AP-GEN: FRAMEWORK GENERADOR DE ESQUELETOS DE PLUGINS DE AUDIO A PARTIR DE ABSTRACCIONES Y TEMPLATES

HERNÁN ORDIALES [1]

[1] Consultor Independiente, Estudiante de Ingeniería en la Universidad de Buenos Aires, Argentina.
h@ordia.com.ar

Resumen – *AP-Gen agiliza y facilita el desarrollo de plugins generando código fuente de base, tanto para diferentes estándares como para diferentes sistemas operativos, logrando de esta forma que el desarrollador pueda concentrarse en su objetivo, el procesamiento de audio. Para lograrlo, parte una serie de definiciones normalizadas y hace uso de un motor de templates. Estas definiciones pueden ser abstracciones generales del plugin, como ser cantidad y tipo de entradas y salidas de datos o controles, o pueden ser más específicas, como cuestiones relativas al sistema de construcción, metadata y detalles de cada arquitectura. Los templates, por otro lado, son los elementos que conservan las características comunes de cada estándar. La ventaja de esta forma de trabajo se haya en que una vez establecidas las propiedades de alto nivel, el paso siguiente ya puede ser el de implementar la función de procesamiento. Dejando de lado, entre otras cosas, el trabajo mecánico de escribir código de librerías, detalles del lenguaje o estándar de turno, y cuestiones de compilación de cada arquitectura. Entre otras cosas, el software provee una interfaz común para todos los generadores de plugins, lo que posibilita el agregado modular de nuevos estándares. Esta publicado como Software Libre y en la actualidad cuenta con soporte para plugins VST, LADSPA y CLAM.*

Abstract – *AP-Gen speeds up and eases the plugin development through base source code generation, both for different standards and operating systems, thus achieving that the developer can focus on his goal, the digital audio processing. To achieve this, starts from normalized definitions and uses a template engine. These definitions could be general abstractions of the plugin, like amount and type of inputs and outputs of data or controls, or more specific ones, like the kind of the build system, metadata and even details of each architecture. On the other hand, the templates are the elements that maintain the common features of each standard. The advantage of this way of working is that, once high level aspects are established, next step can be to develop the signal processing function. Leaving aside, among other things, all the repetitive work of writing library code, programming language or current standard details, and each architecture compilation issues. Among other things, the software provides a common interface for all plugins generators, so support for new standards can be added in a modular form. Is released as Free Software and at present has support to VST, LADSPA and CLAM plugins.*

1. INTRODUCCIÓN

El trabajo con plugins de audio, además de permitir extender la funcionalidad de programas preexistentes, permite el uso y desarrollo de diferentes funciones de procesamiento en forma modular. Característica que otorga visibles ventajas, entre otras, el hecho de poder centrarse en objetivos más pequeños y pasar de lo simple a lo complejo concatenando módulos. Por otra parte, es común enfrentarse con problemas o incomodidades durante su ciclo de desarrollo. Para los principiantes, enfrentarse con la escasa o mala documentación y las dificultades de compilación asociadas con cada arquitectura. Son cuestiones que desmotivan en muchos casos hasta el abandono. Y para los que dominan estas cuestiones, muchas de las tareas repetitivas y la cantidad de detalles y formas que requiere cada arquitectura, hacen que se pierda constantemente mucho tiempo valioso de desarrollo. Además, el camino directo, en muchos casos lleva a un acoplamiento no conveniente entre la lógica general y la arquitectura. Estas situaciones conducen al hecho de que a veces sea útil trabajar con frameworks, o entornos de desarrollo, que organicen y/o alivianen la mencionada tarea.

Este trabajo pretende contribuir a la solución de tal problemática desde un enfoque diferente. Partiendo del objetivo de evitar las malas prácticas de desarrollo de software detectadas, propone en contraposición, un nuevo esquema de trabajo general basado en abstracciones del plugin, y generadores automáticos de código fuente. El cual se encarga, entre otras cosas, de eliminar todo trabajo habitualmente mecánico y repetitivo pero necesario durante el ciclo de desarrollo. En particular, ataca la problemática del desarrollo masivo y abarcativo, en el sentido de una cantidad grande, y soportando diferentes arquitecturas de plugins de audio, tanto de tiempo real, como de procesamiento fuera de línea.

Se comienza con una descripción general del contexto del problema y sus conceptos asociados, incluyendo un análisis pormenorizado de los enfoques actualmente existentes. Para luego pasar a una descripción detallada de los conceptos y criterios generados que dan base a la solución propuesta. Así como los fundamentos del diseño y la descripción de las cuestiones experimentales y de implementación que atravesó. En el final del mismo, se muestran ejemplos de uso y analizan resultados, enumerando las ventajas, para llegar a la conclusión, donde es comparado y se analiza dentro de que contextos de trabajo es conveniente su utilización.

2. CONTEXTO DEL PROBLEMA

Como ya se adelantó, el desarrollo de plugins tiene sus aristas débiles. Una de las razones por la cual implementar algoritmos de procesamiento de señales de audio comúnmente resulta una labor compleja, reside en que es una tarea que necesita profundidad en dos tipos de conocimientos. El teórico, sobre procesamiento de señales en tiempo discreto, y el de desarrollo de software. Cuestión tal vez central, ya que generalmente son áreas de especialización separadas. Este trabajo apunta a facilitar la tarea del segundo ítem, en particular en lo que tiene que ver con cuestiones que surgen al programar extensiones (los plugins). Más allá de las cuestiones inherentes al proceso de desarrollo de software, son notorias las surgidas a partir de la diversidad existente de estándares y sistemas operativos.

