la clase *HMMStream* conteniendo como atributos dinámicos un *índice* (que sirve para ser indexado desde la clase HMMState) y una *lista de gausianas*.

Por último las gausianas están modeladas dentro de la clase *HMMMixture*, que contiene el *índice* útil para la clase HMMStream, el *peso de la gausiana* y los valores útiles con que se modela la *función probabilística*.

Dichos valores están contenidos dentro de la clase *ContinousPDF* que contiene una *constante*, un *vector de medias* y una *matriz de covariancias*. Dicha matriz está modelada

dentro de otra clase que contiene la *variancia* y la *matriz inversa de covariancias*. De esta forma se realizan unos precálculos que agilizan la evaluación probabilística de las observaciones.

Una vez estructurado el diseño de un Modelo Oculto de Markov, es necesario que se diseñe una clase que gestione un grupo de modelos de Markov ya que en muchos escenarios sucede el tener conjuntos de ellos. Con ese fin se diseñó *HMMGroup*, clase que contiene como atributo dinámico una lista de modelos y además indexadores como el mapa de punteros.

Con éste diseño se consigue una fácil pasivación y activación de los datos en XML. Pasivando un objeto de HMMGroup se pasivarán recursivamente todos sus atributos dinámicos de forma jerárquica, es decir, pasivará la lista de HMM, que a su vez pasivará todos sus atributos dinámicos y así hasta la última capa. Lo mismo servirá para la activación.

Éste diseño va a condicionar el diseño del parser de HTK a XML, puesto que tendremos que adecuarlo al XML que generan o aceptan nuestras estructuras.

6.4. Parser

La construcción del Parser se ha hecho siguiendo las especificaciones de la gramática de HTK que se pueden encontrar en el manual de la librería [35]. Por lo que el diseño e la implementación de este módulo ha consistido en la adaptación de la gramática al lenguaje Flex y Bison.

6.5. Generación de Descriptores

En cuanto al diseño de esta componente funcional cabe destacar dos puntos básicos:

- El diseño del algoritmo de Viterbi
- El diseño de los vectores de Modelos de Markov

Una vez se han diseñado e implementado las estructuras de datos para representar los Modelos ocultos de Markov y el algoritmo de extracción de parámetros, el módulo más importante que falta para conseguir generar los descriptores es desarrollar el algoritmo de Viterbi específico y las estructuras de datos necesarias para salvar dichos descriptores.

De esta forma teniendo en cuenta ciertos rasgos concretos del escenario que nos concierne, el algoritmo de viterbi se va a diseñar de forma optimizada para modelos ocultos de Markov libres de gramática. Para simplificar los cálculos, sabiendo que los modelos con que vamos a tratar son de un solo estado, Amadeus computaba la probabilidad de un modelo como la probabilidad de su estado 5.2.4.

Como primera versión se ha implementado este método, quedando como futuro trabajo adaptar el algoritmo para tolerar la evaluación de modelos con más de un estado. Con esta visión se ha diseñado el algoritmo, creando las estructuras y los bucles pensando en esta futura ampliación, faltando solamente la fórmula para computar modelos de más de un estado.

El dato que se guarda después de evaluar la probabilidad de las observaciones, es qué modelo maximiza dicha probabilidad, es decir se guarda el modelo que ha resultado una probabilidad máxima con dichas observaciones. Las observaciones que computa es el resultado de la extracción de parámetros realizada sobre la señal, es decir, una matriz de observaciones donde cada fila es un conjunto de observaciones, siendo la señal representada en su totalidad a través de todas las filas de la matriz. El algoritmo detecta que modelo es el que maximiza las observaciones de la fila 1, luego el de la fila 2 y así sucesivamente hasta llegar al final. De esta forma se genera como resultado un vector de modelos ocultos de Markov, donde dichos modelos son los creados en la fase de entrenamiento. Este vector representa los cambios a lo largo del tiempo que va sufriendo la señal.

La problemática que surge a continuación es cómo guardar y representar dichos vectores de modelos, puesto que tendremos que pasivarlos para crear una base de datos, razón por la cuál tenemos que conseguir una representación eficaz y con un peso de datos minimizado.

La solución propuesta, como se puede observar en la figura 6.7, es la de diseñar dos ProcessingData, uno codificando el concepto de vector de modelos (Descriptor de una señal) y otro conteniendo un vector de dichos Descriptores, emulando una Base de Datos y a través de la cual pasivaremos y activaremos todos los descriptores. Por lo tanto en primer lugar tenemos la clase Vhmm con tres atributos dinámicos básicos necesarios para representar un descriptor:

- *Model*, que consiste en un vector de números enteros, siendo cada número el modelo al que corresponde de la base de datos. De esta forma tan sólo se guarda un entero en vez de toda la estructura citada anteriormente con la que representamos los Modelos Ocultos de Markov. El orden es creciente, es decir, el primer modelo es el de la posición 0. También es fácil detectar una corrupción de memoria puesto que sabiendo la cantidad de modelos que uno tiene en la base de datos, uno puede saber cuál es el rango de valores que puede tomar los enteros del vector.

- *StatesXmodel*, es un vector de enteros cuyos valores representan cuántos estados tiene el modelo en la posición homónima del vector *Model* anteriormente citado. Es importante saber éste dato para realizar la concatenación de los modelos, problema que no consideraban en Amadeus puesto que aceptaban sólo modelos de un estado, por lo que en la práctica concatenar modelos era concatenar estados.

- *Nom*, es una cadena de caracteres en donde se guarda el nombre de la señal descrita; es decir, si generamos el descriptor de la canción “balada para un loco”, guardamos su nombre para poder saber qué descriptor está asociado a dicha canción.

