UNIVERSIDAD DE CANTABRIA Facultad de Ciencias

Proyecto Fin de Carrera

Modelado en SystemC del codificador de vídeo y audio H.264/MPEG-4

Modelling on SystemC of Audio/Video encoder H.264/MPEG-4

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Antonio Cuesta Margallo

Septiembre - 2013

Facultad de Ciencias

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Antonio Cuesta Margallo
Director del PFC: Enrique Vallejo Gutiérrez
Ponente (en su caso): Eugenio Villar Bonet
Título: "Modelado en SystemC del codificador de vídeo y audio H.264/MPEG-4"
Title: "Modelling on SystemC of Audio/Video encoder H.264/MPEG-4"
Presentado a examen el día:

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:
Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

Índice

Agradecimientos ..... *I*

Resumen ..... *I*

Abstract ..... *V*

1.- Introducción y planificación *4*

1.1.- Codificación H.264\MPEG-4 4

1.1.1.- JM8 5

1.1.2.- Codificación IPB 6

1.2.- Descomposición estructurada del Proyecto 6

1.2.1.- Desarrollo de la descomposición 7

1.2.2.- Diccionario 9

1.2.3.- Entradas de la estructura 10

1.2.4.- Salidas de la estructura 10

1.3.- Metodología de trabajo 10

1.4.- Herramientas utilizadas en el desarrollo 12

2.- Motivación y Objetivos *13*

2.1.- Implementación en SystemC de un modelo basado en un codificador H.264/MPEG-4 en C. 13

2.2.- Aprendizaje de SystemC y sus mecanismos 14

3.- SystemC *15*

3.1.- ¿Qué es SystemC? 15

3.2.- Complejidad del sistema 17

3.3.- Elementos del sistema 18

3.3.1.- Módulos 18

3.3.2.- Procesos 19

3.3.3.- Puertos, canales y señales. 19

3.4.- Comparativa con otros lenguajes 20

4.- Estado del arte *22*

4.1.- SystemC 22

4.2.- Compresión de audio y video 22

5.- Captura y análisis de requisitos *24*

5.1.- Requisitos funcionales 24

5.2.- Requisitos no funcionales 25

6.- Diseño *27*

6.1.- Diseño del modelo original en C 27

6.2.- Diseño del modelo SystemC 29

6.2.1.- División en módulos del sistema 29

6.2.1.1.- Reparto de funcionalidad entre los módulos 30

6.2.1.2.- Reorganización de variables globales 31

6.2.2.- Entradas y salidas 32

6.3.- Diagrama de flujo del sistema 32

7.- Implementación *34*

7.1.- Implementación en SystemC 34

7.2.- Pruebas de la implementación 36

7.2.1.- Unitarias ____ **36**
7.2.2.- Integración ____ **36**
7.3.- Problemas y soluciones ____ **38**
7.3.1.- Problemas de entorno ____ **39**
7.3.2.- Problemas de código ____ **39**

8.- Evaluación ***41***
8.1.- Pruebas de Sistema ____ **41**
8.1.1.- Prueba 1: Funcionamiento correcto ____ **41**
8.1.2.- Prueba 2: Porcentaje de utilización de funciones y tiempo de ejecución relativo ____ **42**

8.1.3.- Prueba 3: Velocidad media de compresión de una imagen ____ **45**
8.1.4.- Prueba 4: Comprobación de los ficheros de salida. ____ **51**
8.1.5.- Prueba 5: Comparación de velocidad entre el sistema creado y el original ____ **51**
8.2.- Problemas encontrados y soluciones ____ **51**
8.3.- Conclusiones ____ **52**

9.- Conclusiones generales y trabajos futuros ***53***

IV.- Bibliografía ***54***

Índice de figuras

Figura 1: Estructura general del proyecto 7
Figura 2: Diagrama completo de grano grueso 7
Figura 3: Aplicación del algoritmo al proceso de Requisitos 7
Figura 4: División del proceso de Diseño 8
Figura 5: Especificación del bloque de Construcción 8
Figura 6: Estructura de Pruebas desglosada 9
Figura 7: Diseño total del sistema de trabajo 9
Figura 8: Metodología en cascada modificada 11
Figura 9: Estructura de la arquitectura SystemC 16
Figura 10: Definición estándar de un módulo 19
Figura 11: Comunicación mediante sc_fifo 20
Figura 12: Comparativa de uso de lenguajes 21
Figura 13: Diagrama de casos de uso de nuestro sistema 24
Figura 14: División de los requisitos no funcionales 25
Figura 15: Diagrama de clases del código C 27
Figura 16: Conexiones del módulo "lencod.c" 28
Figura 17: Diseño del sistema que desarrollaremos en SystemC 30
Figura 18: Diagrama de flujo del sistema 33
Figura 19: Módulos sin funcionalidad ni conexiones 34
Figura 20: Aplicación del main.cpp 34
Figura 21: Definición de cada módulo 35
Figura 22: Ambos módulos funcionando 35
Figura 23: Resultados de la ejecución en la prueba de comunicación 37
Figura 24: Resultados de la comunicación 38
Figura 25: Estructura del externC 39
Figura 26: Función definida del operador "<<" 40
Figura 27: Resultado de la ejecución original y en nuestro sistema. 42
Figura 28: Resultado de la ejecución de Gprof que muestra los porcentajes de utilización. 42
Figura 29: Estructura de lectura de tiempos. 43
Figura 30: Periodo de ciclo del sistema 44
Figura 31: Gráfica temporal de cada ejecución 46
Figura 32: Tiempo medio de ejecución de la prueba II 47
Figura 33: Gráfica de tiempos de ejecución totales en la prueba III 48
Figura 34: Muestra de la variación de tiempo de ejecución para la prueba IV 49
Figura 35: Tiempos medidos de ejecución total sobre el video V. 50

Índice de Tablas

Tabla 1: Diccionario del EDT ____ *10*
Tabla 2: Reparto del trabajo en los módulos ____ *30*
Tabla 3: Funcionalidad implementada en los módulos ____ *37*
Tabla 4: Especificaciones de los sistemas de pruebas ____ *41*
Tabla 5: Resultados de la medición temporal ____ *43*
Tabla 6: Muestras temporales y percentuales de carácter general del sistema ____ *43*
Tabla 7: Módulo de las funciones más consumidoras temporalmente del sistema. ____ *44*
Tabla 8: Tiempos separados y porcentaje sobre el total de cada tipo de imagen ____ *44*
Tabla 9: Videos elegidos con su duración temporal y espacial. ____ *45*
Tabla 10: Anotaciones temporales para cada imagen en cada prueba. ____ *46*
Tabla 11: Resultados finales del video I. ____ *46*
Tabla 12: Resultados finales del video II. ____ *47*
Tabla 13: Resultado medio de la prueba II. ____ *47*
Tabla 14: Medidas temporales de la prueba III ____ *48*
Tabla 15: Tiempos medios de la prueba III ____ *48*
Tabla 16: Tiempos medidos en la prueba IV ____ *49*
Tabla 17: Tiempos medios de la prueba IV ____ *49*
Tabla 18: Medidas tomadas en la prueba V ____ *50*
Tabla 19: Tiempos medios de la última prueba ____ *50*
Tabla 20: Resultados finales de la velocidad media ____ *51*
Tabla 21: Ficheros de salida del sistema ____ *51*
Tabla 22: Comparación entre ambos sistemas ____ *51*

Agradecimientos

Los agradecimientos son siempre la parte más complicada de escribir, porque hay mucha gente a la que estoy agradecido, y espero no dejarme a nadie. Por mi propio bien, y mi integridad física.

La primera persona a la que estoy agradecido es mi madre, Fátima Margallo, sin ella no sé dónde hubiera terminado ni qué hubiera acabado pasando conmigo. Siempre has estado ahí, y desde que tengo uso de razón me motivaste para que fuera "informático". Y la verdad es que nunca has dejado que eligiera otro camino, pero no me arrepiento. Gracias, por este y todos los proyectos que hubo y quedan por venir, donde no me cabe duda que me seguirás apoyando.

No hubiera podido llegar hasta aquí sin la ayuda de la Facultad de Ciencias en su conjunto, que ha sido como un hogar para mí en los últimos años, de una forma casi literal. Con una especial mención a Eugenio Villar, por proponerme este proyecto, su infinita paciencia con mi inutilidad con el e-mail y la confianza que ha demostrado tener en mí en los últimos meses. También debo agradecerle a Enrique Vallejo su paciencia y sus útiles consejos a la hora de realizar este documento. A la próxima en El Bosc de las Fades invito yo.

Otro gran apoyo estos últimos meses de trabajo ha sido mi familia en conjunto: mis hermanos, mi abuela y mi padre. Quizá he vivido una época un poco estresante y he estado más susceptible de lo normal, pero eso no significa que dejéis de ser los mejores.

A todos los N+1: Alberto, Carlos, Dani, Payno, Rober y Victor. Gracias por todo el tiempo que hemos compartido, todos los comics que me queden por dibujar y todas las cenas que nos queden por no hacer. Garm, Ale, JJ, Torren, Meneillos, Ruper, Javi, Jaime, Tote, Berto, Manu, Karla, Chente… y toda la gente a la que le he dado la chapa con mis problemas los últimos meses, pero que han sabido escucharme aún sin saber de qué les hablaba. Gracias también a vosotros.

A veces las cosas no salen como uno quiere, pero no por ello puedo dejar de sentirme agradecido en muchos aspectos a una persona que me animó durante la realización de este proyecto. Creo que sin ti hubiera tirado la toalla mucho tiempo atrás, gracias.

Ya llego al final, pues sólo me queda alguien que no necesita que se las de para saber que le estoy profundamente agradecido. Al principio he mentido, no es una persona.

Resumen

Con este proyecto se pretende realizar un estudio sobre la utilidad del lenguaje SystemC como herramienta de evaluación de sistemas co-diseñados HW/SW para las empresas. Además se intentará preparar al autor del mismo para un perfil profesional acorde a este tipo de trabajos, mediante un estudio en profundidad del lenguaje SystemC y el co-diseño de este tipo de sistemas, así como con el aprendizaje sobre la codificación de audio y video.

Para ello se ha elegido transformar un sistema en código C para realizar pruebas sobre una posible implementación Hardware de un conversor de audio y video de alta calidad bajo el estándar H.264/MPEG-4. Con esto se pretende crear un sistema software que nos permita simular y verificar el futuro elemento, a nivel de comportamiento y de rendimiento. Como no es posible calcular el rendimiento real del elemento final, estas medidas de rendimiento serán relativas, es decir, no será un rendimiento absoluto sino un speedup del elemento con respecto al código original y al aplicarle diferentes estrategias de mejora.

Como H.264/MPEG-4 es un estándar, se utilizará la implementación libre ofrecida por el Fraunhofer Heinrich Hertz Institute como código del que partir para trabajar después. Se realizará un encapsulado de las funciones más importantes en módulos propios de un lenguaje de descripción de Hardware (como pueden ser Verilog o VHDL) y a continuación se realizarán pruebas sobre el sistema para comprobar su funcionamiento correcto.

Además de estas pruebas, puramente de comportamiento, para demostrar que el nuevo sistema trabaja igual que el original, también se van a realizar pruebas sobre los distintos elementos para obtener una descripción y un conocimiento más profundo sobre el sistema que estamos creando.

Palabras clave: SystemC, sistemas embebidos, co-diseño HW/SW, simulación HW/SW.

Abstract

This project aims to realize and study about the usability of SystemC as an evaluation methodology of HW/SW co-design of systems to companies. Furthermore, the author attempts to prepare a professional profile in this kind of jobs, through the study of SystemC and the HW/SW co-design, as well as learning about audio and video encoding.

For that, we have chosen to transform a C code into a future and fiction Hardware encoder under H.264/MPEG-4. We want to create a software system that let us simulate and verify the behavior and performance of the element. As it is not possible to know the final performance, we'll use a reference measure for that. We will use that measure to calculate the speedup about the original code.

As H.264/MPEG-4 is a standard, we will use the free implementation offered by the Fraunhofer Heinrich Hertz Institute like initial code. The most important functions will be encapsulated into modules, specifics of a hardware description language (like Verilog or VHDL), and there'll be some tests on the system.

In addition of these tests, which only try the behavior of the new system, there'll be some proofs about the elements and modules that compose the final system, to get a high description and knowledge about it.

Keywords: HW/SW Co-design, SystemC, Embedded Systems, HW/SW simulation.

Capítulo 1

Introducción y planificación

Este primer capítulo introductorio va a tener la finalidad de explicar brevemente al lector la planificación del proyecto y qué tecnología se utilizará para desarrollarlo, así como las herramientas empleadas tanto para el código desarrollado como para el documento. El propio lenguaje SystemC será el único punto explicativo que no se trate aquí, sino que tendrá un capítulo propio más adelante.

1.1.- Codificación H.264\MPEG-4 [3][6]

La historia del estándar de codificación de audio y video H.26x/MPEG-x se remonta a principios de los 90, cuando fue creado por expertos en codificación de video de la Unión Internacional de Telecomunicaciones y el grupo de expertos para imágenes en movimiento de la ISO/IEC, enfocado a la codificación de video de aplicaciones para imágenes en movimiento.

Se desarrollaron a lo largo de los 90 la versión 1, 2 y 3. La 4 es de 2000 y tiene dos partes que hayan visto la luz: la parte 2 y la parte 10. Es un estándar que maneja perfiles diferentes en función del tipo de compresión y la velocidad que se quiera.

Dichos creadores se unieron fundando el JVT, Joint Team Video (al que se hace referencia en el código indicando que se está usando la versión JVT-D101), haciendo que viera la luz la versión H.264 o MPEG-4 en 2003. Cuenta con una sintaxis más sencilla que las versiones anteriores, lo que representa una mejor integración con los protocolos y las arquitecturas múltiples actuales.

Para la codificación y decodificación en diferentes modos, se crearon lo que se llamó perfiles. Cada uno de ellos especifica características y supone límites a la decodificación pero no a la codificación, por lo que en este proyecto no tienen importancia; aun así se explicarán a continuación. Existen 3 perfiles en la primera versión del estándar:

Línea base o baseline: Aplicado a servicios de conversación, como videoconferencias. Además añade:

- Ordenación flexible de los macrobloques: no necesitamos ordenarlos, se usa un mapa.
- Ordenamiento arbitrario de los slices: pueden tener el orden que sea, pero estamos seguros que será correcto.
- Slice redundante: mantiene un slice ya codificado con los mismos datos.

Principal o Main: Almacenamiento de video, datos y TV. Además de las características que se mencionarán más adelante, añade el uso de CABAC como método de codificación de la entropía.

Extendido: Aplicado a servicios multimedia en Internet. Incluye todos los perfiles de la línea base y además:

- Slices SP y SI: slices de mejora respecto a las de tipo P e I que añaden eficiencia en la resolución.
- Particionado de datos: reduce errores de transmisión. El medio inalámbrico es un problema para este tipo de datos susceptibles a un error.

Alto o High: Utilizado en video de 8 bits con crominancia[1] 4:2:0 y aplicaciones de alta resolución. Utiliza el perfil main y además tiene transformación por bloques de tamaño adaptativo y matrices de cuantificación escalada.

H.264 está dividido en dos capas, una primera capa de abstracción (NAL) y una segunda de codificación de video (VCL). NAL abstrae los datos para poder utilizarlos en el bitstream de salida en el proceso de codificación. VCL es la secuencia de video a codificar, cuadros dentro de una secuencia de video, slices dentro de cada cuadro, macrobloques en cada slice y bloques dentro de los macrobloques.

El algoritmo de codificación utilizado es dependiente del slice que se va a utilizar: si son de tipo P o B utilizaremos codificación INTER y si tenemos una trama I utilizamos una INTRA. En ambos casos dependerá del perfil que estemos utilizando si se usará un codificador de entropía u otro (CAVLC[2] o CAB AC[3]).

Ésta es suficiente información para lo que realmente se necesita conocer sobre el codificador para realizar este proyecto. No es necesario entrar en mayor profundidad pues no vamos a trabajar sobre cómo codifica o cómo utiliza las capas o las entropías de codificación.

1.1.1.- JM8

JM8 es la versión que se ha decidido utilizar para la implementación C de la que parte este proyecto. En la actualidad existe la JM16. Este software es una versión programada del estándar H.264/MPEG-4, desarrollado por el Fraunhofer Heinrich Hertz Institute en colaboración con Microsoft.

Originalmente el código provee de un codificador y un decodificador, pero este proyecto se centrará en el codificador, pues el decodificador supondría un trabajo análogo.

Se eligió la versión 8 y no la 16 por la simplicidad a la hora de discernir elementos y de funcionalidad. La última tiene más código y más complejo que la primera.

[1] En una señal de video, la información perteneciente al color.
[2] Context-Adaptative Variable Length Codes
[3] Context-Adaptative Binary Arithmetic Codes

También fue elegida porque se pretende utilizar código C mezclado con código SystemC en lugar de C++, utilizado en las versiones más modernas.

1.1.2.- Codificación IPB

Como ya se explica en el primer apartado de este epígrafe, el estándar H.264 divide las imágenes en slices en función del tipo de perfil que estemos utilizando. Existen tres tipos de perfiles, pero dos de los tipos de slices son comunes a todos ellos:

Tipo I: Predicción de tipo INTRA. Se codifica mediante predicción de muestras del mismo slice.

Tipo P: Predicción de tipo INTER. Se codifica mediante predicción de muestras de slices anteriormente codificados. Utiliza un vector de movimiento y un índice de referencia para la predicción.

El último tipo de slice (B) depende del perfil que estemos utilizando. En este proyecto se utiliza una parte del estándar que corresponde con el perfil de "principal" o main. Este perfil se utiliza para almacenamiento digital de video y datos, además de para la transmisión de televisión.