Estas fueron detectadas en distintos ámbitos de desarrollo, grupos profesionales de la especialidad, proyectos de software y estudiantes universitarios. Para más datos, tales problemáticas, se manifestaron con más frecuencia en el comienzo de los procesos, ya sea en el camino de dominar cierto estándar, o en el de querer pasar del prototipo en otra plataforma a la implementación final en forma de plugin. Dificultad que en muchos casos se transformó en abandono.

2.1. Plugin de audio

La arquitectura de trabajo con plugins reside en un modelo con dos tipos de actores, el host y los plugins. Donde el host o aplicación principal es capaz de comunicarse con uno o más elementos de software más pequeños (los plugins) a través de una interfaz definida (API).

Una práctica común es definir conjuntos de funciones comunes en una librería de carga dinámica y regular en un estándar como es que deben ser utilizadas. Esto permite que un mismo plugin, si respeta un determinado estándar, pueda funcionar en toda aplicación que implemente lo necesario para hacer las veces de host. En pocas palabras, definen como se debe construir un host y como un plugin para que estos sean compatibles entre si. En audio, uno de los más populares si no el más, es el VST. Para una comparativa entre las arquitecturas de los estándares más populares, consultar “Real-time audio plugin architectures” [1].

Es importante destacar que el plugin, por lo general y por cuestiones de diseño, tiene una funcionalidad específica. Es decir, la idea es que ataque un determinado problema o proporcione una funcionalidad bien definida. La principal ventaja de utilizar este tipo de arquitectura se encuentra en que de esta forma se permite a terceros (individuos o empresas) extender una aplicación. Ya sea para simplificar el programa principal, o como una forma de agregar capacidades no previstas originalmente. Por ejemplo, en un reproductor de audio, una extensión o plugin se puede encargar de soportar un nuevo codec.

Otra ventaja, especialmente explotada en procesamiento de audio, reside en concatenar grupos de plugins como módulos formando redes o cadenas de procesamiento. Logrando asi funcionalidades más complejas a partir de unidades más simples y dedicadas (que hagan solo una cosa y bien).

2.2. Problemática del desarrollo

Normalmente una secuencia de desarrollo típica se puede descomponer en los siguientes pasos:

1. (opcional) Prototipar el algoritmo en otra plataforma. Por ejemplo Matlab.
2. Elegir un estándar de plugins según las necesidades o requerimientos.
3. Elegir herramientas de trabajo. Por ejemplo: librerías, compilador, IDE.
4. Crear los archivos de código fuente siguiendo las convenciones y requerimientos del estándar elegido.
   - -Construir clases o estructuras. Escribir defines, constantes, etc.
   - -Respetar convenciones de nombres, por ejemplo para variables o métodos.
   - -Agregar cuestiones particulares del plugin en cuestión a la estructura general, como ser cantidad y nombre de entradas y salidas de flujos de señal de audio y controles.
5. Crear y configurar los archivos del sistema de construcción de acuerdo al sistema operativo elegido como destino (excepto en interpretados).
6. Escribir la función de procesamiento del plugin propiamente dicha y la interacción con los controles.

Los pasos de los puntos 4 y 5 suelen traer conflictos si no se conoce bien el estándar de antemano o no se desarrolla para el mismo frecuentemente. Tampoco ayudan la generalmente escasa o mala documentación sobre el tema. Por otra parte, y por cuestiones de performance, las librerías para desarrollo de plugins de audio suelen estar en lenguaje C o C++. En especial, esta tendencia es todavía mayor cuando los plugins están pensados para procesamiento en tiempo real. Estos lenguajes generalmente involucran detalles muchas veces

complejos, cuestiones en general tediosas (y que no hacen al procesamiento) para todo aquel que no se dedique a programar con los mismos en el día a día. Al menos, es comúnmente aceptado que es más fácil equivocarse con estos que con otros, pensados con otros objetivos. Asunto que se traduce directamente en el tiempo que hay que dedicarle al punto 4. Básicamente, por la gran cantidad de detalles a los que se tiene que estar atento constantemente. También ocurre que debido a su versatilidad, generalmente se permiten varias formas de hacer una misma cosa, hecho que, si no se proponen convenciones de código, termina desembocando en bugs y dificultades en el mantenimiento.

Los plugins más utilizados son del tipo binario (aunque hay casos de interpretados) y por lo tanto necesitan ser compilados, para lo cual precisan de un sistema de construcción (build system) que transforme los fuentes en binarios (en este contexto, librerías de carga dinámica). Cuestión que también puede traer problemas en la práctica, por la cantidad de opciones y necesidades diferentes de cada compilador. Exceptuando los casos donde el IDE elegido es el que se encarga de la mayor parte, la mayoría (de los build systems) son poco intuitivos y confusos para los acostumbrados o principiantes en el tema, sobre todo los basados en Makefiles. El enlazado (link) de librerías en los lenguajes comentados suele ser una tarea no trivial para los que no suelen trabajar con este tipo de arquitecturas.

La consecuencia más importante de esto se manifiesta en que el proceso de debug (eliminación de errores) se encuentra en un porcentaje alto antes (y por fuera) de lo correspondiente a la función de procesamiento, es decir, lo que realmente interesa del plugin.