En Amadeus tenían la necesidad de registrar como atributo de la estructura Vhmm el número de modelos que contenía cada Descriptor. Este atributo era un entero y lo necesitaban para realizar la reserva de memoria para la carga del fichero. En nuestro caso, en el espacio físico del fichero sí tendremos registrado éste dato, ya que XML lo genera automáticamente, sin embargo cuando carguemos en memoria la base de datos, allí no tendremos ese dato ocupando espacio ya que es usado solo para reservar memoria para la estructura que queremos recuperar sin reservar los 32 bits necesarios para grabar su valor. En grandes bases de datos el espacio que se ahorra es considerable. Por lo tanto en CLAM, con éstos tres atributos basta para representar la estructura de un descriptor, sin embargo para poder pasivar y activar un conjunto de descriptores necesitamos diseñar otra clase cuyo atributo sea un vector de *Vhmm*. Ese es el caso de la clase *Gvhmm*, que declarando su vector como atributo dinámico adquiere de forma automática la rutina de pasivación, activación y accesores.

Para agilizar el proceso de generación de descriptores para una cantidad considerable de canciones, evitando tener que escribir cada vez la canción que uno desea registrar, se ha creado una clase que lea el fichero que contiene la lista de canciones, cargando los nombres de forma estructurada en una lista de *strings*.

En el diagrama de secuencia 6.8 se puede observar como interactúan los distintos objetos en un ejemplo de un Cliente que genera descriptores.

6.6. Reconocimiento de Música

En cuanto al diseño de ésta componente funcional, el único módulo que se debió añadir fue el de un algoritmo de Viterbi modificado capaz de encontrar qué Descriptor corresponde a la señal de entrada.

Este concepto se modeló dentro de la clase *ViteGram*, que como se entiende por su nombre es el algoritmo de *Viterbi* adaptado para modelos con gramática. Otra particularidad, igual que en el desarrollo del algoritmo de *ViteFree*, son los datos que se aprovechan para generar los resultados. De tal forma hemos iniciado implementando el algoritmo que soporte la gramática de los modelos de Amadeus, preparando las estructuras para poder soportar las restricciones de gramáticas específicas.

La clase *ViteGram* tiene tres parámetros de configuración:

- *FramesVite*, que define cada cuántos Frames concatenados uno debe encontrar un descriptor que los represente.
- *RecoThreshold*, un límite de validez, si la probabilidad más alta encontrada es menor a dicho límite se da el resultado por nulo.
- *ScoreDifThreshold*, es otro límite de validez, si la probabilidad más alta y la segunda más alta tienen una diferencia menor a dicho límite, el resultado se da también por nulo.

Mediante el ajuste de dichos parámetros se permite una flexibilidad y adaptabilidad a distintos escenarios.

Antes de ejecutar su funcionalidad se le tiene que asignar los Modelos Ocultos de Markov resultantes de la fase de entrenamiento y los vectores de Modelos resultantes de la fase de generación de Descriptores.

Una vez ejecutada su funcionalidad, inicia a buscar el descriptor más idóneo a la entrada. Genera un descriptor aleatorio y evalúa su probabilidad, usando dicho resultado para evaluar la validez del resto de descriptores, restanado al resultado de los dos descriptores más probables, el resultado del descriptor aleatorio para así ajustar más el límite de validez de *RecoThreshold*.

El método para evaluar la probabilidad de que un descriptor haya generado dicha señal consiste en:

- evaluar el frame de entrada y crear un vector de probabilidades con tantas posiciones como modelos resultantes del entrenamiento haya habido. De esta forma se evalúa la probabilidad de cada modelo; guardando cada vector hasta tener el mínimo de frames definido por la variable de configuración *FramesVite*.
- Una vez tenemos este conjunto de vectores con las probabilidades de cada modelo, el paso siguiente consiste en crear otro vector del mismo tamaño que el del descriptor que queremos evaluar. Dicho vector se podría interpretar como el paso del tiempo. Cogemos el número del primer modelo del Descriptor (vector de modelos) y guardamos su probabilidad en la posición 0. Así hasta el último modelo del Descriptor.Dicha probabilidad proviene de la evaluación de cada modelo anteriormente citada.
- Se repite con el siguiente frame el paso anterior sumando el resultado (hay *FramesVite* Frames para evaluar). Como se supone que el siguiente frame que se evalúa es un espacio de tiempo avanzado al anterior, la suma se realiza teniendo en cuenta dicho desplazamiento.

Por último se guarda el valor máximo del vector y la posición en donde ha habido este máximo.

Este método, ejemplificado en la figura 6.9, se aplica a todos los descriptores de la base de datos y de ésta forma se encuentra el descriptor más probable. Raras veces la probabilidad será 1, porque es difícil que la señal que uno esté intentando reconocer sea idéntica a la señal con la que se generó el descriptor. De esta forma es como se entiende la tolerancia a ruidos, como locutores hablando encima de una canción, etc. Puesto que acertará con el descriptor aunque las probabilidades de todos ellos sean más bajas ya que el ruido afecta a la evaluación de todos los descriptores por igual.

En el diagrama de secuencia 6.10 se puede observar como interactúan los distintos objetos en un ejemplo de un Cliente que quiere reconocer una señal.

6.7. Entrenamiento

Para la componente de Entrenamiento, por falta de tiempo, hemos decido usar el sistema LNKNet para generar y entrenar los Modelos Ocultos de Markov quedando como futuro trabajo el diseño y la implementación de esta componente en CLAM.

6.8. Cambios desde la primera iteración a la última

Con el paso de las diferentes iteraciones hubo algunos aspectos que cambiaron.

Uno de estos aspectos que condicionó mayores cambios fue el diseño del *DescriptorSet*, puesto que se discutió diferentes alternativas de diseño aportadas por diferentes proyectos. Una de dichas alternativas fue proveniente de éste proyecto por su necesidad de uso aunque se haya terminado prescindiendo de él.