Tipo B: Predicción de tipo INTER, se codifica mediante predicción de muestras de slices anteriormente codificados (igual que el tipo P). Sin embargo, la diferencia con las de tipo P es que este tipo de slice utiliza dos vectores de movimiento, además del índice de referencia.

También nos proporciona una predicción en función del peso sobre los slices de tipo P y B.

1.2.- Descomposición estructurada del Proyecto

Esta parte del proyecto será importante para discriminar en qué etapas se va a dividir el mismo a la hora de trabajar. Es muy importante tener bien claro qué objetivos se persiguen, pero además cómo se consiguen.

¿Con esta descomposición qué se pretende? Que cada fase de desarrollo en la que lo dividiremos delimite el tipo de trabajo que haremos en cada fase del proyecto. Como es lógico, esta descomposición no distará mucho de las de otros proyectos de similares características, pero eso no lo hace menos importante. Además nos permitirá estimar qué volumen de trabajo y de tiempo vamos a dedicar a la parte del proyecto en que nos encontremos.

1.2.1.- Desarrollo de la descomposición

Para comenzar con la descomposición, hemos detectado las entradas y salidas de nuestro sistema (de las que hablaremos más en profundidad en las subsecciones correspondientes). Una vez quedan establecidas, se realiza un esquema básico de la estructura del proyecto.

Figura 1: Estructura general del proyecto

El proyecto va a estar dividido en 4 etapas bien diferenciadas, que harán uso de las entradas que se muestran en la *figura 1*. La primera será útil en la fase de captura y análisis de Requisitos, y la segunda en la fase de Diseño del sistema. Así que por lo menos existirán esas dos divisiones. Además se añadió una fase de Construcción y otra de Pruebas.

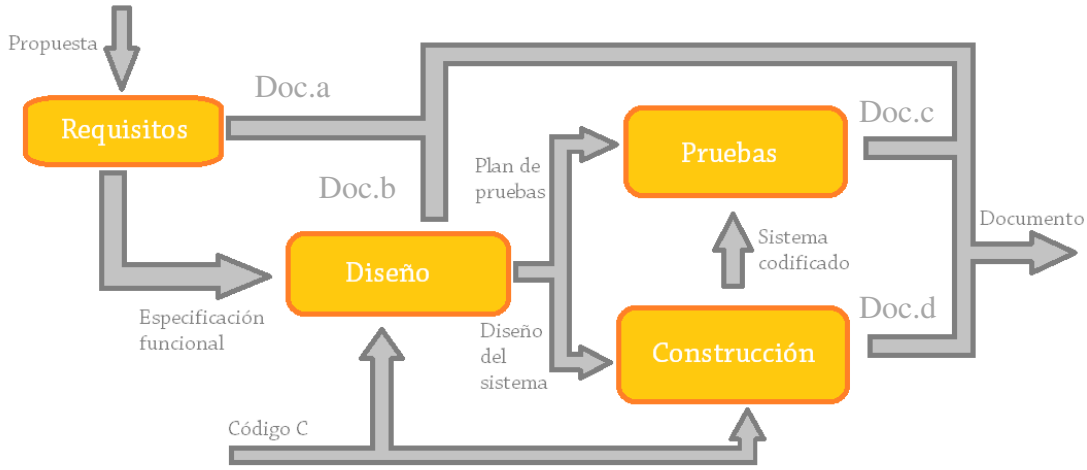

Figura 2: Diagrama completo de grano grueso

Las relaciones establecidas corresponden al flujo de información obtenido a través de la primera fase y que será útil para continuar subdividiendo el esquema.

En la fase de Requisitos se toman tanto los requisitos funcionales como los no funcionales, y será utilizada esta información en la fase de diseño. Así que la fase de Requisitos queda subdividida como se muestra en la *figura 3*

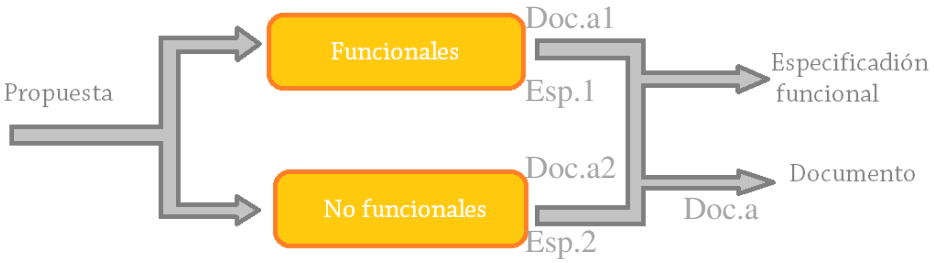

Figura 3: Aplicación del algoritmo al proceso de Requisitos

En la fase de Diseño existirán como entradas el documento de requisitos que se ha generado en la fase anterior y el código C. De él se va a hablar a lo largo de este capítulo y existe un pequeño modelo superficial de cómo funciona el sistema. A partir de este código se crea el modelo de comportamiento. Se ha extraído con qué datos está trabajando el codificador y cuáles son funciones principales. Después se creará un modelo SystemC que tendrá en cuenta este análisis para generar una estructura de módulos y la comunicación entre estos para resolver el problema con la mayor exactitud posible.

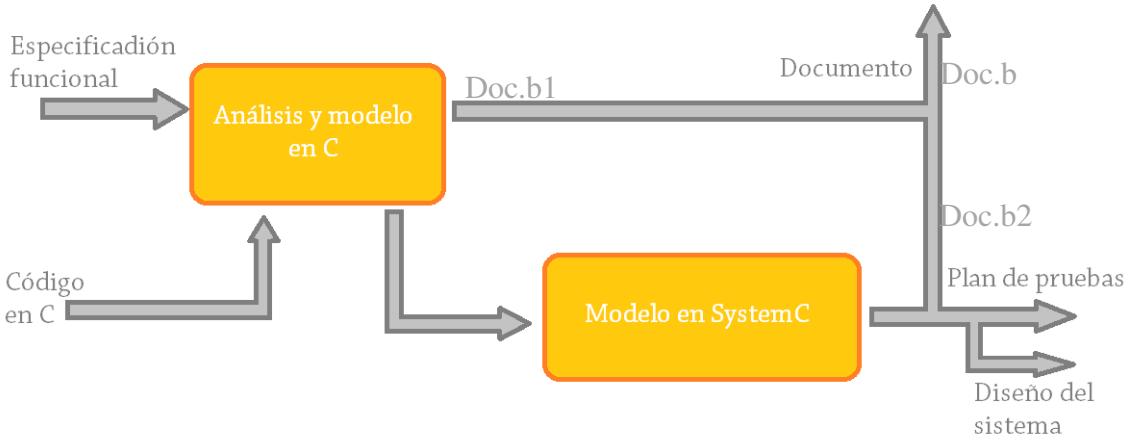

Figura 4: División del proceso de Diseño

Una vez hecho el diseño de cómo será el sistema, se continúa con el apartado de construcción. Conociendo la estructura a posteriori del sistema, vamos a tener 3 fases de creación. En una primera etapa se creará la parte de Testbench, que formará la capa externa del sistema, encargada del proceso de pre y poscodificación, una segunda etapa donde se creará el módulo de codificación como tal, y una etapa final en la que se construirá el módulo principal que instanciará los dos anteriores y los comunicará. Así que el proceso de construcción tendrá el aspecto de la *figura 5*.

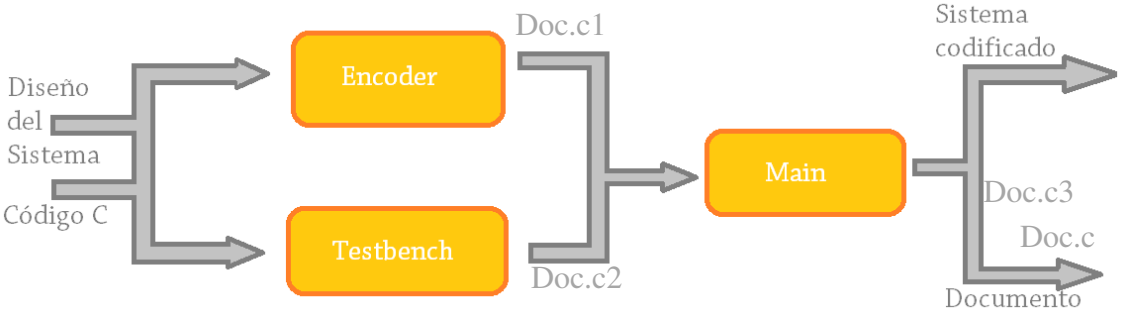

Figura 5: Especificación del bloque de Construcción

La última etapa, que va a depender tanto del Diseño como de la Construcción, será la de Pruebas. Se pretende realizar pruebas a tres niveles diferentes del código: unitarias, de integración y de sistema. Las dos primeras serán de tipo de caja blanca, las últimas de caja negra. A pesar de que se realizarán en momentos diferentes de la etapa de desarrollo del sistema, se incluyen en la misma división en este caso.

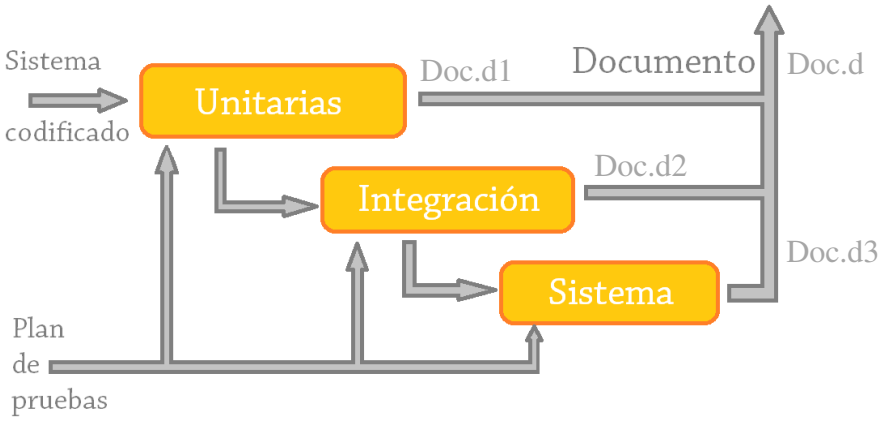

Figura 6: Estructura de Pruebas desglosada

Con esta división de las tareas, se obtiene el sistema de flujos de trabajo de la actividad que se va a desarrollar en este proyecto como se muestra en la *figura 7*. En azul se han señalado los canales de entradas de información al modelo y en verde las salidas, en este caso una sola de documentación.

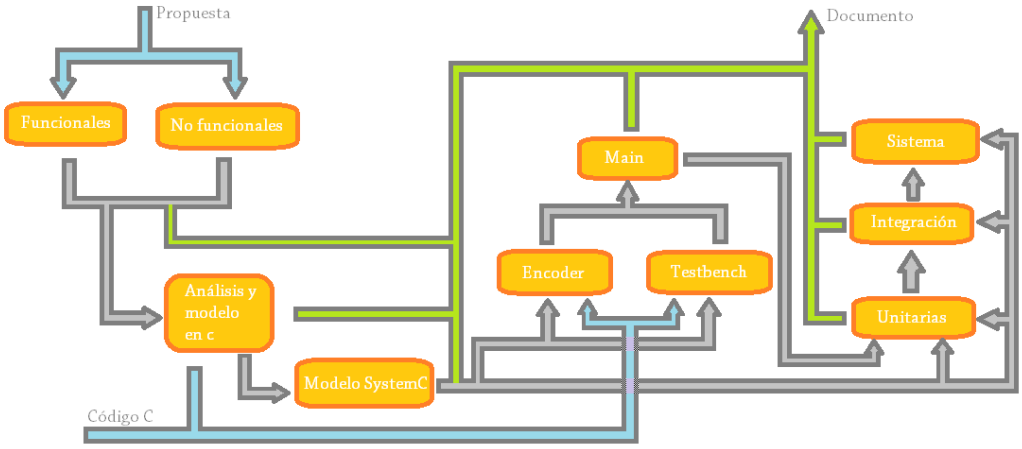

Figura 7: Diseño total del sistema de trabajo

1.2.2.- Diccionario

Como cualquier SFT, se ha creado un diccionario para entender cómo se relacionan las diferentes partes del sistema y de quién depende el entregable DOC. Se ha establecido el diccionario como se muestra en la *tabla 1* a continuación.

El diseño del diccionario es sencillo, pero no existe una relación de salida más que la de documentación, por lo que es la única que está compuesta por más de un elemento.

Valor	Composición
Código C	
Propuesta	
Documento	Doc.a + Doc.b + Doc.c + Doc.d
Especificación funcional	Esp.1 + Esp.2

Plan de pruebas	
Diseño del sistema	
Sistema codificado	
Doc.a	Doc.a1 + Doc.a2
Doc.b	Doc.b1 + Doc.b2
Doc.c	Doc.c1 + Doc.c2 + Doc.c3
Doc.d	Doc.d1 + Doc.d2 + Doc.d3

Tabla 1: Diccionario del EDT

1.2.3.- Entradas de la estructura

Existen dos entradas externas al SFT. La primera va a ser el documento de especificaciones iniciales del cliente. En este caso el cliente vamos a ser nosotros mismos y se utilizará como documento el resumen que encabeza este documento.

Como es lógico y ya se ha explicado, a partir de estas especificaciones se hará un análisis y una elicitación de los requisitos. Así que es importante que sea claro y se puedan extraer el mayor número de requisitos, porque así será más conciso y por tanto más controlado estará el proceso de diseño e implementación.

Una segunda entrada será el código original, que será un apoyo para crear tanto el modelo en el diseño como el sistema en SystemC. Será analizado en profundidad en el apartado de diseño para saber tanto las entradas y salidas de los módulos como la división como tal en módulos de trabajo.

1.2.4.- Salidas de la estructura

La única salida que tendrá el sistema será la información que se plasmará en este documento. Se ha generalizado de esta manera porque es donde irán ambas partes del sistema, la que pertenezca al diseño y la que contenga las conclusiones de las pruebas que se realicen.

Obviamente no será igual la documentación generada en una etapa que en otra, pero como todas convergen en un documento escrito común, se ha tomado la decisión de generalizarlo sin entrar en detalles en el diccionario.

1.3.- Metodología de trabajo

A pesar de las negativas opiniones de la comunidad académica e industrial sobre la metodología en cascada, sigue siendo de las más utilizadas hoy en día en el desarrollo, por lo que se ha decidido seguir este tipo de modelo de trabajo con ciertas modificaciones.

Primero se ha generado la cascada en sí, para tener claro cómo vamos a empezar y qué pasos vamos a seguir. La especificación final está definida de forma visual en la *figura 8*.

Dentro de esta cascada, hay un primer punto en el que se hará un análisis de los requisitos que tendrá nuestro sistema. Una vez se tengan claros estos requisitos, atravesando el "gate" llegaremos al siguiente nivel. Es muy importante que quede todo perfectamente especificado antes de continuar al siguiente nivel.

En un segundo nivel, siguiendo con los requisitos ya especificados, tendremos la fase de diseño. En esta fase se generará parte de la información que aparece en este documento y además será la base sobre la que se construirá todo el sistema de codificación. En esta etapa también habrá que prestar especial atención a no cometer ningún error de diseño, o habrá que volver atrás más adelante con la consiguiente pérdida de tiempo.

Abriendo el "gate" del segundo nivel se llegaría al de la implementación. Este es el nivel más largo y posiblemente el más complicado. En él se completará todo el trabajo que se hizo en los niveles anteriores con un esfuerzo a nivel de programación grande. Se debería crear el sistema entero antes de pasar al nivel de pruebas, por lo que se tendría que estar completamente seguro de que la implementación es correcta. Sin embargo, aquí reside la diferencia con el modelo clásico en cascada: vamos a crear una retroalimentación entre este nivel y el de pruebas. De tal modo que se irá creando el sistema intercalándolo poco a poco con pruebas unitarias y de integración, y una vez completado todo el proceso, se tendrá un sistema funcional a estos niveles.

Cuando esté completado todo este trabajo, se podrá continuar en un nivel más de pruebas, en el que se realizarán pruebas de sistema. Es decir, pruebas de comportamiento general del sistema, que es lo que nos interesa, donde veremos la eficacia, la velocidad, si el comportamiento es correcto…

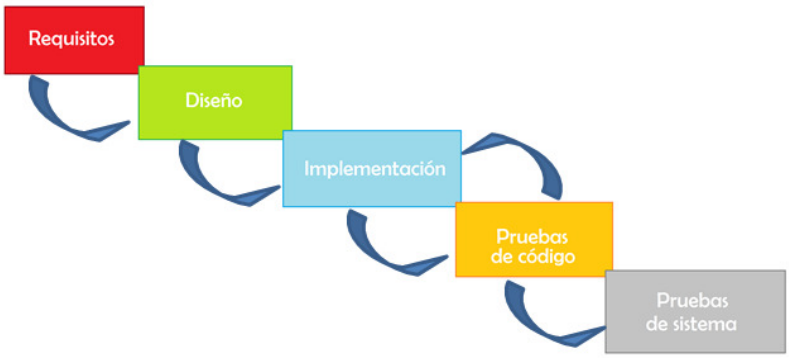

Figura 8: Metodología en cascada modificada

Con este modelo en cascada personalizado, se pretende paliar los principales problemas de estos modelos, sobretodo el inconveniente de que no se puede avanzar de etapa sin que se haya completado una y que un error de pruebas supone un rediseño. Por ello incluir una etapa de pruebas a bajo nivel cuasi-paralela al desarrollo del software nos permitirá eliminar en gran parte los errores que pueda haber.