También ocurre que la estructura general de código fuente que se necesita para generar plugins compatibles con los estándares, suele ser bastante extensa, por lo menos a comparación con lo que puede llegar a ser una función de procesamiento sencilla. Pero el problema en este caso no es la extensión, siendo que esta dentro de los límites de lo manejable, sino que si se compara el código de dos plugins, se cae en la cuenta rápidamente de que, sacando la parte del procesamiento y manejo de controles, la diferencia entre un código y el otro suele ser, por ejemplo, el parámetro de nombre que usa una función o el seteo de la cantidad de entradas. Es decir, hay mucho código parecido, que no es exactamente igual, salvo en pequeños detalles. En otras palabras, hay mucha información redundante. Lo cual suele fomentar en la práctica el uso abundante del 'copiar y pegar', o el 'buscar y reemplazar', práctica habitual en el desarrollo, pero que es fuente común de bugs, y por lo que se verá más adelante, evitable.

Otra problemática, no profundizada en este trabajo, tiene que ver con el hecho de que el tratamiento de la señal audio digital, es naturalmente de bajo nivel de abstracción (buffers que se procesan). Esta es una de las razones por la que se utiliza tanto el lenguaje C para estas tareas, ya que permite un buen control a bajo nivel. Pero esto no quita que sea un problema que se pueda encarar de forma más abstracta, por ejemplo mediante un modelo de programación orientado a objetos. Es frecuente que las librerías C++ de los estándares suelan ser wrappers (envoltorios para conjuntos de funciones) que provean acceso orientado a objetos a APIs estructuradas en C. Esto tiene un punto de contacto con el presente trabajo en el hecho de que hay veces que según lo elegido en el punto 3, influye en como encarar el punto 4, por ejemplo orientar el diseño del plugin usando clases (programación orientada a objetos) o estructuras cuando corresponda. Son tareas, que como se detallara más adelante, se pueden automatizar a partir de simples definiciones de alto nivel.

Esta serie de problemáticas enumeradas, llevaron a pensar que tener presente tantos detalles de sintaxis (por el lenguaje o requerimientos del estándar) va totalmente a contramano de poder concentrarse en el algoritmo de procesamiento, lo que en definitiva más importa en un plugin de esta índole. No por nada mucha gente elige prototipar primero sus algoritmos de audio en plataformas como Matlab (o sus compatibles libres), incluso cuando esta herramienta no esta orientada fuertemente al procesamiento de audio en particular. Alternativas si centradas en el audio, y en especial en el tiempo real, pero con enfoques al problema diferentes al aquí presentado, son descriptas en la siguiente sección. Más adelante se pasa a presentar la solución propuesta por este trabajo e implementada como el software denominado AP-Gen.

2.3. Entornos de desarrollo para crear software de procesamiento de audio

Entre algunas de las herramientas que ayudan en el proceso de desarrollo de plugins audio, se encuentran CLAM y FAUST.

A pesar de que su nombre proviene de 'C++ Library for Audio and Music', CLAM es más que una librería, es un framework de desarrollo. Propone trabajar dentro de un metamodelo conceptual [2] y provee diferentes herramientas que facilitan la investigación y desarrollo, de software multiplataforma relativo al audio y la música [3]. Así como elementos para el análisis, procesamiento y síntesis de señales de audio. Una de sus características más relevantes reside en la posibilidad de construir prototipos de aplicaciones o plugins visualmente [4], interconectando módulos, para luego, en el caso de estos últimos, exportarlos a estándares conocidos, y en la caso de los primeros, permitiendo crear interfaces gráficas de forma amigable. También puede funcionar como librería C++, fuertemente orientada a objetos, al mismo tiempo que permite el manejo típico de bajo nivel. Permite interactuar con plugins de los estándares más

conocidos y crear los suyos propios como módulos (processings) de carga dinámica.

FAUST es un lenguaje de alto nivel para procesamiento de señales en tiempo real. Combina programación funcional con diagramas en bloques [5]. Esta pensado para desarrolladores que quieran lograr plugins de audio eficientes en C/C++ o aplicaciones de audio independientes. Programar con FAUST es parecido de alguna forma a trabajar con circuitos electrónicos y señales. Un programa FAUST es una lista de definiciones que definen un diagrama de bloques de un procesador de señal. Exporta código para diferentes arquitecturas de plugins o crea mini aplicaciones con front-end gráfico.

3. DESCRIPCIÓN Y FUNDAMENTOS DE LA SOLUCIÓN PROPUESTA.

Dado los problemas ya descriptos en la sección anterior, se buscó la forma de disminuir el tiempo que pasaba entre que surgía la idea de programar un plugin, y el momento preciso en el que se empezaba a desarrollar el algoritmo de procesamiento. Para esto, se tuvo como premisa principal el intentar evitar todo tipo de trabajo mecánico pasible de ser automatizado. Para obtener, en última instancia, un sistema que permita el desarrollo rápido de plugins de audio, y que además sea abarcativo, en el sentido de que contemple soporte para múltiples sistemas operativos y estándares de plugins.

Para lograr esto, se plantearon una serie de objetivos a nivel desarrollo de software, y con la idea siempre presente de pensar al plugin en forma abstracta, se logró una propuesta. El desarrollo fue incremental, comenzó con el ProcessingCodeGenerator (ver sección 5), y poco a poco se le fue dando forma a un modelo de trabajo con un enfoque diferente a los ya existentes.

El mismo surgió en gran parte como una recopilación de la experiencia del autor escribiendo plugins para variados estándares, y compilando para diferentes sistemas operativos. Tanto desde cero, encargándose de cada detalle, como trabajando con diversos entornos de desarrollo, sobre todo con el framework CLAM, ya citado en 2.3. El cual, también influenció este trabajo indirectamente, a partir de la gran cantidad de publicaciones asociadas al mismo [2][3][4].