El concepto de *DescriptorSet* es el de una clase que una Descriptores que tengan un escenario en común y que comparten significado si se los trata de forma unificada.

De esta forma se analizó que los principales descriptores vienen modelados en forma de un número flotante, de un vector de flotantes o en casos más complejos, vienen modelados a través de ProcessingData. A partir de dicho análisis se propuso el diseño de tres mapas de la stl de c++, uno para cada tipo de descriptor posible que habíamos reconocido anteriormente.

Sin embargo, este diseño tenía rasgos correctos e interesantes tanto como otras soluciones propuestas por otros proyectos [29], por lo que se analizó las diferentes soluciones para crear una que las unifique a todas.

En la versión de CLAM 0.7 publicada el 19 de noviembre del 2004 se puede observar el diseño definitivo, junto a la documentación de Descriptors Computation and Storage in CLAM [6].

Una de las consecuencias más notables fue el desacoplamiento del uso de *DescriptorSet* de la rutina de *EparamS*, ya que todos los datos estaban centralizados alrededor de su uso.

De esta forma anteriormente cada Processing tenía el atributo dinámico *DescriptorName*, que servía como índice en el mapa interior de *DescritprSet*. En la actualidad dicho atributo se ha suprimido.

Otra razón por la cuál se prescindió del uso de DescriptorSet fue por no ser la versión definitiva de diseño del concepto anteriormente citado; y también por un posterior análisis que reveló la falsa necesidad de su uso dentro de la rutina de *EparamS*, puesto que se podía funcionar mucho más eficazmente prescindiendo del *DescriptorSet*.