1.4.- Herramientas utilizadas en el desarrollo

En la elaboración de este proyecto se han utilizado varias herramientas software tanto para el desarrollo como para la documentación, así como para las pruebas realizadas en el sistema. Por lo que se ha creído conveniente hacer referencia y explicar cada una de ellas:

- **SystemC Eclipse con librerías**: Ésta ha sido la herramienta principal de trabajo a lo largo de todo el proyecto. Eclipse es un entorno de desarrollo integrado (IDE) con la expansión pertinente para permitir la compilación de código C/C++. Este entorno nuevo no es tan evolucionado para trabajar de forma correcta y a veces introduce pequeños errores que han supuesto un tiempo perdido en el progreso del proyecto. Además sobre esto se han añadido las librerías de SystemC instaladas en el SO. Se eligió este IDE por la familiaridad y la aparente utilidad del debugger.
- **Gprof**: Como en cualquier sistema, este proyecto va a tener unas funciones que van a consumir más tiempo de proceso que otras. Esta herramienta va a permitir identificar esas funciones y tratar de mejorar su rendimiento. Al igual que las librerías SystemC este también necesita una serie de librerías instaladas sobre el SO, y se utiliza a través de Eclipse.
- **Visual Paradigm**: Se trata de un software de generación de documentación UML, desde casos de uso hasta requisitos, pasando por modelos de flujo o de actividad. Se ha utilizado para el modelado de todos los diagramas presentes en el documento y se eligió por su simplicidad y el conocimiento previo que se tenía sobre él (usado en asignaturas de Ingeniería del Software).
- **Bibtex4word**: Más que una herramienta de trabajo, este elemento nos ha servido para la creación de una bibliografía uniforme, gracias a la utilización de ficheros en Latex para su estandarización y a la base de datos que proporcionan éstos.

Capítulo 2

Motivación y Objetivos

El proyecto en sí va a tener una doble finalidad que se explicarán a continuación por separado. La motivación y objetivo principal del proyecto es la creación de un sistema en el lenguaje de programación SystemC sobre el que realizar pruebas a partir del código C de un codificador de video.

Por otro lado, vamos a tener un segundo objetivo que será el aprendizaje de este nuevo lenguaje y su forma de trabajar, tanto a la hora de crear el sistema como de realizar diversas pruebas sobre él mismo.

2.1.- Implementación en SystemC de un modelo basado en un codificador H.264/MPEG-4 en C.

El objetivo principal del proyecto será realizar la implementación de este sistema. Para ello contaremos con todos los mecanismos y estructuras que nos proporcione el lenguaje SystemC.

Tratará de conseguirse un sistema de codificación de video H.264/MPEG-4 en lenguaje SystemC funcionando al mismo nivel que el original en C. Pero además, se utilizará este modelo para realizar unas sencillas pruebas de las que saldrán unos resultados que permitan mejorar el sistema o reparar, en caso de ser necesario, algún problema de comportamiento.

Lo que se pretende codificar no es algo tan extraño o prototípico que no tenga una utilización real en un mundo externo al de investigación. Es directamente un elemento útil si se completa la implementación. Por lo que el objetivo es crear algo realmente servicial y no sólo un modelo de aprendizaje y trabajo puramente académico-teórico.

Este software fue elegido porque parecía el más sencillo de implementar dentro de los que proporciona la propia universidad, que llega a tener varias versiones posteriores. Además es necesario que tenga un buen fundamento teórico, ya que si se hubiera podido empezar de cero con cualquier proyecto, no tendríamos la complejidad necesaria para demostrar que realmente SystemC tiene unas ventajas en la cadena de desarrollo de Software y del co-diseño HW/SW.

Esa es la motivación que hubo a la hora de elegir el modelo original: sencillez de código pero complejidad en la base. Así se parte de un código bien estructurado para adaptarlo a SystemC, pero sin caer en el error de tomar algo sencillo que no nos aporte conocimiento ni experiencia real en el manejo del lenguaje.

El objetivo es claro, generar un modelo SystemC que permita probar el módulo de codificación de vídeo utilizando código C ajeno al desarrollador.

2.2.- Aprendizaje de SystemC y sus mecanismos

Lo realmente interesante del proyecto, sin desmerecer el sistema creado con sus muchas horas de trabajo, es el aprendizaje de este nuevo lenguaje para el futuro trabajo en una empresa del sector.

Como ya se explicó en la introducción, el lenguaje SystemC provee de unos beneficios a la hora de realizar pruebas de comportamiento que podrían llegar a representar una ventaja para la empresa que los utiliza frente a la empresa que no. Por tanto es realmente muy interesante el aprendizaje de este lenguaje.

Con esto en mente, se eligió también el proyecto original. Como ya se explicó en el apartado anterior, la complejidad es suficiente para entrenarse en el manejo del lenguaje de forma introductoria.

En cuanto a los manuales que fueron utilizados, reflejados en la biografía, el más importante ha sido la obra "SystemC: From the Ground Up", disponible en la biblioteca de la facultad. Ha sido el libro de referencia principal utilizado a lo largo de todo el trabajo.

Los objetivos de este apartado son también bastante claros:

1. Aprender todo lo posible sobre SystemC como lenguaje.
2. Conocer cómo se trabaja en SystemC, la forma de separar los elementos, de comunicación…
3. Resolver cualquier problema que se nos pueda plantear tanto de programación como de infraestructura: soporte de Eclipse para SystemC, de las librerías en la distribución Linux…
4. Preparar el documento de presentación de la forma más técnica posible.

Capítulo 3

SystemC

Este capítulo estará dedicado a instruir al lector en el lenguaje de programación SystemC. Tratará de darse una visión completamente objetiva y precisa de qué es exactamente, cómo funciona y de qué manera se debe trabajar con él, qué finalidad tiene y qué hay tanto en este momento como en el futuro de este lenguaje.

3.1.- ¿Qué es SystemC? [1]

SystemC es un lenguaje de diseño y descripción de sistemas basado en librerías C/C++, surgido a partir de la necesidad de un lenguaje que permita una mejora de la productividad de los ingenieros en el diseño de sistemas electrónicos complejos.

Estas librerías características de SystemC añaden una dimensión diferente a la hora de trabajar con sistemas de co-diseño Hardware y Software:

- Noción de tiempo: se admite una sincronización temporal entre los elementos del sistema a nivel de comunicación o espera en tiempo real.
- Concurrencia: permite una mayor exactitud a la hora de dividir el trabajo en hilos independientes pertenecientes a cada elemento del sistema que se va a crear.
- Tipos de datos especiales: el programador de este lenguaje podrá hacer uso de los tipos de datos más comunes a nivel de diseño Hardware.
- Jerarquía modular: absolutamente todo tendrá que estar incluido en un hilo, encapsulado a la vez en un módulo. Y cada módulo estará conectado con los demás mediante puertos de entrada y salida.

Debidas a la gran cantidad de modelos de computación y comunicación que se pueden encontrar hoy en día, a los niveles de abstracción que tienen los elementos y a las metodologías de diseño de sistemas existentes en el mundo, existe un reto importante a la hora de diseñar un lenguaje como este. Unos sistemas están diseñados a través de un flujo de datos, otros a partir de una máquina de estados finita… SystemC trata de paliar este tipo de desajustes a la hora de comunicar subsistemas tan heterogéneos y además resuelve el problema de simularlos.

Para acabar con este problema, SystemC añade abstracción por capas, lo que permite al programador incluir nuevos elementos que compartan con los anteriores un motor de simulación eficiente (*Figura 9*). La primera capa incluye un núcleo de simulación que controla los eventos de forma abstracta. Que quiere decir que maneja los eventos y realiza los cambios de contexto sobre los procesos sin saber realmente qué representan esos procesos (sin una discriminación sobre el tipo de proceso o contenido).

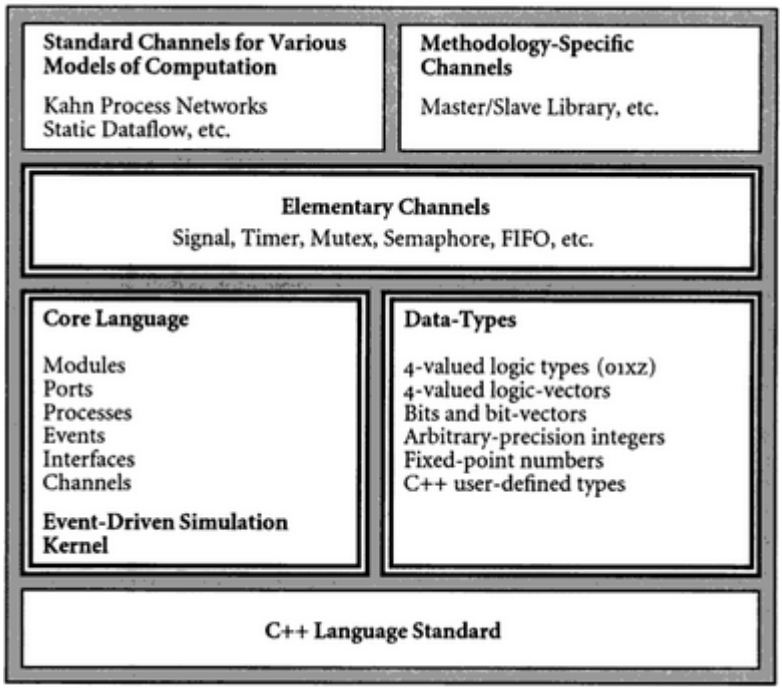

Figura 9: Estructura de la arquitectura SystemC

Además de este simulador, se incluyen módulos, puertos, procesos, eventos, interfaces y canales de comunicación como elementos abstractos. Todos estos elementos forman el núcleo del lenguaje, la parte principal.

Como se aprecia en la figura, tendremos también los tipos de datos pertenecientes a SystemC que hacen referencia a elementos del diseño Hardware de sistemas, vectores lógicos, bits,… además de los definidos por C++ y el propio usuario.

Por encima de estas dos capas se encuentra la de comunicación como tal, en la que se incluyen elementos temporales y de sincronización propios de un lenguaje con soporte de hilos, como pueda ser el mismo C: FIFOs, señales, timers, mutex, semáforo…

Con estas tres capas ya tenemos un lenguaje lo bastante potente como para modelar por sí mismo y simular diferentes elementos del sistema. Sin embargo, podemos apreciar en la figura que existen otras 3 capas, que serán igual de importantes.

La primera de ellas es el lenguaje estándar C++. Eso quiere decir que cualquier sistema que programemos en SystemC es compilable con un compilador C++.

Por encima de estas capas se encuentran los canales que suponen una comunicación estandarizada típica entre elementos del sistema. Resultarán útiles a la hora de implementar un sistema si ya tenemos los canales de comunicación más comunes y no hay necesidad de crearlos.

La principal motivación que llevó a diseñar SystemC fue la productividad de los ingenieros a la hora de diseñar un sistema electrónico moderno en los que la complejidad va en aumento conforme pasa el tiempo. Sin lenguajes del estilo de SystemC, con la complejidad añadida de cada nuevo proyecto, se volvería extremadamente complejo el manejo de estos nuevos sistemas, lo que ralentizaría el avance tecnológico que estos suponen. Y ese avance tecnológico es el verdadero objetivo de todos estos lenguajes y proyectos.

3.2.- Complejidad del sistema

Como ya se ha tratado en el apartado anterior, el principal problema que trata de resolver SystemC es el aumento gradual de complejidad en los sistemas electrónicos modernos.

En esta sección se pretende dar una explicación de cómo el lenguaje hace frente a estos problemas. Y se van a tratar soluciones genéricas primero: abstracción, reutilización y automatización.

La abstracción supone el método más común para aislar los elementos más complejos en capas subyacentes y poder manejarlo desde elementos superiores más sencillos. Esto ha sido utilizado durante mucho tiempo, y es la principal herramienta con la que contamos también en SystemC para llevar a cabo ese avance al que se hacía referencia antes.

En este sentido tenemos dos enfoques diferentes que han evolucionado de forma muy diferente. Por un lado está el enfoque Software en el que se ha pasado de la programación en lenguajes ensamblador, a lenguajes de alto nivel (C, FORTRAN,…), lenguajes cada vez más abstractos (Lisp, C++,…), lenguajes orientados a objetos (Java), programación genérica, excepciones…

Y por otro lado, el enfoque del modelado Hardware que comenzó con la definición de polígonos, avanzó en el diseño a nivel de puerta lógica como tal y se estancó en los años 80 con Verilog y VHDL, que no permiten una programación más allá de la que obliga a hacerlo ciclo a ciclo en cada flanco de reloj.

Como resultado, se puede ver que la comunidad de desarrollo Hardware no ha tenido un avance tan continuado en el campo de la abstracción como lo ha hecho la comunidad Software. Es por ello que los primeros se centrarán más en los otros dos puntos para tratar la complejidad.

La reutilización es una técnica utilizada constantemente en productos de ambos enfoques en la actualidad. Supone una importante reducción de la complejidad en un sistema si estamos reutilizando algo ya probado y funcional. Es posible hacerlo tanto a nivel de diseño como a nivel de implementación, también puede utilizarse con el diseño original o modificarlo ligeramente, incluso con un código reutilizado cambiar el nivel de abstracción, por ejemplo, dividiéndolo en funciones más sencillas, separando funcionalidad…

El tercer y último método genérico antes de comenzar a explicar los propios de SystemC consiste en la automatización de los procesos. Este es el punto más importante en el diseño Hardware, ya que es el que más ha sido explotado por los programadores. Es debido a una serie de mejoras que el proveedor de herramientas de diseño ofrece a la comunidad de desarrollo, mediante mejoras automatizadas como pueden ser la generación automática de estructuras de lenguaje (declaración de módulos).

SystemC tiene sus mecanismos y oportunidades basadas en estos principios generales para evadirse del aumento de complejidad de los sistemas modernos.

Como primera medida SystemC dedica su mayor esfuerzo en permitir al usuario aumentar el nivel de abstracción sobre los elementos tanto como desee. Esta es la herramienta más importante como ya se ha especificado antes.

En cuanto a la reutilización, se puede decir que SystemC tiene bastante suerte, porque puede reutilizar todos los modelos e implementaciones creados previamente por los programadores de C++ (30 años de vida).

Además de esto, SystemC cuenta con el apoyo de más de una docena de empresas fuertes en el sector, lo que le proporciona un feedback y una cantidad de desarrollo y evolución importante.

Estas dos últimas características concretas son las que hacen de SystemC un lenguaje emergente y sólido frente a otros de similar finalidad.

3.3.- Elementos del sistema

En esta subsección se pretende dar una pequeña explicación de qué es realmente todo lo que se hace referencia cuando se habla de módulos, procesos y puertos en SystemC.

3.3.1.- Módulos

Un módulo es una unidad básica de construcción en que se divide el sistema. Permite dividir el trabajo en diferentes partes cumpliendo exigencias del análisis o a un reparto de funcionalidad. Esto permite romper un sistema complejo en varios más sencillos, es el mismo concepto que la modularidad de un lenguaje de alto nivel, por lo que nos proporcionará las mismas ventajas: fácil interpretación del sistema, sencillo a la hora de buscar errores o realizar modificaciones sobre partes específicas, es posible intercambiar módulos con una funcionalidad por otros…

Un módulo típico contendrá o podrá contener los siguientes elementos:

- Puertos: tanto de entrada como de salida, para realizar la comunicación con los demás módulos. Es muy extraño el caso de módulos con una funcionalidad propia y sin comunicación con el exterior.
- Procesos: de los que se hablará más adelante también y que definirán esa funcionalidad que tendrá el módulo. Puede tener 1 solo si es un módulo sencillo, o varios si el módulo se encarga de más de una tarea simultánea.
- Variables locales y de control: como en cualquier módulo en un lenguaje de programación de alto nivel, podrá haber variables locales que tendrán visibilidad desde todos los procesos. Así mismo existirán unas variables locales especiales que harán las veces de control para elementos como máquinas de estados finitas.
- Jerarquía: Dentro de un módulo pueden estar instanciados otros módulos con el fin de generar una dependencia y una estructura más compleja.

Para definir un módulo existe una estructura más o menos fija que se muestra en la *figura 10*. En ella se puede ver que existe un espacio bien definido para cada elemento mencionado anteriormente, aunque son sólo de declaración pues los procesos, por ejemplo, se deben crear como funciones C++.

```
SC_MODULE(Adder) {
    // ports, processes, internal data, etc.
    SC_CTOR(Adder) {
        // body of constructor:
        // process declaration, sensitivities, etc.
    }
};
```

Figura 10: Definición estándar de un módulo

3.3.2.- Procesos

Estos elementos no difieren demasiado del término clásico de proceso en un lenguaje de alto nivel. Se trata de funcionalidad implementada en un hilo que se ejecuta en un módulo. Comparándolo con un hilo normal de un código C que podamos crear, no existe mayor diferencia que el hecho de que estos procesos no forman parte de un SO, sino que representan la funcionalidad que tendrá el elemento Hardware definido. No tienen un PID, ni se generan en el sistema a partir de un proceso padre. Sin embargo, como los hilos C, se ejecutan en paralelo en cierto modo.

Estos procesos serán privados de cada módulo, esto quiere decir que tendrán acceso a las variables locales de módulo, y comunicación con el resto de procesos del mismo, pero no podrán ver ni a procesos de otros módulos ni comunicarse con ellos.

La comunicación dentro del mismo módulo de los procesos es posible mediante unas estructuras especiales. Estas estructuras llamadas sc_events (eventos) van a permitir que un proceso espere a otro dentro del módulo, permitiendo sincronizar perfectamente todos los hilos de una manera rápida y sencilla. Simplemente se debe especificar que un proceso es sensible a unos eventos concretos a la hora de declararlo, y cada vez que un proceso del mismo módulo invoque ese evento el sensible comienza a ejecutarse donde se quedó a la escucha.