3.1. Objetivos

Se puso el foco en atacar los siguientes problemas:

- Desacoplar las características de alto nivel de los plugins, de los detalles de implementación de cada estándar, y configuraciones particulares de los sistemas de construcción de los binarios para cada sistema operativo.
- Evitar todo el trabajo mecánico posible.
  - Evitar prácticas del tipo basarse en otro código y abusar del 'buscar y reemplazar' o el 'copiar y pegar'.
- Poder generar código base o esqueletos para diferentes estándares.
- Crear y proponer estructuras comunes y convenciones en base a prácticas recomendadas.
- Soportar diferentes sistemas operativos, de forma tal que el código compile desde el inicio.

3.2. Abstracción del plugin

La primera abstracción que se puede hacer de un plugin de audio, consiste en pensarlo como un objeto con una función de procesamiento asociada. Una cantidad de entradas y salidas de datos y controles, más una serie de propiedades o configuración interna [1][2]. Con la función de procesamiento aplicándose a los datos que se colocan a la entrada y depositando el resultado en la salida. Ya sea tratando la señal de entrada como un todo, en el caso del procesamiento fuera de línea, o utilizando buffers para trabajar la misma en forma segmentada, como es el caso del tiempo real. Mientras que los controles de entrada, actúan como parámetros del procesamiento, y los de salida como generación de eventos discretos, que pueden ser por ejemplo, entrada de control de otros plugins. En la Figura 1, se puede observar un esquema de esta idea.

- Flujos de señal: entrada/salida.
- Controles: entrada/salida.
- Propiedades.

En el caso de un amplificador mono sencillo, el mismo podría ser descripto como un plugin con una entrada y una salida del tipo audio, más un control de entrada continuo o discreto destinado a modificar el parámetro de ganancia. Un ejemplo de control de salida podría ser, por ejemplo, la generación de un evento MIDI o la emisión de un pulso cuando se detectó que la señal de entrada superó un cierto umbral.

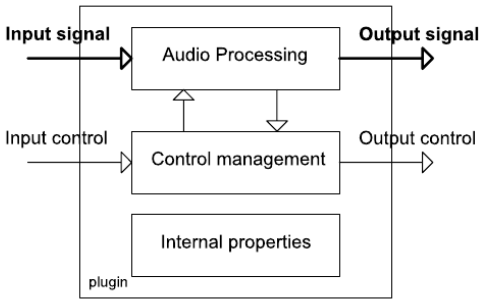

Figura 1: Modelo común de un plugin de audio [1].

El plugin también se puede desglosar en sus funcionalidades básicas, luego conectadas con las interfaces de las APIs de los estándares [2].

- Procesar(Entrada): Salida
- Configurar(Parámetros)
- Empezar()
- Detenerse()

No es un dato menor mencionar que al hablar de datos o controles, se habla adrede en forma genérica, se modela como datos (o controles) de entrada y salida, por que los mismos pueden llegar a ser de distinto tipo, aún limitándose al procesamiento de audio. Por ejemplo, en lugar de recibir los datos en función del tiempo, se podría desear que la entrada del plugin directamente sea de datos en el dominio de la frecuencia, y dejar la transformación tiempo-frecuencia para otro módulo especializado.

3.2.1. Entropía

Como se comentó cuando se habló de los problemas comunes que surgen durante el desarrollo de plugins, el código fuente de plugins de un mismo estándar, comparten entre si una estructura general en común. Es decir, entre el código de un plugin y otro, y en términos de compresión de la información, hay mucha información redundante. A partir de esto se puede concluir que se podría reconstruir completamente un plugin a partir de las abstracciones mencionadas, si después se tiene la forma de expandirlo, por ejemplo con un software. Generando el código fuente completo necesario para compilar un plugin compatible con un determinado estándar. En otras palabras, las mencionadas definiciones generales de alto nivel, se podrían considerar como un subconjunto de información pura del plugin (entropía del plugin). Contenido que si se separa de la implementación o estándar elegido, abre la puerta entre otras cosas, a la posibilidad de generar código fuente base para cualquier estándar, a partir de la misma definición.

Como una primera definición de un plugin en términos de información pura, podemos hablar de:

- Abstracciones generales de estructura.
  - Tipo de entradas y salidas de datos y controles.
- Algoritmos de procesamiento y control.
- Propiedades internas.
- Nombre del estándar y sistema operativo destino.

3.3. Propuesta

Para poder empezar a codificar rápidamente la función de procesamiento de un plugin, es necesario tener armada la estructura general de código que requiere el estándar, típicamente cuestiones relacionadas por ejemplo con el nombre, la cantidad de entradas y salidas, etc. Y para evitar escribir todo eso, hay que buscar la forma de automatizar esas tareas.

La solución propuesta consiste en generar código base, o esqueletos de plugins, a partir de definiciones en archivos XML y un sistema de templates. Es decir, a partir de un archivo que almacene parte de la entropía descripta en 3.2.1 (el concepto abstracto del plugin, y una configuración compuesta por el tipo de estándar de salida deseado y el sistema operativo destinto), obtener el código fuente de base, listo para compilar, de forma tal que el paso inmediato siguiente pueda ser el de desarrollar el algoritmo de procesamiento de la señal de entrada.