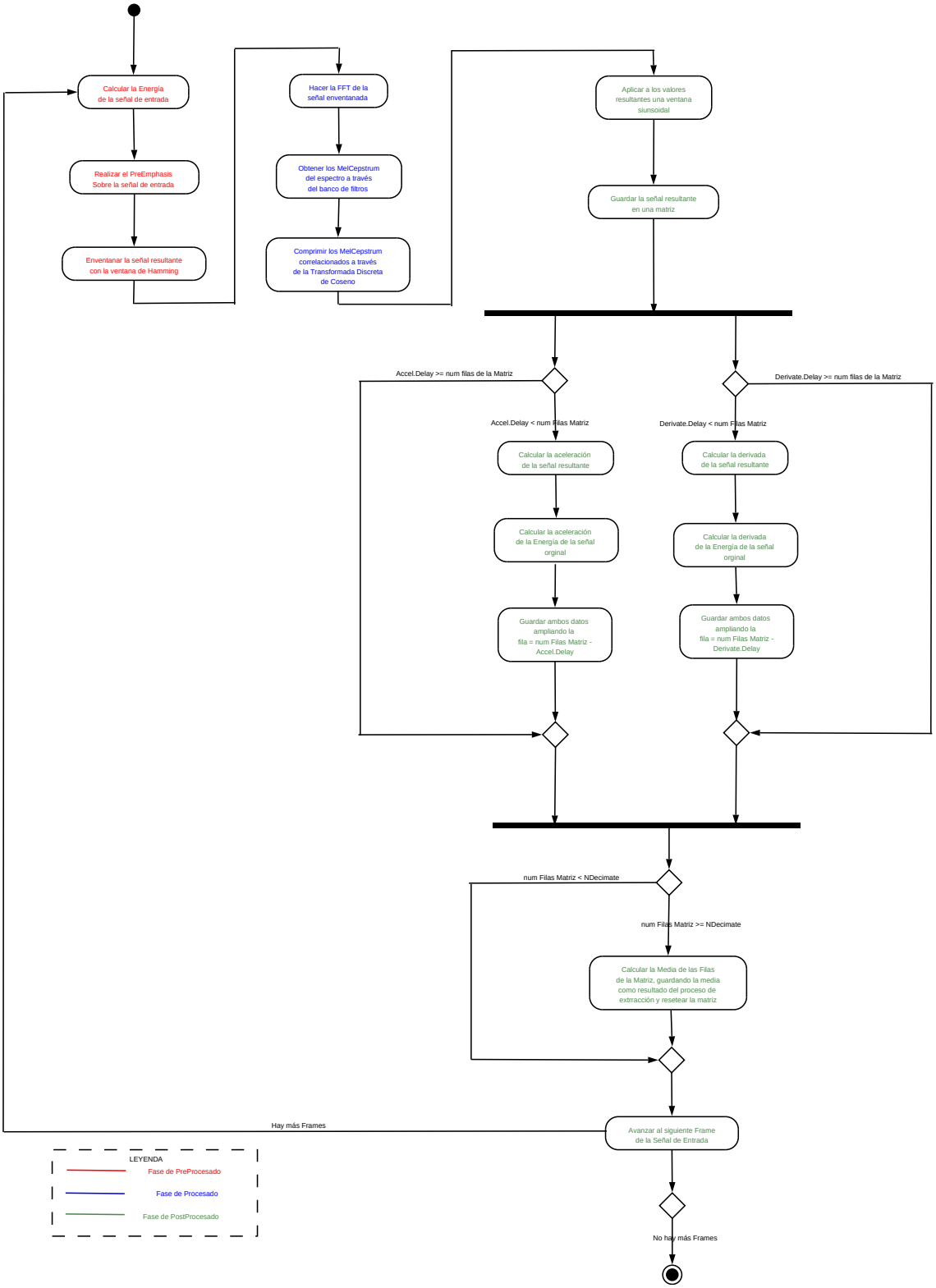

Figura 6.5: Diagrama de Actividad del algoritmo de EparamS

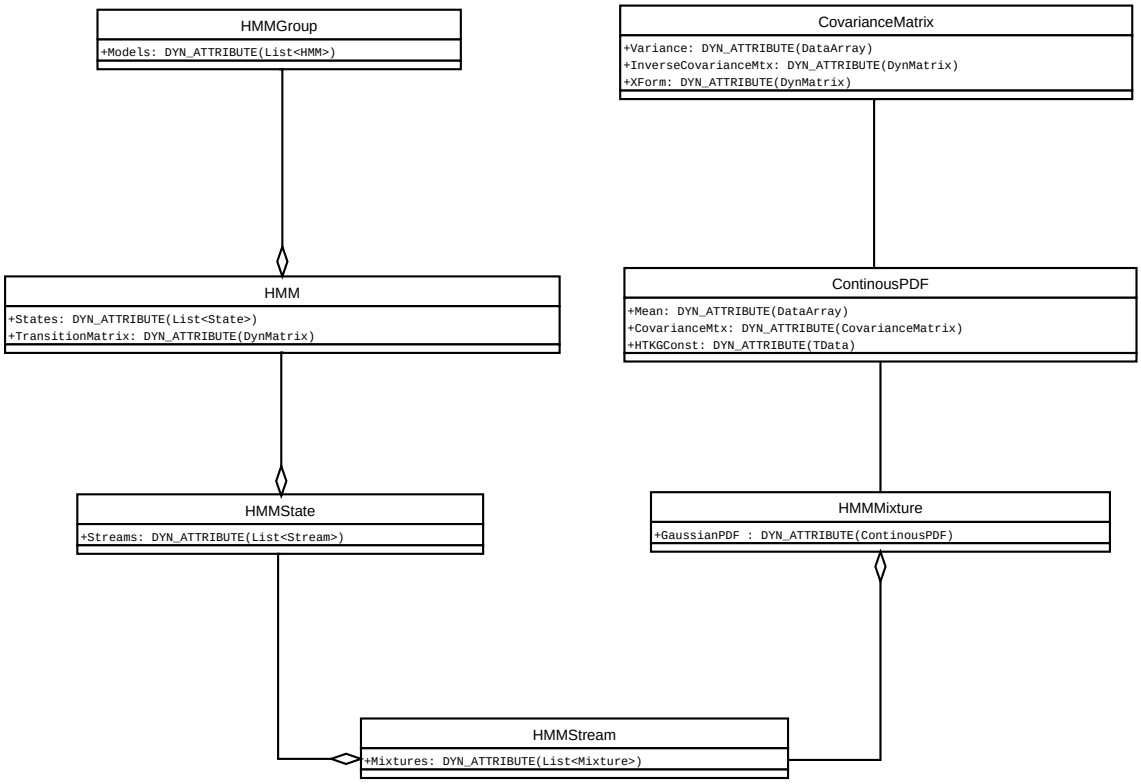

Figura 6.6: Diagrama de clases de HMM

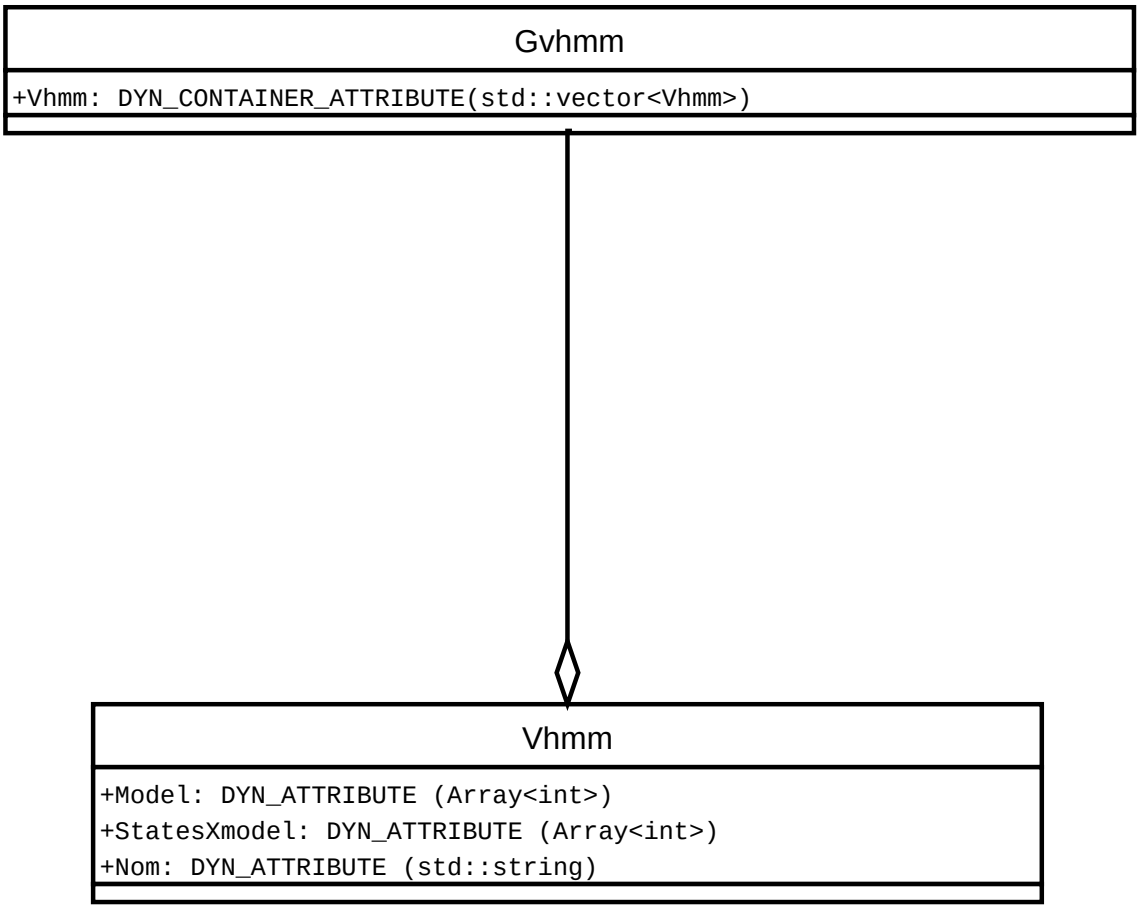

Figura 6.7: Diagrama de clases del Contenedor de modelos

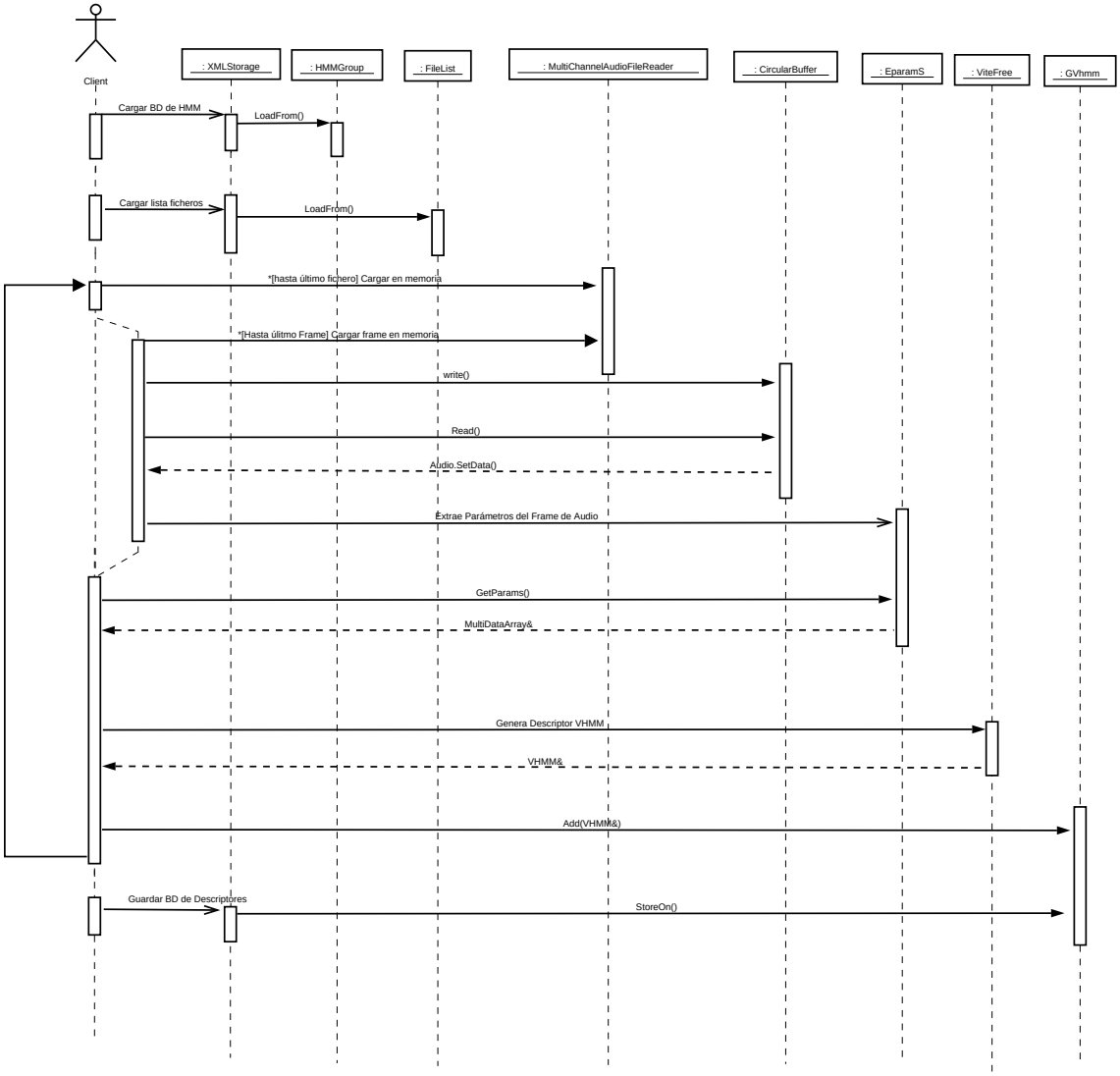