Con esta buena comunicación que se puede generar entre los procesos, se tiene una herramienta bastante potente para el diseño de módulos complejos multiproceso.

3.3.3.- Puertos, canales y señales.

Como ya se ha mencionado en este epígrafe la comunicación entre los distintos módulos es fundamental, por lo que el tema de puertos de entrada y salida en los elementos que conformen el sistema será bastante importante. En esta subsección se pretende dar una idea general de cómo se utilizan al lector para que pueda entender mejor el desarrollo del proyecto.

La comunicación entre los diferentes módulos se realiza a través de puertos. Estos se convierten en puntos de comunicación externa del módulo y establecer un canal a partir de él. Estos puertos son interfaces de tipo abstracto que necesitan un tipo de dato con el que trabajar. Una vez se establece ese tipo de dato, tendrá también que proveer al sistema de los métodos virtuales necesarios para manejarlos. Estos métodos están prestablecidos para los tipos de datos habituales (enteros, caracteres,…).

Además de estos puertos, la clase manejadora del sistema (la que tiene asignado el método main y arranca todo el sistema), deberá crear canales de comunicación entre cada puerto de cada módulo, estableciéndolos por pares. Estos canales también deben tener un tipo y será del mismo tipo que el puerto, o lógicamente no funcionará.

Cuando se realiza una llamada de tipo write() sobre un puerto conectado a un canal de tipo fifo sucede lo que se muestra en la *figura 11*, el manejador de la clase sc_fifo tomará el dato y lo almacenará hasta que quien esté conectado al otro lado del canal decida realizar una llamada de tipo read().

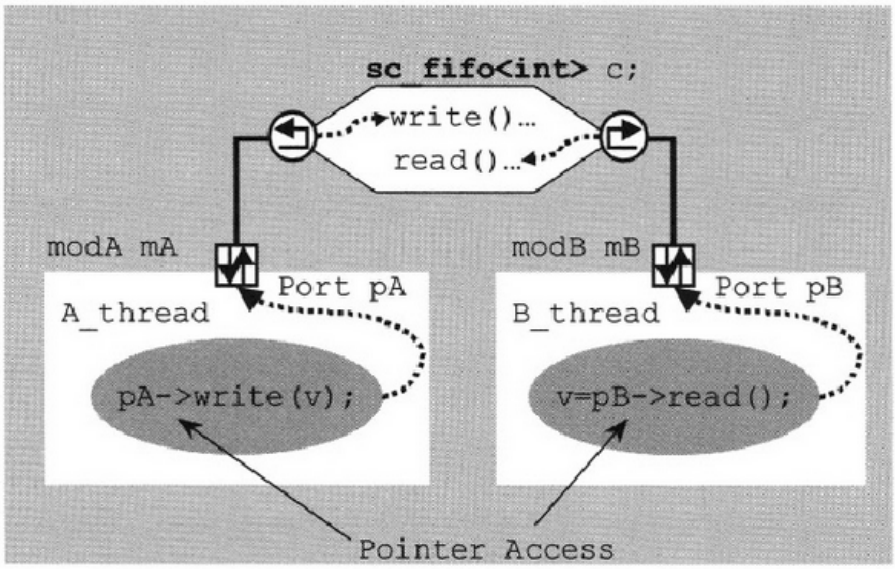

Figura 11: Comunicación mediante sc_fifo

3.4.- Comparativa con otros lenguajes

SystemC no representa un lenguaje en sí mismo, ya que se trata de una serie de librerías dentro de un lenguaje formado como es C++. Sin embargo aporta herramientas externas (del propio equipo de desarrollo de SystemC) que permiten trabajar de forma eficaz. Un ejemplo de estas herramientas es el SCV, que nos permite generar conjuntos de prueba para nuestro sistema de forma controlada, con lo que poder verificar las salidas y los valores intermedios del sistema de forma sencilla y rápida. También nos permite utilizar un lenguaje tan común como C++ en el desarrollo tanto de software como de hardware, lo que le da cierto valor añadido por los mecanismos de validación y verificación propios.

Existen lenguajes que resuelven problemas para paliar la complejidad y realizar el diseño concreto de una parte del sistema en el mercado, por ejemplo: C/C++ predominan en el diseño de software embebido, lenguajes de definición de hardware (HDL) como Verilog o VHDL están utilizándose en simulación y sintetizado de elementos hardware, Vera es un lenguaje muy utilizado a la hora de verificar circuitos integrados de tipo ASIC y Matlab y otros lenguajes del estilo son utilizados para capturar los requisitos de los sistemas.

En la *figura 12* se puede apreciar cómo SystemC es un lenguaje más completo que cualquiera de los mencionados anteriormente, y que además podría sustituir el uso de varios de ellos a la vez. Ya que llega desde el nivel RTL hasta el de requisitos.

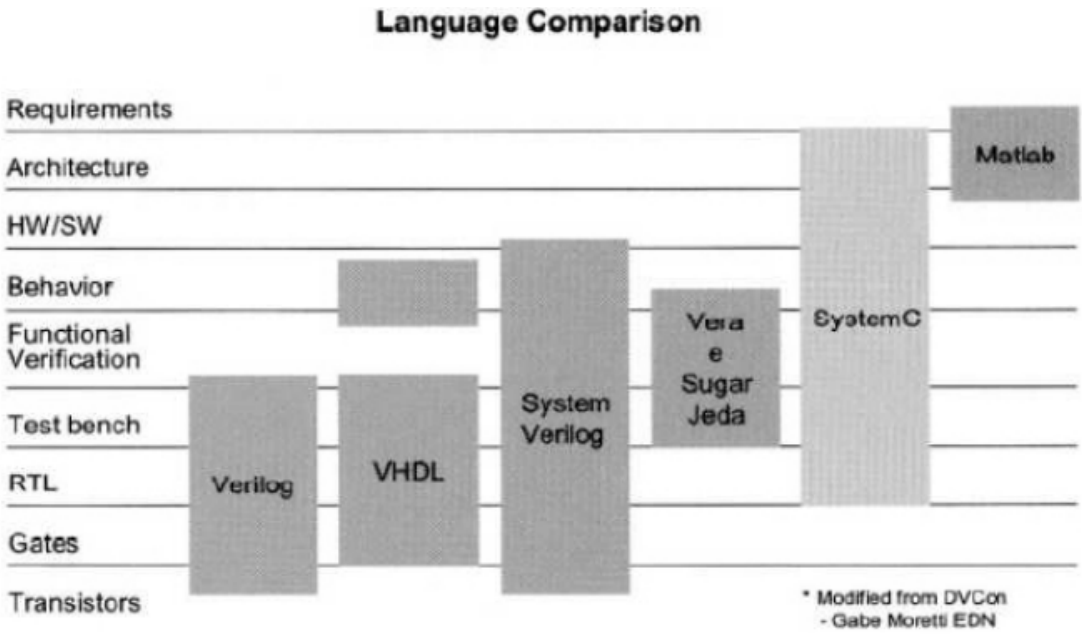

Figura 12: Comparativa de uso de lenguajes

Capítulo 4

Estado del arte

Este capítulo hará una presentación al lector sobre el estado en que se encuentran las herramientas de codiseño, así como su evolución desde sus inicios hasta las tecnologías actuales, las principales compañías volcadas en este asunto y un análisis final sobre lo económicamente viable que resulta la utilización de este tipo de herramientas.

También se expondrá el estado de la codificación de audio y video.

4.1.- SystemC

SystemC es un lenguaje concebido como un conjunto de librerías y macros de C++ con la finalidad de permitir una simulación completa de elementos hardware, mezclando la simulación de estos elementos con la de código en alto nivel, principalmente su lenguaje primario: C++.

Las posibilidades posteriores que ofrece SystemC como motor simulador de elementos hardware respecto a otros como puedan serlo VHDL o verilog, lenguajes puramente de descripción como tal de hardware, es la posibilidad de introducir código C/C++ añadiéndole la profundidad temporal propia de estos elementos.

En la actualidad, el lenguaje SystemC no sólo termina en la simulación de elementos como el que se presenta en este documento: elementos sencillos que pretenden funcionar por sí mismos con respecto a un computador sobrentendido. Si no que además, se pueden simular elementos tan complejos como los propios procesadores. Incluso se puede conseguir cierta heterogeneidad entre los elementos, dentro tanto del mundo de la electrónica como de otras disciplinas físicas. Como es el caso del proyecto europeo “Beyond Dreams” que pretende incluir otros ámbitos al electrónico, como puedan ser el mecánico, óptico…

Más allá de este tipo de intrusión en otros ámbitos para utilizar SystemC como simulador heterogéneo, existe una segunda versión del lenguaje que nos permite la simulación de sistemas analógicos de señal mixta. Esto son circuitos tanto con parte analógica como con parte digital.

4.2.- Compresión de audio y video

Cuando se menciona la codificación de audio y video en este proyecto, se hace referencia a la tarea de reducción del peso de un fichero de video en un sistema software. Esto se lleva a cabo reduciendo la cantidad de información que se tiene de cada imagen. Lo

que nos aporta beneficios más allá del mero ahorro de memoria, sino que además nos va a permitir el envío menos costoso de ese video a través de la red, por ejemplo. Es decir, de una disminución del ancho de banda necesario para la comunicación.

Hoy en día este punto es importante, debido a la cantidad de servicios de video en streaming existentes, cuyo caso más agudo se encuentra en Youtube. Además, otro factor que impulsa el crecimiento de la tecnología de compresión es el hecho de que los sistemas de captura de audio y video es cada día mejor, es decir que tiene mayor resolución, lo que se traduce en un aumento de la cantidad de información que se almacenan en los vídeos.

A pesar de todo esto, el origen de la codificación no es tan novedoso, aunque sí bastante rápido. Se remonta al año 1948 con la aparición de la codificación Shannon-Fano, y se consolida varios años después (1952) con la técnica de codificación Huffman. Ambas serán un referente desde entonces para todas las demás técnicas de codificación. Los diversos métodos de compresión han ido evolucionando con el tiempo hasta llegar a día de hoy, en que los métodos están basados en complejos algoritmos.

Capítulo 5

Captura y análisis de requisitos

Ésta será una de las etapas más importantes del proceso, ya que de ésta dependen todas las demás por lo que se intentará ser lo más concisos posible a la hora de definir estos requisitos. Es también importante porque, además de darnos una idea de qué es lo que tiene que hacer el sistema, va a controlar y limitar el esfuerzo que se dedicará a cada parte. Así los requisitos nos darán una visión general y una concisión sobre los objetivos del proyecto, pero además nos recordarán en qué partes hay que dedicar más tiempo y qué funcionalidad debe cumplir cada una. Se intentará no desatender ninguno de los requisitos, tanto funcionales como no funcionales.

5.1.- Requisitos funcionales

Los requisitos de tipo funcional son aquellos que son utilizados para referirse al comportamiento propio del sistema en una situación concreta o el que debe tener ante unas entradas determinadas. En este caso concreto, sólo habrá una única línea de acción, y será exclusivamente por parte del usuario. Así que se han encontrado requisitos funcionales de usuario únicamente, recoge una descripción de alto nivel de qué es lo que se pretende que haga el software que se va a desarrollar.

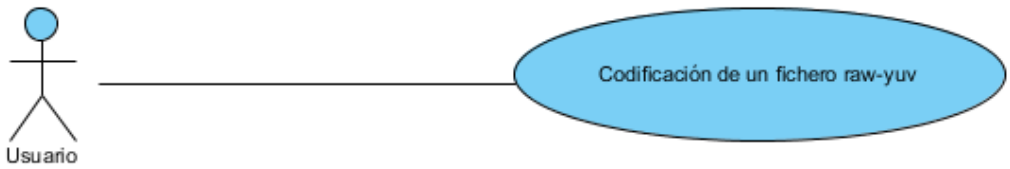

Figura 13: Diagrama de casos de uso de nuestro sistema

La *figura 13* muestra los casos de uso disponibles para el usuario que pretenda acceder al sistema. Es bastante sencillo, ya que se trata de un elemento de bajo nivel y con un objetivo muy claro. A partir de él se definen los requisitos funcionales:

Codificación del fichero RAW-YUV: El usuario podrá utilizar el sistema para codificar cualquier fichero en formato .yuv[4] que desee en uno que cumpla con el estándar H.264/MPEG-4 o, como es este caso, realizar pruebas de comportamiento y rendimiento en el código C/C++. Como el nivel de la aplicación es bajo, el usuario tendrá únicamente la opción de introducir los parámetros mediante comando en un terminal. El propio sistema será el encargado de comprobar que se ha introducido el comando correctamente y que el fichero que se pase sea válido.

5.2.- Requisitos no funcionales

Este tipo de requisito va a restringir al sistema a un nivel diferente al que lo hacen los funcionales. Son requisitos no dependientes tanto de la función que cumplen los algoritmos desarrollados sino con unos objetivos más abstractos pero igualmente concretos. Estos objetivos deberá cumplirlos el código desarrollado a nivel de sistema, y no como un comportamiento que deba tener. Así como los requisitos funcionales son prácticamente inexistentes en este sistema, no pasa lo mismo con los requisitos no funcionales. Necesitamos que el sistema funcione exactamente igual que el original escrito en C, es decir, que el requisito no funcional más importante sea mantener el comportamiento invariable.

Este es el más importante y a la vez más complejo a priori, ya que requiere coordinar correctamente todos los módulos en que vamos a dividir el sistema para que no se pierda ningún dato, o se trabaje con un "frame" de forma desordenada, lo que podría llevar a generar secuencias erróneas de video.

Además de este hay otros requisitos no funcionales en el sistema que se van a desarrollar. Primero se establecen la división en tipos de estos requisitos. (*Figura 14*)

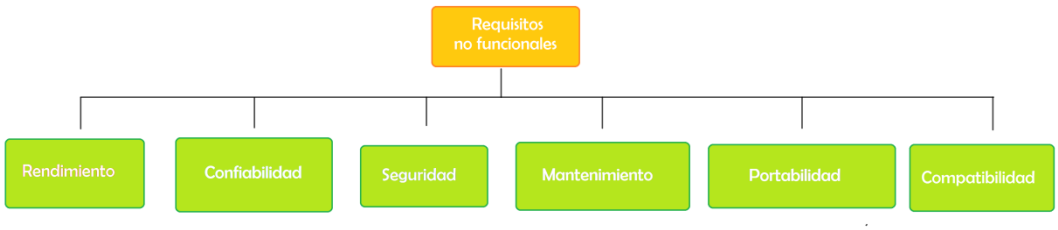

Figura 14: División de los requisitos no funcionales

Requisitos de eficiencia de rendimiento:

El tiempo de ejecución del sistema deberá ser igual al del código original más el tiempo añadido de la estructura SystemC. Es decir que queda establecido qué % de tiempo añade este sistema respecto el original. El tiempo va a ser el factor limitante del sistema, ya que será el más importante a la hora de hacer mejoras. No podremos medir el tiempo exacto del elemento final[4], pero podemos establecer relaciones de proporción.

Requisitos de compatibilidad:

Este tipo de requisito incide sobre la capacidad de comunicación mediante una interfaz conocida por ambos del sistema. Con este requisito lo que se pretende es que la comunicación entre los módulos que creemos sea estandarizada lo más rigurosamente posible. Así como la comunicación del sistema con el exterior, con el fin de facilitar el trabajo futuro. Para ello quedará constancia de toda la comunicación entre módulos y externa en la documentación del proyecto.

[4] Ya que nosotros realizamos las ejecuciones sobre un procesador Intel.

Requisitos de confiabilidad:

La madurez del sistema será total una vez haya concluido el proyecto. Esto quiere decir que no se podrán producir errores en el sistema como resultado de una mala ejecución de la parte software. Esto le proporciona un nivel de robustez inicial considerable, ya que no existirán problemas en cuanto al funcionamiento interno del sistema.

Requisitos de seguridad:

No se establece ningún criterio de seguridad en la aplicación, ya que el proyecto que se pretende realizar no requiere ningún tipo de autentificación del usuario ni de almacenado de datos que puedan estar sujetos a la Ley de Protección de Datos. La única seguridad que podría tenerse en cuenta sería la privacidad de los videos que se codifican, pero al generar un sistema de pruebas que posteriormente se convertirá en un sistema hardware no hay posibilidad, en principio, de que los datos sean robados o modificados desde el exterior.

Requisitos de mantenimiento:

Este apartado también es importante, ya que se está creando un sistema que se va a utilizar para realizar diferentes pruebas sobre él para intentar mejorar en el futuro su funcionamiento o reutilización, es necesario que cumpla ciertos requisitos de mantenimiento.

Por una parte habrá que potenciar la modularidad, para permitir que en el futuro cualquier desarrollador que trabaje sobre el código desarrollado, o cualquier persona ajena, pueda comprenderlo y navegar por él de la manera más intuitiva posible.

La analizabilidad será obligatoria al buscar optimizar el uso de recursos del sistema en el futuro, ya que a través de este requisito se asegurará que se pueda identificar cada elemento y artefacto del sistema, así como la posibilidad de realizar un análisis en busca de errores y estadísticos.

Como ya se ha dicho, la posibilidad de una mejora futura del sistema, crea unos fuertes requisitos de mantenimiento. A los que además habrá que añadir que sea sencillo el método de realizar pruebas e intuitiva la manera en que se pueda modificar.

Con este grupo de requisitos se pretende que las revisiones futuras sean lo más cómodas posibles. Esto queda asegurado mediante una documentación muy detallada y claridad en el código, por ejemplo en el nombre de variables y funciones, una forma de programación lo más estandarizada posible…

Requisitos de portabilidad:

No existen requisitos de portabilidad ya que el sistema se basa en las pruebas para un elemento externo. Cualquiera puede utilizar el código siempre que tenga un compilador de C/C++ y las librerías de SystemC, independientemente del S.O. que utilice. Así que ese sería el único requisito, pero hoy en día cualquier equipo puede disponer de todo esto y está preparado para utilizarlo.