Como forma de poner en práctica estas ideas, se desarrollo un software que las implementa, y se lo denominó AP-Gen, por “AudioPlugins Generator”.

4. DISEÑO DE AP-GEN

Se planteó la idea de un mini-framework de desarrollo, en el sentido de ser un software cuya finalidad es desarrollar otro software, en este caso los plugins. Proponiendo una manera de trabajo dentro de una estructura conceptual y proveyendo herramientas. Como ser el concepto de generar esqueletos de código a partir de la entropía del plugin almacenada de forma estandarizada. Código que se genera al pasar los archivos XML con las definiciones del plugin por una aplicación, denominada motor de templates. La cual combina el mencionado archivo XML con un template o plantilla para generar información de salida, que consiste en todos los archivos necesarios para construir un plugin determinado.

Este modelo de trabajo estuvo fuertemente influenciado por prácticas comunes en otros frameworks pensados para otras disciplinas (por ejemplo desarrollo web con Django o Ruby on Rails). Heredadas del principio de ingeniería de software DRY (“Don't repeat yourself” o “No te repitas”) de utilizar generadores de código vía templates, a partir de información estandarizada, a fines de evitar o reducir información duplicada [6].

En contraposición al esquema mencionado en 2.2, la secuencia de pasos para desarrollar un plugin de audio que AP-Gen propone consiste en:

1. Crear un archivo XML con un editor de texto plano o por medio de una interfaz gráfica.
   - Definir el plugin audio en forma abstracta.
   - Definir el tipo de plugin de salida deseado. Elegir sistema de construcción y sistema operativo.
2. Generar el esqueleto del plugin pasando el XML por el motor de templates (aplicación de AP-Gen).
3. Implementar la entropía faltante del plugin.
   - Función de procesamiento.
   - Interacción con controles.
   - Ajustar detalles menores según el estándar elegido.

De esta forma, se elimina prácticamente toda tarea mecánica. Generando todo aspecto posible que no contenga información en si mismo, es decir, los aspectos compartidos por todos los plugins, y dejando sus diferencias para el archivo de definición. Reemplazando de esta forma todo el trabajo enumerado anteriormente en los pasos 4 y 5 de 2.2, por la construcción de tal archivo. Es importante aclarar, que no es una cuestión de comparar la cantidad de pasos, sino la simplificación y reorganización de los mismos. En este último caso, y a diferencia del primero, la tarea de debug, solo se encuentra presente en el último paso. Cuando se llega al punto 3, el plugin ya compila y "funciona", aunque no todavía no haga nada, ya es posible cargarlo en cualquier host. Esta última característica, también promueve el desarrollo iterativo e incremental [7] del plugin, algo que bien encarado, por ejemplo como proponen muchas metodologías de desarrollo ágiles [8], tiene numerosas ventajas.

El hecho de trabajar con templates, permite, además de la generación de código, proponer como estructuras comunes esqueletos de código con prácticas y elecciones recomendadas. Como ser decisiones estructurales de base, por ejemplo relativas al uso de clases o estructuras, así como convenciones de código y otras posibles decisiones. Cuestiones que luego se traducen en ventajas, tanto por razones de una mejor eficiencia del código propuesto, portabilidad, y en la simplificación del mantenimiento del mismo. De esta forma, se limitan un poco las malas prácticas a nivel código fuente que suelen ser comunes en desarrolladores principiantes.

Se decidió que el soporte para diferentes arquitecturas de plugins debía ser modular, para permitir así, un desarrollo incremental de la aplicación. Es decir, encarándolo de esta forma se podía comenzar soportando primero solo un estándar, y luego, a medida de que iba siendo necesario o había tiempo para el desarrollo, agregar otros.

Para esto se definió una interfaz común (documentada) con las funcionalidades básicas que todo módulo de salida o backend debía cumplir. De forma tal, que implementándola, cualquier programador pueda construir un nuevo módulo sin tener que conocer los detalles internos de AP-Gen. Se habla de backend, para hablar de la abstracción de la parte del sistema que se encarga de procesar la información de entrada, con una configuración en particular, en este caso, el tipo de estándar de salida.

4.1. Formato XML para definir un plugin

Como medio para almacenar las características de alto nivel del plugin, y algunos parámetros de configuración, se definió una estructura en lenguaje XML. El mismo, es un lenguaje estándar para el intercambio de información, que cuenta entre sus ventajas, la sencillez, el hecho de ser legible por humanos (es texto plano) y de ser extensible. Para más datos sobre las ventajas de trabajar con este tipo de archivos de texto plano, consultar [6].

Expandiendo lo visto en 3.2.1, se puede decir que la entropía o información esencial de un plugin de audio, queda definida por:

- Abstracciones generales de estructura.
  - Tipo de entradas y salidas de datos y controles.
- Algoritmos de procesamiento y control.
  - Parámetros internos.
- Metadata.
- Configuraciones de la arquitectura destino.
  - Estándar.
  - Sistema operativo.

El resto, es información redundante y automatizable a partir de la entropía.

Las abstracciones del plugin, consisten en lo mencionado en la sección 3.2. Es decir, definir tipo y nombre de cada una de las entradas y salidas de datos o controles. Para algunos casos, también es posible definir otros valores, por ejemplo en los controles que manejan valores reales, es posible definir valores mínimos, máximos y por defecto. Los algoritmos, y sus parámetros que no dependan de los controles, son parte de la entropía que le queda por generar al programador. La metadata consiste en datos que describen a otros datos. En este caso, el nombre y categoría del plugin, el nombre de los autores, la licencia, etc. Incluso el identificador numérico y único, que requieren algunos estándares. La configuración, contiene el estándar y template elegido para generar la salida (el backend), sistema operativo destino y tipo de sistema de construcción, entre otras cosas. Como el seteo de detalles dependientes de cada estándar.