Figura 6.8: Diagrama de secuencia de Generación de Descriptores

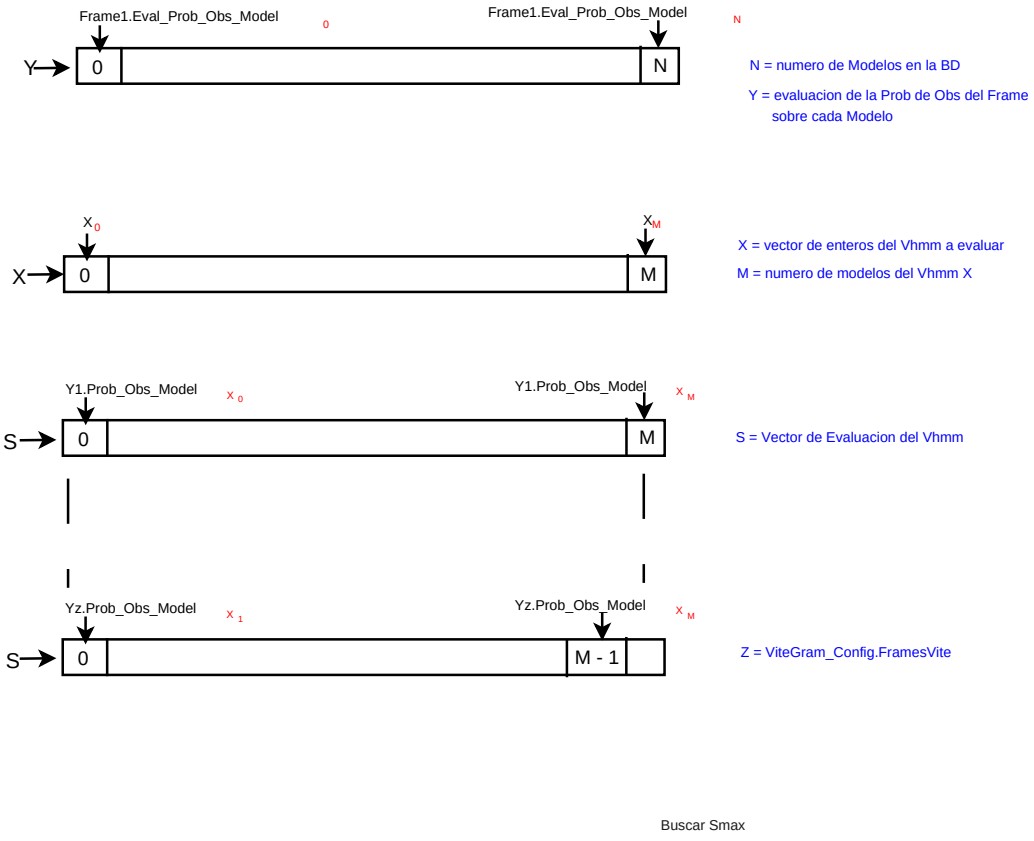

Figura 6.9: Algoritmo de Evaluación de un Vhmm

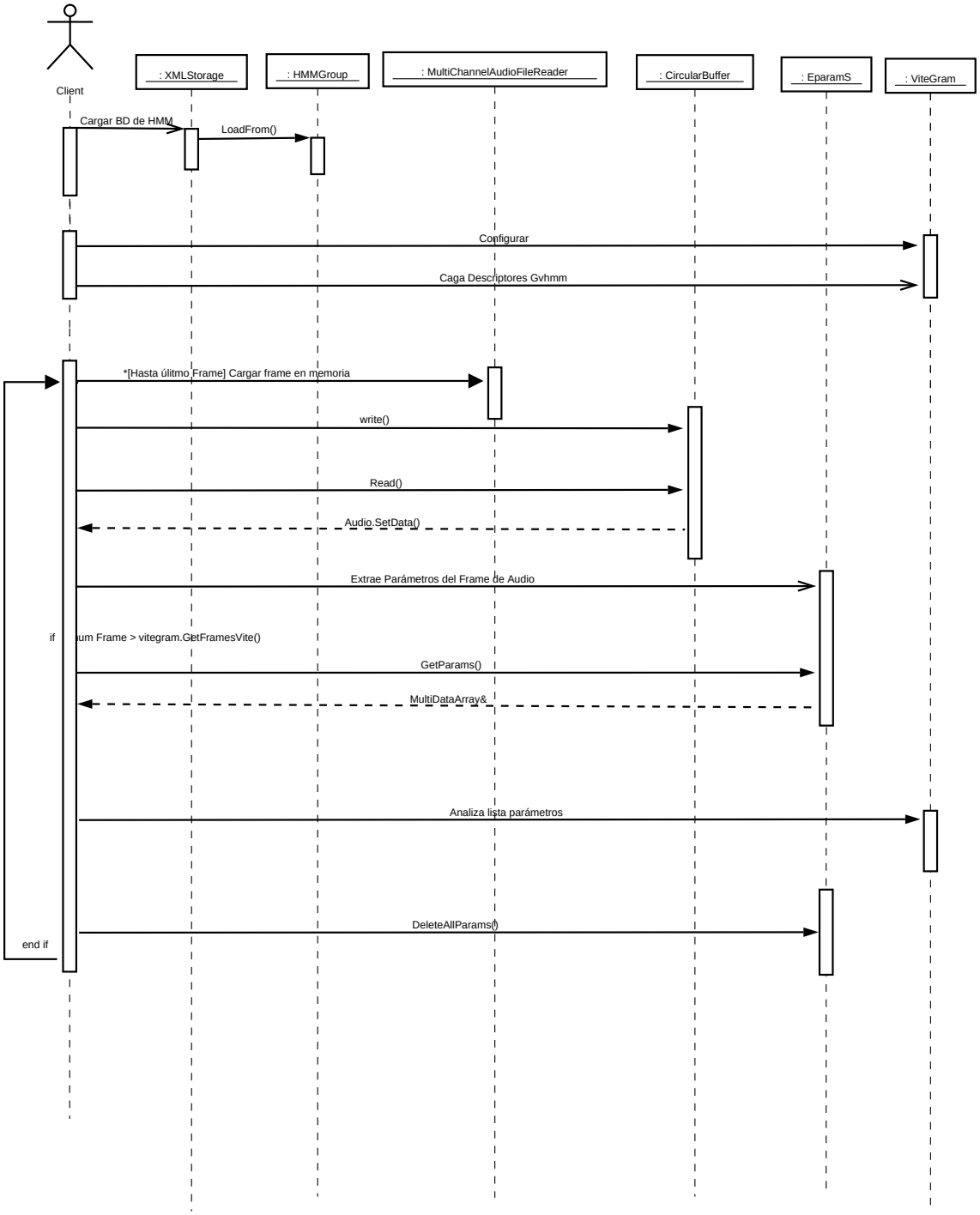

Figura 6.10: Diagrama de secuencia de Reconocimiento de música

Capítulo 7

Conclusiones

Llegado a este punto del proyecto uno tiene que practicar el ejercicio de la retrospectiva, observando cuáles fueron los puntos con los que partimos en la introducción, concluyendo qué rasgos hemos sido capaces de desarrollar, observando el capítulo de análisis de requisitos 3, y cuales son necesarios de seguir trabajando en un futuro.