Capítulo 6

Diseño

En este apartado se va a explicar el proceso de Diseño del sistema como se muestra en la introducción. Primero se tomará el código C y será generado un diagrama de clases y un pequeño análisis del funcionamiento del código. Posteriormente se mostrará la forma en que se ha modelado el sistema bajo el lenguaje SystemC, en qué partes quedará dividido y cómo se reorganizarán las variables globales que pudiera tener el código C para que funcione conforme la división de los módulos. Después se explicará cuál es la tarea asignada a cada parte del código original. Y por último las entradas y salidas de los módulos y del sistema final.

6.1.- Diseño del modelo original en C

Comenzamos este proyecto descargando el código[5] correspondiente a la implementación JM8 del estándar H.264/MPEG-4. Con este código realizamos el diagrama de clases de la *figura 15*. Para realizar el proyecto no tenemos por qué tener conocimiento concreto de qué hace cada función, pero si un esbozo de cómo está funcionando el código, por ejemplo quien es la clase principal, donde se encuentran las declaraciones importantes…

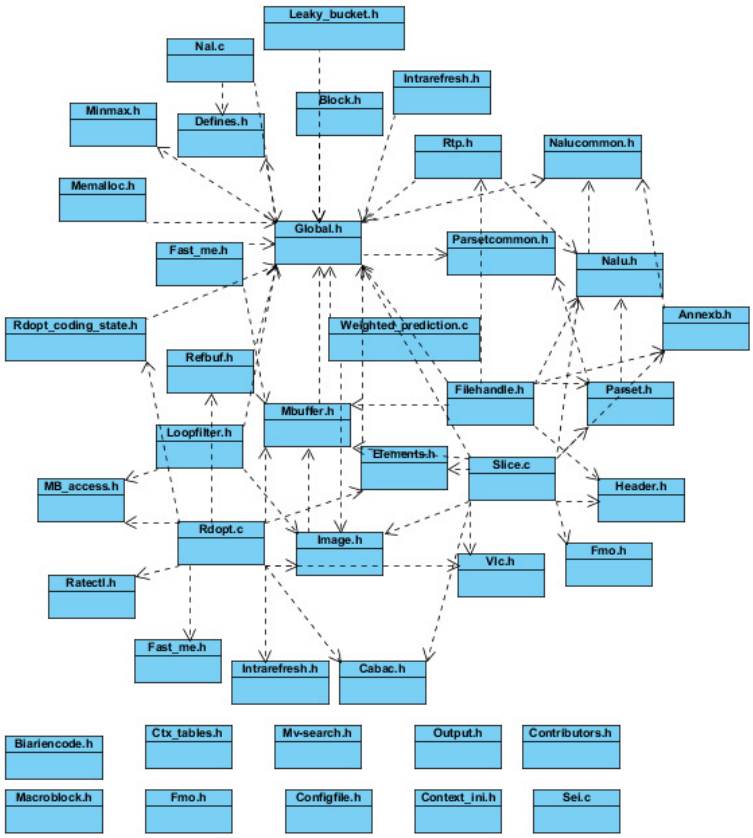

Figura 15: Diagrama de clases del código C

[5] http://iphome.hhi.de/suehring/tml/download/old_jm/

Como se puede ver el sistema está bastante enmarañado, teniendo relaciones de doble sentido en varios casos. Podemos apreciar cómo “global.h” es el módulo central sobre el que trabajan todos. Echando un rápido vistazo a su contenido vemos que se trata de una cabecera de declaración tanto de variables globales como de tipos de estructuras. También tenemos una cabecera global que nos muestra todas las constantes del sistema: “define.h”.

Sobre este diagrama que hemos mostrado tenemos una clase llamada “lencod.c” que controla todo el proceso de codificación. Tiene una función main que va llamando en orden a todas las demás de los diferentes archivos. Para facilitar la comprensión, se ha mostrado solo las conexiones del “lencod.c” en la *figura 16* como un módulo sobre del sistema.

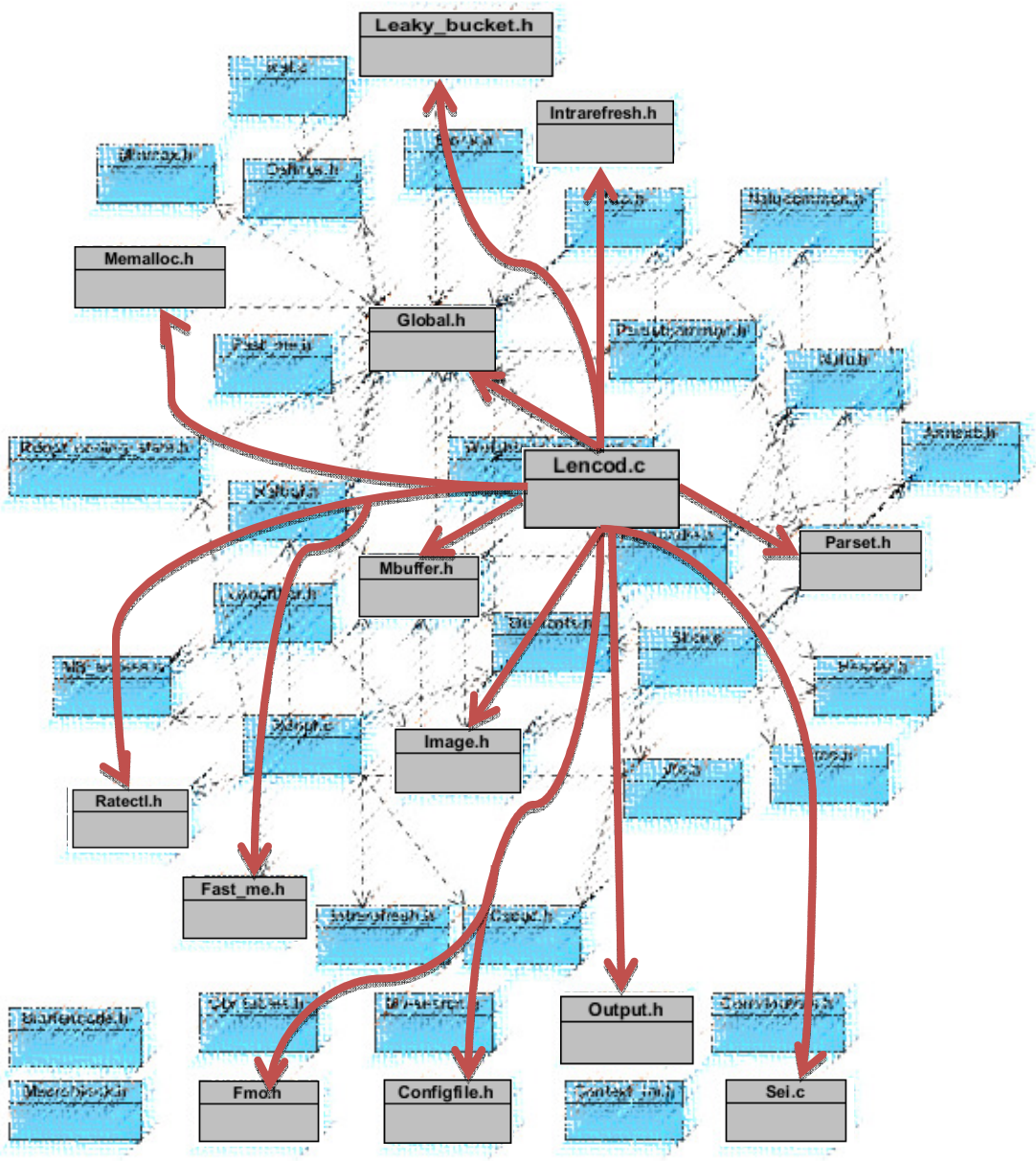

Figura 16: Conexiones del módulo “lencod.c”

Dentro del código del fichero principal ("lencod.c") se encuentran las funciones auxiliares a la función main. Toda la funcionalidad que más adelante repartiremos entre los módulos que creemos estará sacada de esta función main. Podemos apreciar 4 partes en esta implementación, una primera en la que se calcula si el fichero que hemos introducido por parámetro es correcto, una segunda parte en que se inicializan todas las variables, los buffers…, una tercera parte en la que se realiza un bucle que parece codificar cada frame, y una cuarta y última parte en que se destruyen todos los espacios de memoria reservados, se realiza un reporte de errores y se termina el programa.

6.2.- Diseño del modelo SystemC

En esta sección se va a desarrollar el diseño del modelo SystemC que se va a implementar. Primero se hará una división en módulos de todo el código C, intentando desglosarlo en función de las diferentes funcionalidades que haya implementadas. Este paso es el más importante, ya que si se consigue un diseño funcional y sencillo, se podrá adoptar en un futuro una postura de trabajo más orientada a la forma modular de trabajo de alto nivel, lo que permitirá una optimización más sencilla y menos costosa en el sistema.

En una segunda parte, se obtendrán las variables globales al sistema C, para trabajar sobre ellas y conseguir que la visibilidad de cada módulo esté limitada al uso real de las variables. Esto es importante, porque en un sistema hardware final no existen las variables globales como tal. Se prescindirán las variables de configuración física estándar: las especificaciones del tipo de video que se codifica se supone un valor que todos deben conocer una vez se implementan físicamente.

Como última parte se establecerán las entradas y salidas de cada módulo.

6.2.1.- División en módulos del sistema

El primer paso para comenzar con la definición del sistema será establecer qué datos van a ser importantes en el proceso y de qué manera de está dividiendo el trabajo. De esta manera tendremos un modelo jerarquizado del sistema con entradas y salidas entre los módulos.

Siguiendo el análisis que se hizo sobre el código C, se puede apreciar que existe una parte bien clara y diferenciada de codificación. Este será el que forme nuestro módulo principal. Con esto en mente, lo que nos queda en el código es meramente líneas de configuración del sistema, y eso será trabajo del SO al que se adhiera nuestro codificador. Por lo tanto vamos a tener dos módulos inicialmente (*figura 17*), uno que realice la codificación como tal y otro que realice todo el proceso de inicio de variables, lectura de ficheros, envío de las imágenes a codificar,…

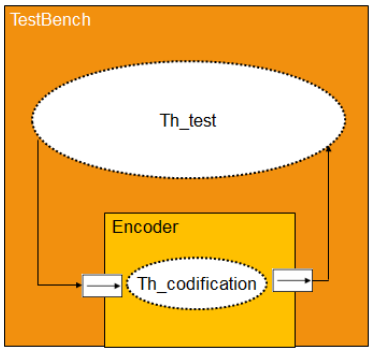

Figura 17: Diseño del sistema que desarrollaremos en SystemC

6.2.1.1.- Reparto de funcionalidad entre los módulos

Como ya hemos dicho vamos a tener que dividir nuestro sistema en base a la funcionalidad que tendrá cada parte. Esto nos permitirá aumentar el paralelismo, con lo que ganaremos tiempo frente a una implementación software tradicional. El beneficio se verá incrementado en nuestra futura implementación hardware.

Siguiendo con el apartado anterior, tendremos que dividir el sistema en 2 módulos. El primero será el encargado de emular el comportamiento del sistema sobre el que estableceremos nuestro codificador, y el segundo será el que tenga el proceso de codificación como tal.

De esta manera, y como ya se ha explicado previamente, dispondremos de un módulo de Testbench, que será eliminado en la construcción física del modelo, y un módulo Encoder, que tendrá lo estrictamente necesario para el trabajo de codificación.

Así que, acorde a la *tabla 2* tendremos un módulo que se llevará tres cuartas partes del trabajo y otro más pequeño a simple vista que solamente se encargará del trabajo del bucle de codificación. Esto es así, porque el módulo que se lleva más funcionalidad representará al SO en la futura implementación física, por lo que tampoco vamos a tenerlo en cuenta a la hora de realizar las pruebas. Nos centraremos en el pequeño módulo de codificación.

Que el módulo de Testbench tenga mayor número de tareas no influye en que sea más lento a la hora de ejecutarse. Al contrario, es muy probable que tarde menos en realizarlas todas que el módulo Encoder en codificar todas las imágenes.

Testbench	Encoder
Comprueba la validez del fichero de entrada	Codifica la imagen
Inicializa los buffers y las variables globales	
Envía datos por el puerto correspondiente a las variables de configuración	
Escribe todos los datos en el fichero de salida	
Elimina los elementos de memoria	
Devuelve el control al usuario	

Tabla 2: Reparto del trabajo en los módulos

6.2.1.2.- Reorganización de variables globales

Como es lógico un sistema hardware es susceptible de tener variables globales (a pesar de ser una mala práctica de programación), así que como trabajo inicial hay que discriminarlas del código e intentar transformarlas para que sólo tengan visibilidad en el módulo que las utiliza (que será cómo funcione el sistema real).

Siguiendo con el apartado anterior de la memoria, tendremos dos módulos: Encoder y TestBench. En el primero irá toda la funcionalidad perteneciente al bucle de codificación, y en el segundo la inicialización y posterior destrucción de variables, así como la escritura en el fichero de salida del proceso. Ahora sólo hay que encontrar las variables globales que se están utilizando entre uno y otro módulo y transformarlas hasta que sean locales de cada módulo y cada clase.

Se ha optado por trabajar de forma sistemática para eliminar las variables globales. Por lo que, cada variable global que haya tendrá que propagarse en todo el código y funciones que lo usen, añadiéndolo como atributo.

Después de analizar el código, se han encontrado 4 variables globales: input, image, snr y stat:

- **Input**: Es la estructura que almacenará toda la información global del proceso de codificación, ya sea el tipo, el tiempo, el número de frames totales…
- **Image**: Contendrá un puntero a una lista de imágenes que forman el video .yuv.
- **Snr**: Recoge la información perteneciente a la cantidad de ruido en la señal o imagen que haya leído.
- **Stat**: En esta estructura se guardará la información del estado del proceso de codificación en todo momento.

Estas cuatro variables aparecen a lo largo del código en todas las secciones así que lo que se ha resuelto hacer es, de forma sistemática, cambiar simplemente la declaración de estas variables al código perteneciente a cada módulo donde se utilice.

Para ello, se comienza por comprobar a qué módulo pertenece cada una de ellas. Con una revisión en profundidad se encontró que todas las variables se utilizan tanto en un módulo como en el otro, así que hubo que declararlas de forma global en ambos módulos.

Sin embargo, la principal diferencia entre uno y otro módulo, es la forma en que se tratan estas cuatro variables. En el primero se están escribiendo y estableciendo sus valores, y en el segundo se leen y se modifican solamente la de tipo image. Habrá entonces una comunicación completa de los datos del TestBench al Encoder, pero no es necesario que el Encoder devuelva nada más que las imágenes codificadas, ya que lo demás permanecerá inalterable.

Para resolver esta situación se ha creado una variable local en cada módulo con la que trabajar, y que tomará valores en función de si es del fichero original, o si viene

del canal de comunicación. Además el código C tendrá que trabajar con este nuevo modelo. Comprobando que las funciones se utilizan indistintamente tanto de un módulo como del otro en la mayoría de ficheros .c, la solución más sencilla ha sido cambiar el uso de la variable global a parametrizar en cada función los puntos a las estructuras pertinentes.

6.2.2.- Entradas y salidas

Conforme las variables globales encontradas que hay que utilizar en la comunicación entre ambos módulos, se han generado unas entradas y salidas en ambos módulos.

En primer lugar habrá una cola de tipo FIFO para los datos de las imágenes (image) ya que es necesario mantenerlas en el tiempo, almacenadas hasta que sean procesadas. Existe una estructura que cumple con estos requisitos en SystemC: sc_fifo. Como el modelo SystemC realimenta el módulo Testbench con la imagen ya codificada, y este se encarga de aglutinar todas en un fichero, el sistema necesita dos colas en cada módulo: una de entrada y una de salida.

Además de esta primera cola FIFO, también será necesario que se propague más información de un módulo al otro. En todos los casos restantes serán datos de configuración, por lo que sólo se necesita un canal unidireccional entre el Testbench y el Encoder. Así, se comunicará información también del Input, el Snr y el Stat. Es muy importante que todo se haga antes de comenzar a lanzar las imágenes para ser procesadas, por lo que en el proceso de comunicación se enviará toda la información de configuración y acto seguido se empezará a ejecutar el bucle de lectura de datos, codificación y reenvío.

6.3.- Diagrama de flujo del sistema

El funcionamiento concreto del sistema se explica a continuación, a partir del diagrama de flujo de la *figura 18*. Según la funcionalidad repartida en el apartado 6.2.1.1, se establece que el módulo de TestBench va a realizar las tareas propias del handler del módulo Encoder. Estas tareas serán completamente software una vez se concluyera todo el proceso seguido en este proyecto, y pertenecientes al SO o software concreto de codificación (Si lo implementamos en un equipo más grande o embebido).

En un primer momento se ha establecido un usuario como principal agente que decide codificar una imagen. Existe una capa de comunicación con el módulo “main” que se utiliza como mediador entre el modelo SystemC y el usuario. Esta simplemente se encargará de crear los módulos y de comunicarlos. Por ese motivo se ha obviado su existencia y se ha comunicado directamente el usuario con el módulo Testbench (que sería quien hace de handler).