El archivo XML que contiene todas estas definiciones se puede considerar como una regla de construcción de un plugin, ya que provee todos los elementos necesarios para la generación de la estructura base del mismo, el esqueleto y su sistema de construcción. El formato que debe respetar la regla XML esta definido en un archivo del tipo DTD, que define la estructura para un tipo de documento XML. Lo que permite chequear antes de procesar la consistencia de los mismos, es decir, que no falten etiquetas obligatorias o que no existan problemas de sintaxis. En el Fragmento 1 se puede ver una regla XML que ejemplifica todo lo explicado en esta sección.

Es importante destacar que este archivo XML puede ser, tanto construido con cualquier editor de texto plano o generado completando campos en una interfaz gráfica.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AudioPlugin SYSTEM "AudioPluginDef.dtd">
<AudioPlugin Version="0.1">
 <Metadata>
  <Name>clamTestRtPlugin</Name>
  <Description>This is a test realtime audio plugin</Description>
  <Authors>Fulano, Sultano, Mengano</Authors>
  <Copyright Year="2010">Club de Audio FIUBA</Copyright>
  <License>GPL</License>
  <Category>Plugins</Category>
 </Metadata>

 <Inputs>
  <Port Name="L Input">AudioInPort</Port>
  <Port Name="R Input">AudioInPort</Port>
  <Control Name="Gain" Min="0." Max="1.0"
DefaultValue=".5">FloatInControl</Control>
 </Inputs>
 <Outputs>
  <Port Name="L Output">AudioOutPort</Port>
  <Port Name="R Output">AudioOutPort</Port>
 </Outputs>

 <OutputPlugin Standard="CLAM" BuildSystem="Scons" OS="Linux">
  <BaseTemplateName>Default</BaseTemplateName>
  <CLAM_DefaultConfig> <!--CLAM plugin specific configuration-->
   <BaseClass WithConfig="True">Processing</BaseClass>
  </CLAM_DefaultConfig>
 </OutputPlugin>
</AudioPlugin>
```

Fragmento 1: Ejemplo de archivo XML con definición de plugin.

4.2. Generador de código a partir de templates

El esquema de trabajo es el que se puede observar en la Figura 2. En este caso los datos de entrada son los .xml que definen parte de la entropía del plugin, mientras que los templates son estructuras que se definen para cada estándar que se desea soportar (conservan las características comunes).

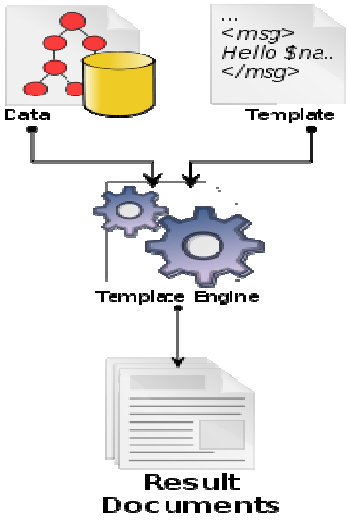

Figura 2: Motor de templates - Wikipedia [9]

Como salida se obtienen todos los archivos necesarios para compilar un plugin, una serie de archivos de código fuente y opcionalmente, el/los archivos correspondientes al sistema de construcción, documentación y descripción o metadata.

El sistema de templates funciona a través de variables predefinidas que se reemplazan de forma genérica y automática haciendo uso de expresiones regulares. Los templates se construyen a partir de la estructura común para todos, reemplazando por las variables ya mencionadas, en las partes donde el contenido debe variar (expande el código a partir de la entropía).

5. EXPERIMENTACIÓN E IMPLEMENTACIÓN

Se puede decir que el desarrollo de AP-Gen fue incremental [7][8]. Comenzó como un script sencillo, y fue tomando forma a medida que se le agregaban funcionalidades y fortalecían conceptos, hasta alcanzar su estructura definitiva.

La primera aproximación a la implementación de las ideas descriptas por este trabajo fue dentro del proyecto CLAM. Consistió en unos scripts capaces de crear esqueletos de processings de carga dinámica (los plugins para CLAM) a partir de parámetros pasados por línea de comando, como ser nombre, autor y licencia. Luego evolucionó permitiendo la generación de un código más completo a partir de especificar el processing por sus características de alto nivel. Para rápidamente incorporar una pequeña interfaz gráfica donde poder definir de forma más amigable, la metadata y la cantidad y tipo de entradas y salidas de datos y controles (ver Figura 3). Así como también encargarse de generar el sistema de construcción, para permitir que con solo ejecutar la orden de compilar, se pudiera usar inmediatamente.

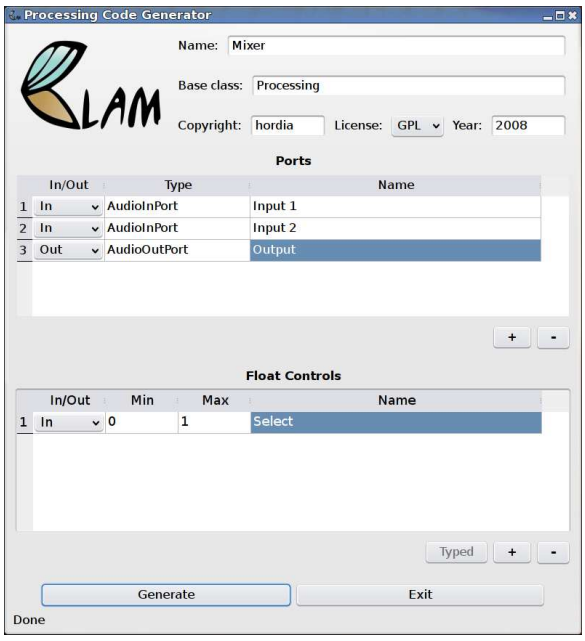

Figura 3: CLAM ProcessingCodeGenerator.

Con esto, se logró poder especificar el processing pensándolo desde el alto nivel, y en el siguiente paso, ponerse a codificar el Do(), el que se encarga del procesamiento en CLAM. La idea tuvo una buena recepción dentro del proyecto y actualmente esta incluido en sus repositorios oficiales. Usando esto, crear una distorsión del tipo clipping, se resume en especificarla en esta aplicación (de forma similar al mixer de la Figura 3) y agregar un código de procesamiento (como el mostrado en Fragmento 2) en uno de los archivos generados. Trabajando así, estos dos pasos son suficientes para poder utilizar el processing dentro del framework CLAM. Por ejemplo, armando una red en el NetworkEditor que consista en un lector de archivos de audio, o la entrada de línea de la pc, conectados a la entrada de este processing para clipping, y su salida, conectada a la salida de audio de la computadora.

Esta idea se perfeccionó en el contexto del Club de Audio de la FIUBA, donde ante la propuesta de trasladar a plugins algoritmos inicialmente prototipados en lenguaje de scripting Matlab, se observaron dificultades para comenzar con su programación, sin importar el sistema operativo de base (Linux o Windows).

Con una serie de refactorings sobre el ProcessingCodeGenerator, y apuntando a un diseño más general que incorpore la ideas explicadas en 4, se le dio la forma final a AP-Gen. Básicamente se reorganizo todo el código, incorporando el tema de las definiciones en XML y el motor de templates. Empezando por implementar la misma funcionalidad que originalmente estaba soportada para CLAM, pero con el nuevo diseño. Para después, agregar soporte para VST y LADSPA, programando uno a uno sus respectivos módulos.