En la introducción se plantean una serie de preguntas 1.1, algunas de las cuales sería interesante responder.

En primer lugar cabe decir que la adecuación de conceptos entre Process Network y los Modelos Ocultos de Markov, y más concretamente la aplicación de un sistema de reconocimiento de música, es bastante directo. Es decir, teniendo claro el modelo de Redes de Procesos de CLAM (DSPOOM 2.1) que sería el modo con el cuál debemos modelar una realidad, asimilando dichos conceptos resulta cómodo implementar en nuestro caso la realidad de los Modelos Ocultos de Markov. Se podría dividir la adecuación de conceptos en dos apartados:

- Todos las estructuras donde residen los datos, las funciones probabilísticas, los datos de los modelos, las estructuras, etc. Todos estos conceptos son datos modelables dentro del concepto de ProcessingData.
- Por otra parte, todos los algoritmos que manejen dichos datos, algoritmos de inferencia, de generación de datos, etc. Son posibles de adecuar al concepto de Processing, modelando todos los datos con los que interactúan dentro de la idea de ProcessingData.

De esta forma los dos elementos más importantes del modelo de CLAM como son los Processing y los ProcessingData, quedan ligados a los dos conceptos relevantes del modelo probabilístico que hemos ensayado en este proyecto: las estructuras de los Modelos Ocultos de Markov y los algoritmos de inferencia (Viterbi, etc.) respectivamente.

A su vez dicha separación de ambos conceptos ayuda a la comprensión del modelo,

aportando semántica por el simple hecho intuitivo de conocer a qué significantes corresponden los Processing y a cuáles corresponden los ProcessingData.

En segundo lugar resulta interesante hablar en términos de eficiencia del sistema.

Cuando uno observa CLAM, librería que concierne una gran cantidad de conceptos sobre el procesado de señal, cuesta imaginar que semenjante "buque" pueda ser eficaz espacial y temporalmente. Sin embargo, el diseño y la implementación de CLAM tienen dos rasgos esenciales:

- Sólo existe lo que se usa. Es decir, el peso del sistema que uno construya será el suyo propio ya que no usará toda la librería de CLAM, tan solo lo esencial, siendo justamente lo esencial el hecho que aporta un modelaje de redes de Procesos.
- Cuando uno imagina, por ejemplo, un objeto como el mencionado anteriormente HMMGroup 6.6, viajando a través de diferentes processing, resulta difícil creer que sea eficiente, puesto que mover semejantes cargas de datos lo último que parece es ser eficaz. No obstante existe una equivocación, puesto que ese pensamiento es el conceptual concerniente al modelo de CLAM, siendo el modelo físico muy distinto. Por ejemplo, cuando uno imagina que se desplaza todo un objeto, lo que realmente se desplaza no es más que el puntero a un ProcessingData.

Entender Dichos rasgos nos ayuda a entender que implementar un sistema con intenciones de casos de uso más generales, puede ralentizar la eficacia del sistema comparado con el uso de otros, como puede ser Amadeus, optimizados para situaciones más específicas. Lo importante es encontrar el equilibrio entre ambas ambiciones: generalización del sistema y mantenimiento de su eficacia.

7.1. Trabajo Futuro

Algunos rasgos que figuran, tanto en el análisis de requisitos 3 como en la introducción1.1, no pudieron ser llevados a cabo dentro del término de este proyecto. No obstante son características muy interesantes para tener en cuenta como futuras ampliaciones.

Una de dichas características que no se pudieron solventar hasta la fecha, corresponde a algunos rasgos mencionados en el análisis de generalización del sistema 3.5.

Algunos aspectos se pudieron solucionar, como fue el diseño de estructuras más flexibles en la representación de Modelos Ocultos de Markov, la aportación de flexibilidad en la extracción de parámetros en tiempo de ejecución, etc.

Sin embargo un aspecto muy interesante que no hubo tiempo de solventar fue la flexibilización del algoritmo de Viterbi. En este sentido quedó pendiente adaptar el algoritmo para que soporte Modelos Ocultos de Markov de más de un estado. Una posible solución que se planteó fue el uso del patrón de diseño *Strategy*, con el fin de preparar diferentes estrategias de cálculo según las distintas topologías de los modelos.

El mismo concepto de uso del patrón es aplicable a la implementación de *ViteGram*, algoritmo de Viterbi que debía soportar las restricciones de la aplicación de gramáticas. En este sentido también se podría implementar una estrategia de comportamiento para distintas gramáticas posibles.

Por otra parte, la componente funcional de Entrenamiento que ha sido solucionada con el uso del sistema LnkNet 4.11 6.7 3.1.1, sería interesante desarrollar para la misma una solución diseñada dentro del framework de CLAM. Una de las principales ventajas sería el ahorro de la interacción y dependencia a un sistema externo que tiene su modo propio de funcionamiento y al cual debemos adaptarnos. En este sentido, si se construye un módulo en CLAM para el entrenamiento de modelos, se hará conforme a las estructuras de datos diseñadas en este proyecto 6.6, evitando la necesidad de usar un parser que conecte y adapte ambos sistemas.

El análisis de más sistemas que realicen Reconocimiento de música mediante HMM sería muy interesante [10], así como el desarrollo de un estudio comparativo de diversos aspectos, eficiencia, flexibilidad, etc. entre todos ellos.

Resultaría también de gran utilidad, de cara al usuario, crear una interfaz gráfica (GUI) mediante la cual se puedan gestionar los Modelos Ocultos de Markov.

Por último sería interesante, al haber acabado ya el proceso de "refactoring" 4.3, estudiar qué patrones de diseño serían aplicables en los ámbitos desarrolados y por desarrollar.