La funcionalidad implementada para el TestBench es completamente de gestión, primero va a realizar tareas de creación de buffers, reserva de memoria, asignación de variables de configuración, lectura del fichero fuente a codificar y su comprobación, extracción de los metadatos que necesita y división en unidades que pueda procesar. Una vez termina este proceso, se comienza por enviar los datos imprescindibles al módulo Encoder, que espera su configuración. Y éste, una vez configurado, queda a la espera de cualquier posible imagen a codificar. Cuando vaya enviando y recibiendo datos, el TestBench llevará la cuenta, y una vez terminado, lo escribirá todo en el fichero de

salida, eliminará toda reserva de memoria y buffers, y ejecutará la función de reporte. Acto seguido morirá el hilo.

El Encoder está preparado con otra mentalidad. Como el paso siguiente al proyecto sería implementarlo en un elemento hardware que, como ya se ha dicho, nos proporcione una velocidad de codificación superior, está programado y preparado de una forma más orientada a la programación de hardware. Solamente se ha añadido el proceso de preconfiguración que espera del TestBench con una finalidad software, pero este proceso no existiría en el modelo final. El resto es un bucle infinito (continuo polling del elemento sobre el sistema) en petición de datos. Dentro de ese bucle introducimos la comunicación de recepción de una imagen, el proceso de codificación completo y el envío de la imagen codificada.

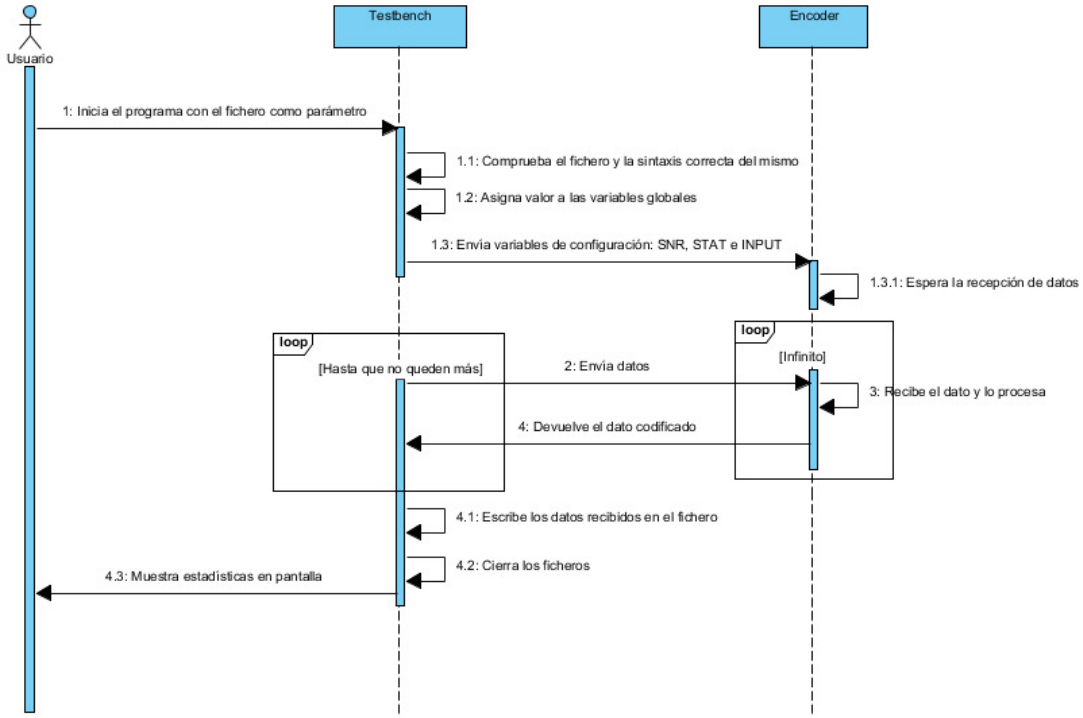

Figura 18: Diagrama de flujo del sistema

Capítulo 7

Implementación

En esta sección se va a dar una explicación lo más precisa posible sobre la implementación del sistema. En ella se explicará cómo se ha dividido el código en módulos de forma explícita, cómo hemos repartido las variables y todo lo argumentado en la sección de Diseño.

Además, y de forma paralela, según método de trabajo, hemos realizado pruebas a bajo y medio nivel, de tipo unitarias y de integración. Por lo que tendremos un pequeño apartado dedicado a explicar cómo las hemos hecho y qué resultados hemos tenido. Así mismo también hay un pequeño apartado donde se explican los problemas que más daño han hecho al periodo de desarrollo.

7.1.- Implementación en SystemC

Comenzamos con la creación de los módulos de test y codificación (Testbench y Encoder). Como ya se explicó en los primero capítulos, se han ido creando los módulos poco a poco, para permitir realizar pruebas de funcionamiento y aprender el máximo posible. Así pues el primer paso que se siguió fue el proceso de creación y compilación de ambos módulos. Sin incluir funcionalidad como se muestra en la *figura 19*.

```
#ifndef TESTBENCH_H_
#define TESTBENCH_H_

SC_MODULE(testBench){

};

#endif /* TESTBENCH_H_ */
```

```
#ifndef ENCODER_H_
#define ENCODER_H_

SC_MODULE(encoder){

};

#endif /* ENCODER_H_ */
```

Figura 19: Módulos sin funcionalidad ni conexiones

En este paso se aprovechó para crear un fichero main.cpp que recoja en un futuro ambos módulos, buscando un modelo de trabajo como el de la *figura 20*, que nos aloje los dos módulos y les permita y controle su comunicación.

```
#include <systemc.h>
#include "testBench.h"
#include "encoder.h"

int sc_main(int argc,char* argv[]){

        testBench testB("TESTBENCH");
        encoder encoder("ENCODER");

        sc_start(1000, SC_SEC);
        return 0;
}
```

Figura 20: Aplicación del main.cpp

El siguiente paso consistió en generar unas funciones de hilo en los módulos. Se decidió que cada módulo llevaría toda su funcionalidad recogida en un único hilo, así que se añadieron uno en cada uno con su correspondiente estructura de definición para que se arranque al definir el módulo en main.cpp (*Figura 21*)

```
SC_MODULE(testBench){
        void th_test();

        SC_CTOR(testBench){
                SC_THREAD(th_test);
        }
};
```

```
SC_MODULE(encoder){
        void th_codification();

        SC_CTOR(encoder){
                SC_THREAD(th_codification);
        }
};
```

Figura 21: Definición de cada módulo

Una vez tenemos ambos módulos, los declaramos en el fichero main para probar que realmente están funcionando. Se añadió una funcionalidad muy sencilla a sus hilos para las pruebas y se declararon sus funciones, para mayor comodidad, en el propio fichero main. Queda reflejado en la *figura 22* el estado del sistema en este momento.

```
#include <systemc.h>
#include "testBench.h"
#include "encoder.h"

int sc_main(int argc,char* argv[]){

        testBench testB("TESTBENCH");
        encoder encoder("ENCODER");

        sc_start(1000, SC_SEC);
        return 0;
}

void testBench::th_test(){
}

void encoder::th_codification(){
}
```

Figura 22: Ambos módulos funcionando

El siguiente paso ha sido comunicar ambos módulos, así que fueron creados dos puertos de tipo sc_fifo en ambos módulos, uno de entrada y otro de salida. Los declaramos de tipo entero con el fin de enviar mensajes sencillos y poder ver por pantalla lo que se estaba haciendo. El resultado de esto se explica en detalle en la sección de pruebas.

Una vez estuvieron comunicados ambos módulos, y por consiguiente ya se tenía una estructura estable en cierto modo, se hizo la inclusión del código C en el sistema. Esto ha sido la base de todos los errores que ha habido en el proyecto y la mayor parte del tiempo consumido en la implementación. Se detallan todos estos problemas en las secciones pertinentes, pero basta decir que una de las fuentes más importantes de problemas ha sido el propio editor Eclipse, por lo que mezclar ambos códigos no hubiera sido un problema tan complejo a priori como lo ha sido.

Para combinar estos dos sistemas en uno solo y repartir la funcionalidad se ha seguido todo lo explicado en la fase de diseño. Se comenzó dividiendo las tareas de cada módulo: Testbench (TB) se encarga de generar ficheros, comprobar la entrada de consola, enviar los datos y escribir los ficheros de salida; Encoder (En) solamente consiste en un bucle infinito que va a realizar la codificación de cada estructura Image que se le envíe. Así pues, se dividió el código de la función principal del codificador en C y se introdujo en las funciones de los threads de cada módulo.

Además de esto, hubo que volver a crear todos los puntos de comunicación entre ambos módulos con el tipo de dato necesario en cada caso, estructuras en su mayoría. En este momento del proyecto hubo una ingente cantidad de errores, debidos a la mezcla de lenguajes y a la adaptación de un sistema cerrado a una nueva estructura, que se llevaron gran parte del tiempo de desarrollo y que, sin embargo, tuvieron su solución. También hubo que añadir las funciones propias del fichero lencod.c y asignárselas a los módulos que hacían uso de ellos.

Comprobado el buen funcionamiento en este punto, se realizaron pruebas a nivel de sistema para comprobar que el proceso queda inalterable y se da por concluido el proceso de implementación.

7.2.- Pruebas de la implementación

Siguiendo el planteamiento de análisis y diseño del sistema, a la par que se implementaba todo el mismo, se realizaron las pruebas por nivel. Aquí se desglosan las de nivel unitario y de integración, que como es lógico se han ido haciendo a lo largo del desarrollo. Las pruebas de sistema tendrán un capítulo propio en este documento, tanto por su importancia respecto a los objetivos como por su extensión.

7.2.1.- Unitarias

Estas han sido las pruebas que menos tiempo han llevado ya que, por el tipo de trabajo, no es necesario una comprobación exhaustiva de cada módulo. Dado, además, que toda la funcionalidad que se implementa está, teóricamente, reprobada por sus programadores, no se realizaron unas pruebas sobre el código C, por su complejidad y desconocimiento exacto. Así que, las pruebas unitarias se llevaron a cabo sobre los módulos que se implementaron anteriormente y más enfocadas a comprobar que funcionaba como tal la estructura SystemC que se creó.

Además, se realizaron pruebas de cobertura sobre el modelo y las funciones pertenecientes a los threads, así como a la funcionalidad añadida de control de los módulos. Una vez concluidas satisfactoriamente las pruebas unitarias se continuó con el proceso de pruebas de integración.

7.2.2.- Integración

Las pruebas de integración se realizaron conforme un orden descendiente. Esto es, primero probamos el funcionamiento correcto de cada módulo y después la interconexión entre ellos.

Como las pruebas unitarias ya recogen el funcionamiento de cada módulo, se realizaron pruebas directamente sobre la comunicación entre los módulos y el módulo principal que hace de manejador de ambos. Así mismo también se hicieron pruebas una vez se terminó de introducir el código C.

Primero se realizaron pruebas de comunicación entre ambos módulos. Como ya habian sido creados y están funcionando individualmente, se añadió cierta funcionalidad sencilla en el thread creado para cada uno de ellos. Para esta prueba se eligió crear una variable en el módulo de Testbench, asignarle un valor, enviarla al Encoder, este lo modifica y la devuelve, Testbench imprime el valor final en pantalla. El código desarrollado queda reflejado en la *tabla 3*.

Testbench	Encoder
Variable entera var1	Variable entera var2
Asignación de un valor estático	
Envío del dato →	Recibe el valor
	Se asigna a la variable var2
	Se realiza una sencilla operación aritmética
	Se devuelve el dato
Recepción del dato ←	
Muestra por pantalla el dato	

Tabla 3: Funcionalidad implementada en los módulos

Se estableció un mensaje inicial de arranque del sistema, así como una muestra del valor al envío y a la recepción (*Figura 23*). Así el sistema devolvía si la comunicación es correcta o se está realizando mal algún paso de la comunicación.

```
            SystemC 2.2.0 --- Apr 22 2012 15:21:08
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
Inicio del módulo TB (TB)
Variable preparada para ser enviada. (TB)
Esperando la variable modificada. (TB)
Inicio del módulo encoder(Enc)
Esperando a que llegue la variable. (Enc)
Valor recibido: 5 (Enc)
Valor que se enviará: 25 (Enc)
Valor final: 25
```

Figura 23: Resultados de la ejecución en la prueba de comunicación

Al trabajar en orden, primero se crea el módulo de Testbench, y cuando este queda a la espera, se arranca el segundo módulo Encoder, que toma el control del sistema y comienza a trabajar. Tiene la ventaja de que ya comienza con el dato en el bus, por lo que no tiene que esperar prácticamente a que llegue ningún dato. Por lo demás el proceso de envío y recepción es correcto.

Con esto queda probado que ambos módulos tienen una buena comunicación y, además, que el módulo principal está bien implementado.

La segunda parte de este apartado consistirá en comprobar, una vez hemos introducido el código C en el sistema, si los datos que deben se están pasando correctamente. Esta prueba será bastante más complicada que la anterior, ya que lo que se envían y se

reciben son estructuras. Así que primero se decidió probar que cada proceso de comunicación respeta el orden de envío y no se salta ningún trozo de código.

Para esto, se generan flags antes y después de cada proceso, para comprobar que el orden es correcto. Colocaremos en el envío de los datos de configuración, el envío de los datos de la imagen (de cada una de ellas, y comprobaremos que el número de la imagen es continuo) y de la imagen codificada. Además colocándolo antes y después mostrará si el proceso no realiza un envío sin recepción o se queda esperando una recepción infinita (problemas de clásicos de threads).

```
Máximo en salida: 16
Elementos en cola: 2
ImagenE: 0
0000(NVB)     168
0000(IDR)   25960 0 28  37.049  36.605  36.944      610       0     FRM    99
En la salida hay: 15
Elementos en cola: 1
ImagenE: 0
0000(IDR)   25960 0 28  37.049  36.605  36.944      409       0     FRM    99
En la salida hay: 14
Elementos en cola: 0
Leido: 0
Leido: 1
```

Figura 24: Resultados de la comunicación

En el texto de salida que se ha mostrado en la *figura 24* hay varias líneas del tipo: "0000 (Tipo)…" y una serie de datos. Estas líneas corresponden a la salida del resumen que ejecuta cada vez que llega al final del código del módulo Testbench. Así que esas deben ser ignoradas en el análisis. Como se puede ver en las demás tenemos una línea que muestra el tamaño máximo de las cosas fifo que hemos implementado (16 de manera estándar), los elementos que hemos dejado en la cola y que se van procesando y un flag de lectura del número de la imagen leída.

Con esto damos por concluidas las pruebas de integración, en las que hemos demostrado que nuestros módulos están correctamente relacionados, tanto con el módulo main como entre ellos, antes y después de añadir la funcionalidad C. Se llevará a cabo una segunda prueba de sistema sobre este proceso de comunicación en la sección pertinente.

7.3.- Problemas y soluciones

Varios han sido los problemas que han surgido en este proyecto. Muchos de ellos, posiblemente, fueran fruto de la inexperiencia y la complejidad del algoritmo elegido. Sin embargo, de los problemas es de lo que más se aprende.

Se ha dividido en dos tipos los problemas:

Problemas debidos al entorno: Estos han sido los menos importantes, pero que han permitido aprender qué problema hay a la hora de utilizar SystemC sobre Eclipse y una distribución Linux. Con el fin de no repetir errores ni perder tiempo en este tipo de problemas en proyectos futuros.

Problemas con el código: Los problemas encontrados debidos al desconocimiento del nuevo lenguaje. Se obviarán los problemas más sencillos de los primeros días de uso, y se hará un análisis más detallado sobre los más complejos que hayan surgido.

7.3.1.- Problemas de entorno

Comenzamos el proyecto preparando las librerías SystemC y enlazándolas con Eclipse. Esto llevó algo de tiempo, ya que es bastante sencillo una vez se conoce, pero algo complicado si no se ha hecho nunca. Además los tutoriales disponibles en la red (ya que en soporte físico no hay nada que explique algo tan concreto), no eran del todo específicos o no estaban completos. En poco tiempo se consiguió poner a funcionar el Eclipse para C/C++ con las librerías correctamente.

Otro problema que surgió fue que el Eclipse modificado para trabajar en C/C++ con las librerías de SystemC no permitía el uso de librerías referenciadas. Para resolver esto se probó en un nuevo proyecto el usar rutas absolutas externas y una vez funcionó, probamos sobre el proyecto del sistema. Este problema no pudo solucionarse, así que la solución fue cargar todos los ficheros en la ruta del proyecto definitivo.

Una vez resueltos este par de problemas y una vez terminado el sistema, también surgieron conflictos a la hora de utilizar la herramienta gprof.

7.3.2.- Problemas de código

Como en cualquier lenguaje de programación en el que se aprende a trabajar, se cometieron varios errores menores que no serán recogidos. Así que solamente se hará referencia a los que hayan supuesto un problema.

Un primer problema al que se hizo frente con respecto al código, fue la incompatibilidad al utilizar C con C++ de una forma directa. Por lo que se ha optado por utilizar la cadena de la *figura 25* como modelo estándar a seguir en todos los ficheros .h del código original.

```
#ifdef __cplusplus
        extern "C"{
#endif

Código desarrollado en C

#ifdef __cplusplus
}
#endif /* CPLUSPLUS */
```

Figura 25: Estructura del externC

Además de esto, surgieron conflictos de tipo entre las estructuras de comunicación entre los módulos. Así en vez de utilizar estructuras sencillas de tipo “sc_fifo” se han utilizado unas más complejas llamadas “sc_fifo_in_if”. A su vez, la utilización de estructuras fifo que trabajan sobre estructuras C, genera una cadena de error que nos indica que no existen operadores de tipo << (En C++ representan operadores de salida

de escritura) con estas estructuras. Esto es lógico pues está preparado para utilizar datos sencillos: enteros, reales, caracteres… Así que para solucionar esto se tuvo que crear la operación con las estructuras C en el propio fichero de main.cpp. Aquí también se siguió un modelo estandarizado para la implementación como se refleja en la *figura 26*, aunque hay que definir cada elemento que se quiere obtener mediante el operador de forma personalizada dependiente de cada estructura.

```
inline ostream& operator << ( ostream& os,  ImageParameters const & v ) {
        os << "Number: "<<v.number;
        os << " ";
        os << "Pn: "<<v.pn;
        os << " ";
        os << "Nb_references: "<<v.nb_references;
        os << " ";
        os << "current_mb_nr: "<<v.current_mb_nr;
        os << " ";
        // ...
        os << "Type: "<<v.type;
        os << " ";
        os << "Structure: "<<v.structure;
        os << " ";
        os << "Num_reference_frames: "<<v.num_reference_frames;
   return os;
}
```

Figura 26: Función definida del operador "<<"

El último error de este tipo que se solucionó fue el referente a las trazas. Lo mismo que pasaba con el caso anterior, SystemC está preparado para realizar trazas con tipos de datos sencillos. Así que volvimos a utilizar la solución anteriormente citada, creando las funciones de "sc_trace" y mejorándolas para poder utilizar las estructuras que se necesitan para este caso concreto.