```
bool Do(const Audio& in, Audio& out)
{
   int size = in.GetSize();
   const DataArray& inb = in.GetBuffer();
   DataArray& outb = out.GetBuffer();

   for (int i=0;i<size;i++)
   {
      if ( fabs(inb[i])>0.8 )
         outb[i] = inb[i]<0.? -0.8:0.8;
      else
         outb[i] = inb[i];
   }
   return true;
}
```

Fragmento 2: Código de processing CLAM.

Todo el desarrollo de AP-Gen se hizo utilizando el lenguaje de programación Python, principalmente por la claridad de código resultante que aporta y las facilidades para el tratamiento de texto plano incluidas en su librería estándar. Lo que se traduce directamente en menos errores, y un mantenimiento y escalabilidad más sencillo. Aunque por ahora solo fue probado en ambientes Linux, debería funcionar sin problemas en cualquier plataforma que Python soporte, como ser Windows o Mac.

6. RESULTADOS

Una vez funcionando, se pudo corroborar con creces la teoría inicial sobre que había una gran cantidad de código que se podía evitar escribir, cada vez que se comenzaba con el desarrollo de un plugin. Lo que se traduce directamente en un ahorro considerable de tiempo, no solo debido al que demanda escribirlo, sino también en el derivado de solucionar posibles errores humanos que pueden surgir al tener que realizar tal tarea en forma manual.

Todas las pruebas realizadas fueron exitosas, siempre que se especificó un plugin de la forma explicada, se pudo crear automáticamente el esqueleto del mismo, de forma tal de poder pasar rápidamente a la programación del algoritmo de procesamiento, es decir, el objetivo inicial. En los casos que surgió algún problema, fue debido a errores menores en la implementación, los cuales fueron solucionados en el corto plazo, o a lo sumo, en el peor de los casos, motivaron pequeños re-diseños internos.

Pero más allá de esto, lo interesante en este contexto, son los resultados de las ideas. El modelo no perdió validez, y desde su concepción, cada vez que surgió la ocasión, el autor recurrió sin dudar a AP-Gen como herramienta de apoyo en el desarrollo de plugins de audio. Así como, a pesar de su reciente implementación, se tuvo la oportunidad de mostrar y validar con éxito AP-Gen y sus ideas entre los integrantes del Club de Audio de la FIUBA y otros grupos de la especialidad.

6.1. Ventajas del framework

Las ventajas de trabajar en la forma que propone AP-Gen, se pueden resumir en:

- Ahorra tiempo. Permite que el programador se pueda concentrar rápidamente, y prácticamente solo, en el desarrollo de la función que efectúa el procesamiento (DSP).
- Provee una estructura favorable para el desarrollo comercial de plugins multiplataforma (diferentes estándares y sistemas operativos). Permite diferentes salidas a partir de la misma definición.
- Simplifica la tarea de portar código de un plugin ya codificado a un estándar diferente. Facilitando el desarrollo en el lenguaje nativo, evitando el uso de wrappers o librerías externas.
- Propone buenas prácticas de desarrollo que se traducen en mejores resultados (abstracciones, desacoplamiento del código,

estructura general templatizada y convenciones).

- Facilita el desarrollo para la distribución de algoritmos ya prototipados en Matlab o similar, como plugins compatibles con cualquier programa host.
- Posee soporte modular y extensible para las diferentes arquitecturas de plugins.
- Evita que el desarrollador tenga que ocuparse del sistema de construcción.
- Fines educativos. Sirve como referencia introductoria para conocer un determinado estándar.
- Puede ser usado desde consola o desde una interfaz gráfica.
- Esta publicado como Software Libre.

7. CONCLUSIÓN Y TRABAJO FUTURO

Teniendo presentes las habituales complicaciones que surgen durante el ciclo de desarrollo de plugins de audio, se logró proponer una nueva forma de trabajo basada en la herramienta AP-Gen. La cual plantea un modelo abstracto del plugin, y a partir del mismo genera código fuente de base automáticamente, lo cual permite el trabajo simultáneo con múltiples estándares de audio, y por sobre todo, ahorrar tiempo valioso a programadores experimentados y facilitar el comienzo a los principiantes.