7.2. Valoración Personal

Volviendo a mi situación en octubre del 2003, cuando inicié el proyecto, recuerdo haber visto que se me proporcionaba la posibilidad de investigar como ingeniero diferentes temáticas que eran de mi interés: Inteligencia Artificial, Procesado de Señal, Ingeniería del Software, entre otros.

Al iniciarlo, la magnitud del proyecto no me dejaba ver un horizonte claro hacia donde lo dirigía, sin embargo con el paso del tiempo fui adquiriendo nuevos conocimientos que me ayudaron, tanto desde el punto de vista organizativo como de conocimientos sobre distintos temas relacionados con el proyecto.

Otro aspecto que me interesó fue el hecho de poder conocer como funciona un equipo de trabajo que intenta desarrollar una librería, herramientas que usan, usuarios que participan, interacciones con otros proyectos; e intentar hacer las abstracciones necesarias para realizar el sistema del proyecto dentro del marco de CLAM.

Al terminar, veo que uno adquiere confianza para buscar y analizar conocimientos, tanto externos como internos, hecho clave para poder ejercer de Ingeniero en el mundo real.

Bibliografía

[1] Bison 1.35. URL: `http://www.gnu.org/software/bison/manual/html_mono/bison.html`.

[2] Samer A. Abdallah. *Towards music perception by redundancy reduction and unsupervised learning in probabilistic models*. PhD thesis, Department of Electronic Engineering, King's College London., 2002.

[3] Xavier Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with focus on Audio and Music*. PhD thesis, Departament de Tecnologia, Universitat Pompeu Fabra, 2004.

[4] Kent Beck. *Test-Driven Development, By Example*. Addison-Wesley, 2002.

[5] Pierre Brémaud. *Markov Chains: Gibbs Fields, Monte Carlo Simulation and Queues*. Springer, 1998.

[6] Descriptors Computation and Storage in CLAM. URL: `http://www.iua.upf.es/mtg/clam/devel/doc/descriptors/Descriptors.html`.

[7] Morris H. Degroot. *Probability and Statistics*. Addison-Wesley, 2 edition, 1988.

[8] CLAM documentation. URL: `http://www.iua.upf.es/mtg/clam/documentation.html`.

[9] CppUnit Documentation. URL: `http://cppunit.sourceforge.net/doc/lastest/index.html`.

[10] The CMU Sphinx Group Open Source Speech Recognition Engines. URL: `http://cmusphinx.sourceforge.net/html/compare.php`.

[11] A fast scanner generator Flex, version 2.5. URL: `http://www.gnu.org/software/flex/manual/html_mono/flex.html`.

[12] Helm R. Johnson R. Gamma, E. and Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 3 edition, 1995.

[13] Clare D. McGillem George R. Cooper. *Probabilistic Methods of Signal And System Analysis*. Oxford University Press, 3 edition, 1999.

[14] Perry Cook George Tzanetakis. Marsyas: A framework for audio analysis. pages 1–13, 2000.

[15] Standard Template Library Programmer's Guide. URL: `http://www.sgi.com/tech/stl/`.

[16] Robert Esser Jörn W. Janneck. High-order petri net modelling techniques and applications. *ICATPN'2002*, 12, 2002.

[17] L. Lamport. *LaTeX, A document preparation system, second ed.* Addison Wesley, 1994.

[18] Edward A. Lee and Alberto Sangiovanni-Vincentelli. Comparing models of computation. november 1996.

[19] John Brant William Opdyke Don Roberts Martin Fowler, Kent Beck. *Refactoring: Improving the Design of Existing Code.* Pearson Education, 2002.

[20] Terrence J. Sejnowski Michael I. Jordan. *Graphical Models, Foundations of Neural Computation.* Jordan and Sejnowski, 2001.

[21] Judea Pearl Home Page. URL: `http://bayes.cs.ucla.edu/jp_home.html`.

[22] UML Resource Page. URL: `http://www.uml.org/`.

[23] Thomas M. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, EECS Department,University of California, 1995.

[24] Lawrence R. Rabiner. A tutorial on hidden markov models and seceted applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[25] M. Ramírez. Clam visualization module: Disseny dún mòdul de visualització oo per a un entorn de processament d'àudio independent de plataforma. Master's thesis, UPF, 2002.

[26] X. Rubio. Un sistema de control de fluxe per síntesi i processament d'àudio en temps real. Master's thesis, UPF, 2003.

[27] L. R. Rabiner S. E. Levinson and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition. *System Technical Journal*, 62(4), April 1988.

[28] LNKnet Pattern Classification Software. URL: `http://www.ll.mit.edu/IST/lnknet/index.html`.

[29] Existing solutions for the descriptors problem. URL: `http://www.iua.upf.es/mtg/clam/devel/doc/descriptors/ExistingImplementations.html`.

[30] Ian Sommerville. *Ingeniería de Software*, chapter Speech, Song and Emotions, pages 146–156. Pearson Educación, 6 edition, 2002.

[31] Rob Strom. A comparison of the object-oriented and process paradigms. *SIGPLAN Noticies*, 21:88–97, october 1986.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 2004.

[33] PeterÑorvig Stuart Russell. *Artificial Intelligence: A Modern Approach*, chapter 13-21, pages 462–763. Pearson Education, 2 edition, 2003.

[34] Music Group Technology. URL: `http://www.iua.upf.es/mtg/clam/documentation.html`.

[35] The Hidden Markov Model Toolkit. URL: `http://htk.eng.cam.ac.uk/`.

[36] Basu Vaidyanathan. A study on process network. october 1999.

[37] Andrew Webb. *Statistical Pattern Recognition*. Wiley, 2 edition, 2002.

[38] A validating XML parser Xerces-C++. URL: `http://xml.apache.org/xerces-c/`.

[39] Michael I. Jordan Zoubin Ghahramani. Factorial hidden markov models. *Machine Learning*, 29:245–273, 1997.

Índice alfabético