Además se encontraron problemas a la hora de declarar las funciones procedentes del fichero "lencod.c", donde antes residía el main, que hizo que se analizara cada módulo con su funcionalidad para encontrar qué parte de ellos hacía uso de qué funciones auxiliares.

Capítulo 8

8.- Evaluación

En este capítulo del documento vamos a presentar todas las pruebas que hemos realizado a nivel de sistema, las pruebas de rendimiento general del código desarrollado, así como los problemas que hemos tenido a la hora de desarrollar el proyecto en conjunto y las conclusiones que hemos extraído de esos problemas y de lo que hemos codificado.

8.1.- Pruebas de Sistema

Todas las pruebas que se van a realizar a continuación están basadas en un sistema virtualizado sobre un equipo portátil. Ambos disponen de las siguientes especificaciones:

Máquina Host	Máquina Virtual – VMWare Workstation
Intel Core i3-2330M, 2.2Ghz	Monoprocesador de doble núcleo
4GB	1280 MB
Windows 7	Ubuntu 11.04 -- Natty Narwhal

Tabla 4: Especificaciones de los sistemas de pruebas

8.1.1.- Prueba 1: Funcionamiento correcto

Esta será la primera prueba de sistema y la más sencilla, ya que simplemente se pretende comprobar que realmente el sistema concuerda en cuanto a estadísticas de proceso y resultados con el sistema originario en C.

Para esto se hace uso de un modelo dorado original y es comparado con los resultados de salida al ejecutar sistema completo. Como se ve en la *figura 27* se obtienen unos resultados bastante similares.

Analizando los resultados podemos apreciar que existe un ligero aumento del tiempo de ejecución. Esta es la única diferencia que con respecto al modelo dorado. Resulta obvio este incremento ya que estamos añadiendo líneas de código al crear los módulos y crear una función main que los instancia. Por lo demás el sistema creado funciona correctamente.

ORIGINAL

```
Freq. for encoded bitstream       : 15
Hadamard transform                : Used
Image format                      : 176x144
Error robustness                  : Off
Search range                      : 16
No of ref. frames used in P pred  : 5
No of ref. frames used in B pred  : 5
Total encoding time for the seq.  : 1.524 sec
Total ME time for sequence        : 0.411 sec
Sequence type                     : IBPBP (QP: I 28, P 28, B 30)
Entropy coding method             : CABAC
Profile/Level IDC                 : (77,30)
Search range restrictions         : none
RD-optimized mode decision        : used
Data Partitioning Mode            : 1 partition
Output File Format                : H.264 Bit Stream File Format
------------------ Average data all frames  -----------------------------------
SNR Y(dB)                         : 35.17
SNR U(dB)                         : 33.65
SNR V(dB)                         : 34.97
Total bits                        : 157216 (I 98840, P 15936, B 42272 NVB 168)
Bit rate (kbit/s)  @ 30.00 Hz     : 1572.16
Bits to avoid Startcode Emulation : 0
Bits for parameter sets           : 168
```

SYSTEMC

```
Freq. for encoded bitstream       : 15
Hadamard transform                : Used
Image format                      : 176x144
Error robustness                  : Off
Search range                      : 16
No of ref. frames used in P pred  : 5
No of ref. frames used in B pred  : 5
Total encoding time for the seq.  : 3.637 sec
Total ME time for sequence        : 1.106 sec
Sequence type                     : IBPBP (QP: I 28, P 28, B 30)
Entropy coding method             : CABAC
Profile/Level IDC                 : (77,30)
Search range restrictions         : none
RD-optimized mode decision        : used
Data Partitioning Mode            : 1 partition
Output File Format                : H.264 Bit Stream File Format
------------------ Average data all frames  -----------------------------------
SNR Y(dB)                         : 35.17
SNR U(dB)                         : 33.65
SNR V(dB)                         : 34.97
Total bits                        : 157216 (I 98840, P 15936, B 42272 NVB 168)
Bit rate (kbit/s)  @ 30.00 Hz     : 1572.16
Bits to avoid Startcode Emulation : 0
Bits for parameter sets           : 168
```

Figura 27: Resultado de la ejecución original y en nuestro sistema.

8.1.2.- Prueba 2: Porcentaje de utilización de funciones y tiempo de ejecución relativo

Para realizar esta prueba se ha utilizado una herramienta de análisis de rendimiento propia de Unix: Gprof =[2]=[2]. Es una herramienta que va a servir para monitorizar el uso que se hace del código en su ejecución. Es descendiente de la herramienta Prof, pero añade la capacidad de poder crear gráficas con los resultados. Y esta característica va a ser la más importante, porque es la que se va a utilizar para conocer qué funciones son las que más tiempo de ejecución están utilizando.

Para poder utilizar la funcionalidad mediante Eclipse se instalan varios paquetes en el sistema Unix. Después se compila el sistema añadiendo la opción –pg, tanto en el proceso de compilación como en el de linkado. Esto genera un fichero “gmon.out” que se ejecuta sobre el fichero binario resultante de la compilación. Hecho esto se puede observar el resultado de la prueba como se indica en la *figura 28*.

Name (location)	Samples	Calls	%Time
▼ Summary	266		100,0%
▶ SetupFastFullPelSearch	72	495	27,07%
▶ FastFullPelBlockMotionSearch	23	20295	8,65%
▶ dct_luma	22	180213	8,27%
▶ biari_encode_symbol	16	7622369	6,02%
▶ SubPelBlockMotionSearch	16	20295	6,02%
▶ SATD	12	943367	4,51%
▶ write_significant_coefficients	7	204029	2,63%
▶ write_significance_map	6	204029	2,26%
▶ UnifiedOneForthPix	6	8	2,26%
▶ FastPelY_14	5	26424768	1,88%
▶ UMVPelY_14	5	3421616	1,88%
▶ writeRunLevel_CABAC	5	1472566	1,88%
▶ biari_encode_symbol_eq_prob	5	1347749	1,88%
▶ RDCost_for_4x4IntraBlocks	5	151701	1,88%
▶ UMVLine16Y_11	4	1702720	1,5%
▶ getNonAffNeighbour	4	989170	1,5%
▶ get_mb_block_pos	4	962546	1,5%
▶ dct_luma_16x16	4	1071	1,5%
▶ sign	3	4677952	1,13%

Figura 28: Resultado de la ejecución de Gprof que muestra los porcentajes de utilización.

Una vez obtenido este resultado, tomamos las 5 primeras funciones como las más influyentes en el tiempo de ejecución del sistema. Todas ellas pertenecen al módulo Encoder, por lo que son las más interesantes a la hora de realizar el análisis.

Después de realizar la búsqueda de todas las funciones, se continúa con el análisis temporal de cada una de ellas. Esto es, se medirá el tiempo que tarda desde el inicio hasta el fin de cada una de las funciones, y además el tiempo de ejecución de todo el sistema y de cada módulo (no vale uno solo, pues habrá tiempo residual de la instanciación de los módulos).

```
#include <sys/time.h>
struct timeval t_ini, t_fin;
double timeval_diff(struct timeval *a, struct timeval *b)
{
  return
                (double)(a->tv_sec + (double)a->tv_usec/1000000) -
                (double)(b->tv_sec + (double)b->tv_usec/1000000);
}
```

Figura 29: Estructura de lectura de tiempos.

En la imagen de la *figura 29*, se muestra la estructura que se utilizará para tomar las medidas. Con esa estructura se han tomado todas las medidas de la *tabla 5*.

Función	Sample	Calls	% Tiempo	Tiempo de ejecución(s)
SetupFastFullPelSearch	72	495	27.07	1.115284
FastFullPelBlockMotionSearch	23	20295	8.65	0.35638
Dct_luma	22	180213	8.27	0.340724
Biari_encode_symbol	16	7622369	6.02	0.248024
SubPelBlockMotionSearch	16	20295	6.02	0.248024

Tabla 5: Resultados de la medición temporal

Con esto podemos ver que las funciones que más porcentaje, de este grupo de las 5 que más consumen, tienen prácticamente menos llamadas que las siguientes. Es un dato interesante que indica que tenemos una función que se llama pocas veces (relativamente) pero que consume bastante tiempo de proceso.

Una vez medidos estos datos, se ha tomado también muestras de los tiempos que tarda cada uno de los módulos y el sistema total, y se ha calculado el tiempo restante como se muestra en la *tabla 6*. Además se ha evaluado el porcentaje de utilización de cada módulo del sistema. Como se puede apreciar, el tiempo destinado a la preparación del módulo TestBench y a la instanciación y preparación de los módulos en el método main(), son totalmente irrelevantes. El módulo que, obviamente, más tiempo consume es el Encoder, que se hace cargo de 99'6% de la carga de trabajo. Por lo que, como propuesta, este sería el módulo a optimizar, pero como modelo final, sería el único que se podría optimizar.

Módulo	% Tiempo	Tiempo de ejecución (s)
Sistema Completo	100	4.44
TestBench	0.0012	0.00195
Encoder	0.996	4.426
Residual	0.0028	0.0125

Tabla 6: Muestras temporales y percentuales de carácter general del sistema

Una vez realizadas estas medidas temporales, se ha procedido con la búsqueda de cada uno de las funciones (*Tabla 7*). Esto servirá en el momento de determinar si estamos preparando bien el plan de mejora, o por el contrario, se están utilizando funciones del TestBench, que no tienen relevancia en este análisis, ya que no podremos paralelizar su funcionamiento en el sistema final, ya que no existirá.

Función	Módulo origen
SetupFastFullPelSearch	Encoder
FastFullPelBlockMotionSearch	Encoder
Dct_luma	Encoder
Biari_encode_symbol	Encoder
SubPelBlockMotionSearch	Encoder

Tabla 7: Módulo de las funciones más consumidoras temporalmente del sistema.

Es bastante interesante y obvio el hecho de que todas las funciones pertenecieran al Encoder, lo que va a limitar su frecuencia de trabajo, sobre el que se generará la especificación final del elemento. Además esto deja al descubierto las 5 funciones que, con diferencia, consumen más tiempo del encoder, lo que supone que se podrían paralelizar en un proyecto futuro y obtener un speedup mucho mayor que si sólo hubiera una o dos funciones lenta.

Llegados a este punto, se subdivide la medición temporal de las funciones según el tipo de imagen que están codificando. Esto se muestra en la tabla 8 a continuación. Se muestran tanto el tiempo como el porcentaje de tiempo que se invierte en cada función de acuerdo al tipo.

Función	Imagen I (s)		Imagen II (s)	
SetupFastFullPelSearch	0.0446		1.07	
FastFullPelBlockMotionSearch	0.014		0.34	
Dct_luma	0.0136		0.32	
Biari_encode_symbol	0.0099		0.24	
SubPelBlockMotionSearch	0.0099		0.24	
Total	0.092	2%	2.21s	56%

Tabla 8: Tiempos separados y porcentaje sobre el total de cada tipo de imagen

Con esta información se ha confeccionado la *figura 30* que muestra el uso que se hace de cada tipo de imagen dentro del periodo de un ciclo del sistema.

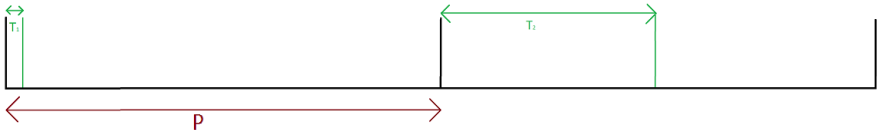

Figura 30: Periodo de ciclo del sistema

En donde T_i representa el tiempo invertido en cada uno de los tipos de imagen y Periodo de procesado de imagen.

Con toda la información recopilada en esta prueba, podemos establecer un periodo de trabajo de nuestro sistema final.

8.1.3.- Prueba 3: Velocidad media de compresión de una imagen

Los datos obtenidos en esta prueba serán la velocidad media que tarda en comprimir una imagen y la cantidad de imágenes que tiene de media un vídeo de determinados segundos.

Para la medida de velocidad de compresión, se van a utilizar diversos (5) videos descargados de la página web de la Universidad de Arizona[6], todos en formato .yuv y de una duración máxima de 10 segundos. Estos videos son de temas aleatorios y tienen una cantidad de colores y movimientos indeterminados.

Se ejecutará cada uno 10 veces, se obtendrá una media del tiempo de ejecución, y se extraerá el tiempo de ejecución de cada imagen en que se subdivida el video, volveremos a hacer 10 pruebas y obtendremos la media. Esto lo haremos con cada video, por lo que al final tendremos 50 medidas sobre las que ejecutar una media y obtener un resultado más o menos aproximado. En la *tabla 9* se muestran los videos seleccionados y su duración temporal. Además se incluyen el número de frames totales de cada vídeo.

Video	Duración(frames)	Tamaño (Kb)
Stephan	90	13.365
Miss America	150	5.569
Waterfall	260	8.192
Akiyo	300	11.138
Highway	2000	74.25

Tabla 9: Videos elegidos con su duración temporal y espacial.

1) Video I: Stephan

Se va a realizar un procedimiento estándar para los 5 videos. Primero se crea un medidor temporal como en pruebas anteriores que devolverá el tiempo entre cada imagen. Y además un medidor al final del Encoder que mostrará el tiempo total de procesado de ese video. Así queda, como se muestra en la *figura 33*, la gráfica del primer video después de 10 ejecuciones.

[6] http://trace.eas.asu.edu/yuv/

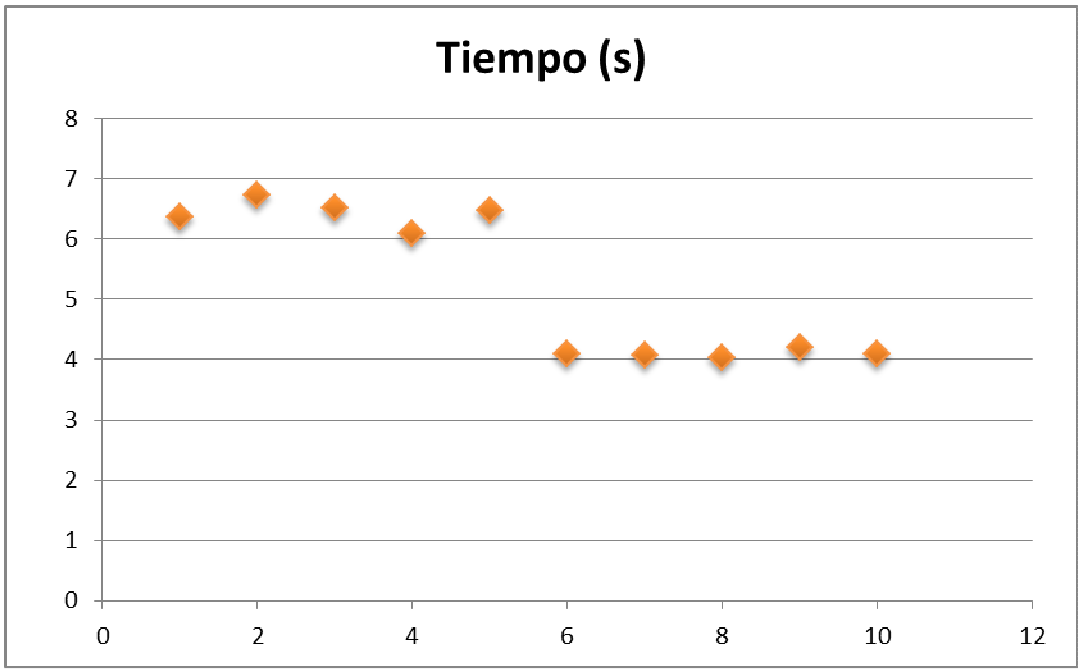

Figura 311: Gráfica temporal de cada ejecución

Media: 5.269072 s

Además se han tomado los valores en esas 10 ejecuciones del tiempo de cada imagen codificada y se han apuntado en la *tabla 10*. Calculamos las medias de cada imagen procesada y los datos quedan reflejados en la *tabla 11*.

Prueba	Tiempo Imagen I (s)	Tiempo Imagen II (s)
1	1.88	4.4
2	2.32	4.352
3	2.648	3.856
4	2.3824	4.0752
5	2.376	3.7064
6	0.864	3.216
7	0.872	3.1992
8	0.8544	3.176
9	0.8676	3.272
10	0.87544	3.2

Tabla 10: Anotaciones temporales para cada imagen en cada prueba**.**

Prueba	Tiempo medio (s)	Tiempo / frame (s)
Imagen I	1.593896	0.01771
Imagen II	3.64528	0.0405032
Total	2.619588	0.0291066

Tabla 11: Resultados finales del video I.