En cuanto a sus diferencias con otros entornos de desarrollo que facilitan o potencian la programación de plugins, no se considera un enfoque mejor ni peor, sino simplemente distinto y en algunos casos, complementario. En pocas palabras, se trata sobre generar código y de evitar el trabajo repetitivo, pero necesario. Nada que no se pueda hacer manualmente sin utilizar esta herramienta. Pero no deja de ser una cuestión que se vuelve sumamente útil cuando se tiene que generar varios plugins diferentes, o se necesita llegar más rápido o fácil, al momento del desarrollo de la parte de DSP. Como siempre, para cada caso, si este es el enfoque indicado o no, dependerá de los objetivos y requerimientos así como de los tiempos y background del desarrollador de turno.

Pero por sobre todo, y debido a los múltiples beneficios que trae, se concluye que lo más importante, más allá de la herramienta utilizada, consiste en el hecho de pensar en todo momento al plugin como una estructura abstracta, desacoplada de la implementación.

El contraste de las ideas aquí presentadas, fue siempre positivo, pero queda como tarea pendiente aumentar la difusión del proyecto. Para el futuro, se deja abierta la posibilidad de tomar diferentes direcciones. Una interesante, por ejemplo, consiste en portar la automatización que provee AP-Gen a un complemento del IDE Eclipse, para poder realizar todas las actividades que comprenden al desarrollo desde el mismo programa, de forma similar a como actualmente ocurre en el mismo cuando se quiere crear una clase nueva, la que es capaz de construir a partir del nombre y otras definiciones opcionales.

Por otra parte, por diseño, siempre esta la posibilidad de implementar nuevos módulos que den soporte a nuevos estándares, por ejemplo Audio Units, RTAS y lv2 que actualmente no están soportados. O los estándares de instrumentos VSTi o dssi. Así como construir templates generativos para otros lenguajes como Java o Python. Otra línea de investigación, podría consistir en intentar conectar de algún modo el estándar XML aquí propuesto y utilizado para la definición general del plugin, con el formato rdf que propone el estándar lv2, el cual todavía no se investigó a fondo.

Por la línea de extender el framework a otro tipo de funcionalidades, también puede ser interesante proveer, como funcionalidad separada, una herramienta de testing automatizado del código de plugin resultante. Es decir, una aplicación que se encargue de correr una serie de pruebas estandarizadas para cada arquitectura, para que funcione como una primera verificación de su implementación. Como un host que carga el plugin e interactúa con el mismo, tratando de encontrar puntos débiles en su programación y viendo como se comporta ante ciertas cosas tal vez no esperadas, pero permitidas por el estándar, y que pueden llegar a ocurrir normalmente.

NOTA: AP-Gen se puede descargar de http://code.google.com/p/ap-gen/ y se distribuye bajo una licencia de Software Libre. La cual permite estudiar el programa y adaptarlo, su utilización con cualquier propósito, la distribución de copias y hacer públicas posibles mejoras, de modo que se pueda beneficiar la comunidad de usuarios.

1. REFERENCIAS

[1] Goudard V and Muller R. “Real-time audio plugin architectures”. Comparative study. IRCAM - Centre Pompidou. France. 2003.

[2] Amatriain X. “An Object-Oriented Metamodel for Digital Signal Processing”. PhD thesis. Universitat Pompeu Fabra. Spain. 2004.

[3] Amatriain X., Arumi P. and Garcia D. “CLAM: A Framework for Efficient and Rapid Development of Cross-platform Audio Applications”. Proceedings of ACM Multimedia. pp. 951-954. Santa Barbara, CA, USA. 2006.

[4] Garcia D. and Arumi P. “Visual prototyping of audio applications”. Proceedings of 5th International Linux Audio Conference. Berlin, Germany. 2007.

[5] Orlarey Y., Fober D. and Letz S. “FAUST : an Efficient Functional Approach to DSP Programming”. New Computational Paradigms for Computer Music. Editions Delatour France. 2009.

[6] Hunt A. and Thomas D. “The Pragmatic Programmer: From Journeyman to Master”. Book. Addison-Wesley Professional. 1999.

[7] Larman C. and Basili V. R. “Iterative and Incremental Development: A Brief History”. IEEE Computer. Vol. 36 no. 6, pp. 47-56. June 2003.
[8] Cockburn A., “Agile Software Development”. Paperback. Addison-Wesley Professional. 2002.
[9] Wikipedia, the free encyclopedia. “Template processor”.http://en.wikipedia.org/wiki/Template_processor. Revision 12:23, July 1st, 2010.