2) Video II: Miss América

Con el segundo video se han realizado las mismas pruebas. Primero la gráfica temporal de ejecución total del sistema, 10 pruebas:

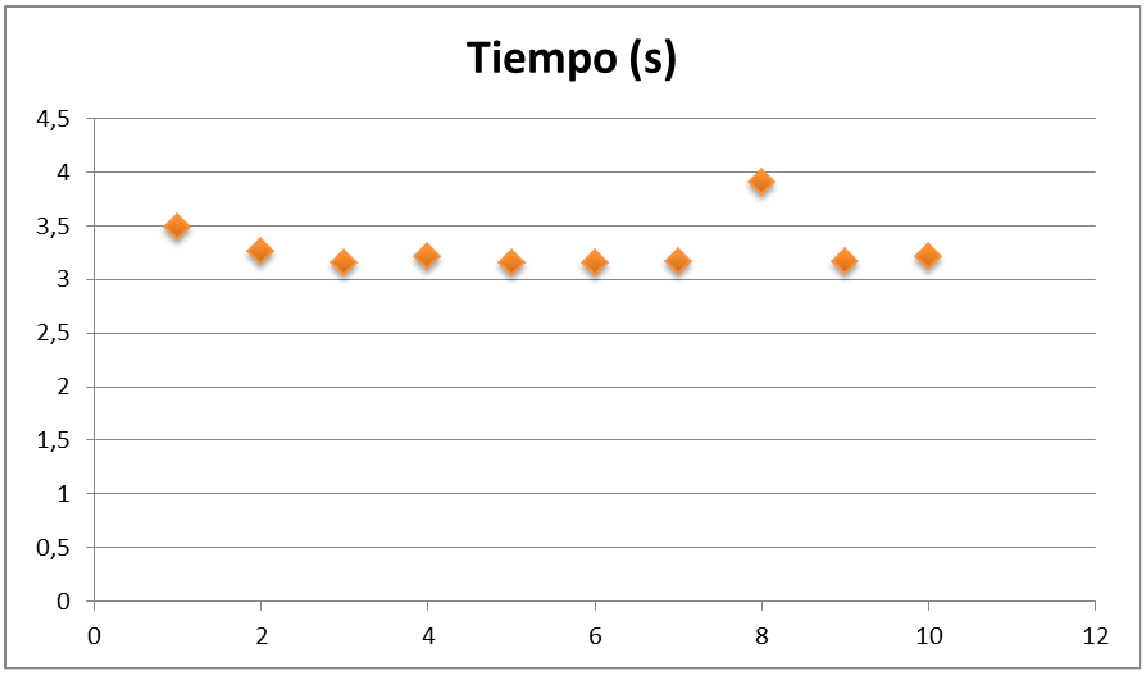

Figura : Tiempo medio de ejecución de la prueba II
Media: 3.28508 s

Después los cálculos anotados durante la ejecución con valores de cada imagen.

Prueba	Tiempo Imagen I (s)	Tiempo Imagen II (s)
1	0.74776	2.6544
2	0.4832	2.76
3	0.4928	2.648
4	0.4848	2.7112
5	0.5104	2.744
6	0.49464	2.64
7	0.4944	2.692
8	0.4852	2.656
9	0.4864	2.6508
10	0.4928	2.653752

Tabla 12: Resultados finales del video II.

Y para terminar el valor medio de todas las medidas:

Prueba	Tiempo medio (s)	Tiempo / frame (s)
Imagen I	0.962416	0.006416112
Imagen II	2.4387352	0.0162584
Total	1.7005756	0.011337256

Tabla 13: Resultado medio de la prueba II.

3) Video III: Waterfall

Valores medios de ejecución del tercer vídeo:

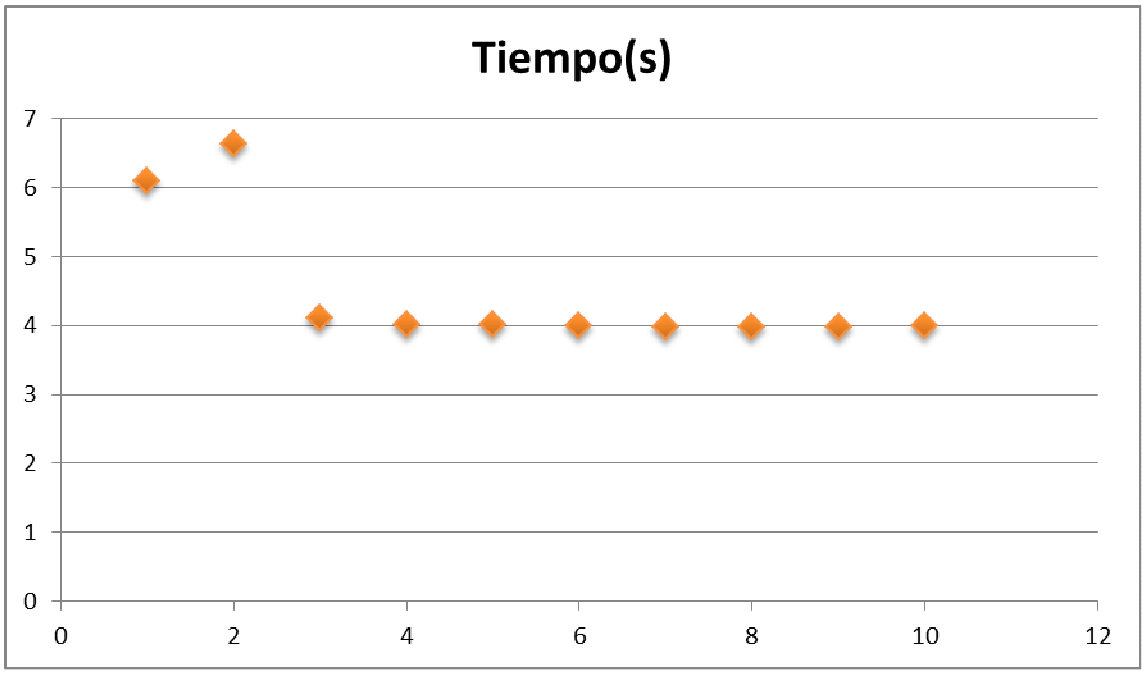

Figura 32: Gráfica de tiempos de ejecución totales en la prueba III
Media: 4.476576 s

Anotaciones de los tiempos

Prueba	Tiempo Imagen I (s)	Tiempo Imagen II (s)
1	2.2672	3.76
2	1.904	4.696
3	0.6752	3.412
4	0.664	3.32
5	0.656	3.3256
6	0.692	3.296
7	0.6648	3.3064
8	0.6792	3.28
9	0.67032	3.3
10	0.6712	3.304

Tabla 14: Medidas temporales de la prueba III

Valor final de las medias

Prueba	Tiempo medio (s)	Tiempo / frame (s)
Imagen I	3.275672	0.0125984
Imagen II	3.5	0.0134616
Total	3.387836	0.1303

Tabla 15: Tiempos medios de la prueba III

4) Video IV: Akiyo

Valor medio del cuarto video:

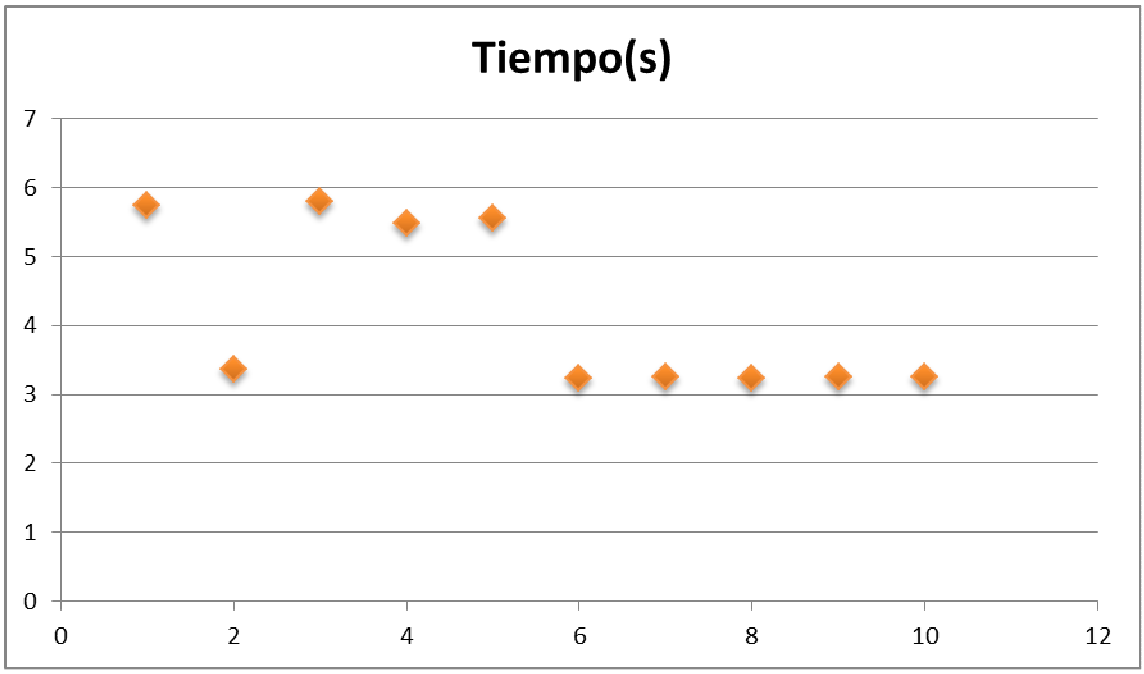

Figura 33: Muestra de la variación de tiempo de ejecución para la prueba IV
Media: 4.2149592 s

Tiempos de cada imagen codificada:

Prueba	Tiempo Imagen I (s)	Tiempo Imagen II (s)
1	1.72104	3.9888
2	0.552	2.6696
3	1.512	4.274
4	1.544	3.9384
5	1.488	4.048
6	0.546	2.684
7	0.54312	2.688
8	0.5556	2.6608
9	0.54544	2.68688
10	0.54936	2.698

Tabla 16: Tiempos medidos en la prueba IV

Media de los tiempos:

Prueba	Tiempo medio (s)	Tiempo /frame (s)
Imagen I	0.955656	0.00318552
Imagen II	3.233648	0.0107792
Total	2.094652	0.06976

Tabla 17: Tiempos medios de la prueba IV

5) Video V: Highway

Gráfica de tiempos medios de las 10 ejecuciones:

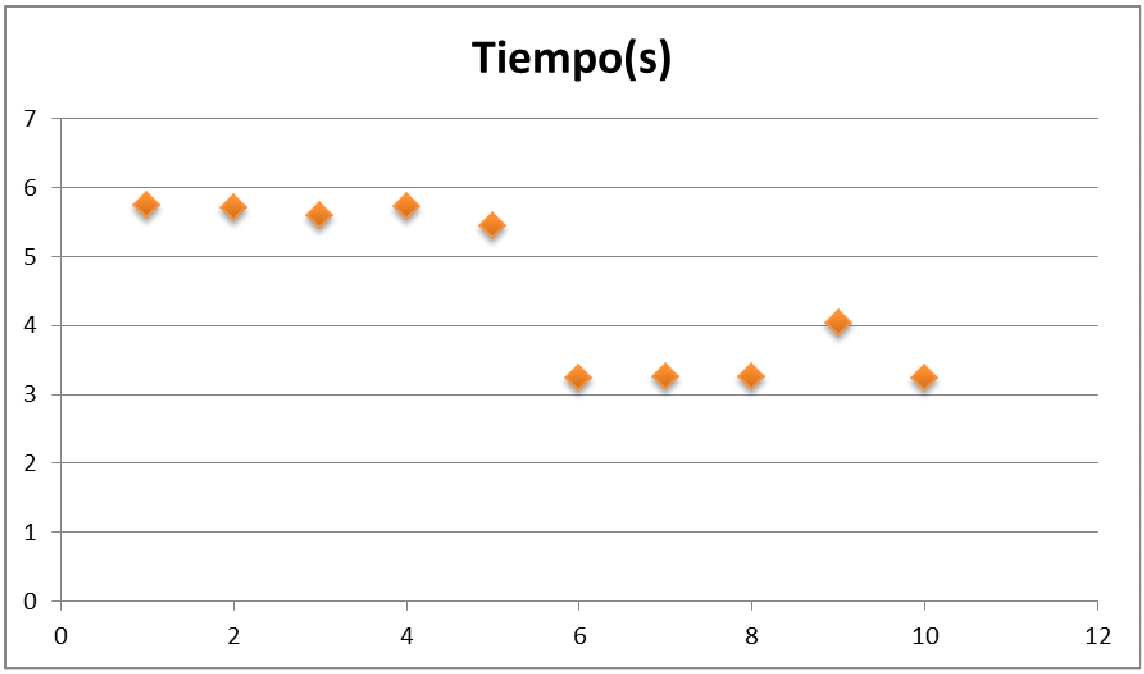

Figura 34: Tiempos medidos de ejecución total sobre el video V.
Media: 4.519416 s

Tiempo de cada una de las imágenes codificadas durante las 10 ejecuciones:

Prueba	Tiempo Imagen I (s)	Tiempo Imagen II (s)
1	1.368	4.3536
2	1.4496	4.24
3	1.5088	4.08
4	1.336	4.368
5	1.512	4.096
6	0.5136	2.7056
7	0.51392	2.7216
8	0.59896	2.7168
9	0.51496	2.71824
10	0.517576	2.704

Tabla 18: Medidas tomadas en la prueba V

Tabla de valores finales:

Prueba	Tiempo medio(s)	Tiempo medio / frame (s)
Imagen I	0.9833416	0.000491672
Imagen II	3.470384	0.0017352
Total	2.2268628	0.001113436

Tabla 19: Tiempos medios de la última prueba

Una vez concluidos estas 5 pruebas (50 ejecuciones en total), se calculan todos los tiempos medios para el cálculo de tiempo medio final de la velocidad de codificación del sistema. El resultado de los cálculos se incluye en la *tabla 20*, y es referente al tiempo medio por frame de ejecución.

Tipo de prueba	Tiempo medio final (s)
Global	4.35302072
Imagen	0.0123126584

Tabla 20: Resultados finales de la velocidad media

8.1.4.- Prueba 4: Comprobación de los ficheros de salida.

Esta prueba determinará si el resultado codificado es correcto o si estamos realizando una codificación libre e independiente de la original. Una vez se ejecutan ambos sistemas se tiene una salida de varios ficheros que representan tanto las estadísticas como el video codificado (*Tabla 21*).

Input YUV file	**stefan_cif.yuv**
Output H.264 bitstream	**test.264**
Output YUV file	**test_rec.yuv**
Output log file	**log.dat**
Output statistics file	**stat.dat**

Tabla 21: Ficheros de salida del sistema

El primer fichero es el de entrada que se ha utilizado para codificar. Se genera un fichero de salida correspondiente al H.264, así como el fichero de tipo YUV, un log y unas estadísticas. Además de este se ha ejecutado con los mismos parámetros y condiciones el codificador original de Fraunhofer Heinrich Hertz Institute.

Además de haber generado estos ficheros, con una rápida comprobación se puede apreciar que los 4 son del mismo tamaño y tienen el mismo contenido en el caso de log y stat. Por lo que nuestras salidas son idénticas hasta el momento.

8.1.5.- Prueba 5: Comparación de velocidad entre el sistema creado y el original

Se han ejecutado los mismos tests en el sistema software original, tomando los mismos videos y con la misma cantidad de pruebas, y la velocidad media de las compresiones se muestran en la *tabla 22*.

Sistema	Tiempo medio de una imagen	Tiempo medio total
Modelo SystemC	0.0123126584	4.35302072
Sistema JM8 en C	0.017	2.53

Tabla 22: Comparación entre ambos sistemas

Como puede apreciarse, tenemos una velocidad superior en el sistema de pruebas, como ya se dijo previamente, pero esto no es un problema, ya que la finalidad de estos lenguajes no es esta, sino verificar el comportamiento y saber qué mejoras nos proporcionan un speedup real y porcentual. Habiendo no aplicado ninguna mejora, es obvio que el tiempo es mayor.

8.2.- Problemas encontrados y soluciones

El problema más importante que se ha encontrado en el proyecto a la hora de realizar estas pruebas ha sido en el primer punto, en llegar a que el sistema devolviera los mis-

mos resultados. Ha sido complicado obtener unos resultados tan exactos debido a lo complejo del sistema original elegido. Sin embargo, se ha conseguido un modelo SystemC estable sobre el que realizar las demás pruebas con la misma exactitud en los cálculos que el original.

A este motivo surgieron varios problemas, como el hecho de que las variables globales pertenecientes a "global.h" degradaban el código metiéndose en prácticamente todos los ficheros. Resultó complicado localizar todas esas instancias de variables globales y suplantarlas localmente. Esto reportó varios errores que perduraron en el tiempo lo suficiente como para ralentizar el trabajo durante largos periodos.

Otro de los problemas más importantes y que más tiempo de trabajo consumió del proyecto fue la toma de datos del Testbench y el envío hacia el Codificador. El complejo entramado de estructuras del código original a la hora de declarar los datos (estructura Image compuesta de varias estructuras en cascada en varios niveles) hizo que un error en la cantidad de bits contadas por el sistema se llevara bastante tiempo en el buceo dentro de las estructuras de datos y del código y las funciones. Problema añadido la poca utilidad del debugger de Eclipse en códigos C cuando se trabaja con estructuras enlazadas. El problema en sí quedó resuelto fácilmente, pero se perdió mucho más tiempo en la búsqueda del disponible.

Además de estos problemas, han surgido otros relacionados con el resto de pruebas. Por ejemplo, la utilización de las librerías de SystemC generan cierta incompatibilidad y crea algunos problemas a la hora de utilizar el GProf.

También hubo que reinstalar en varias ocasiones las librerías SystemC hasta conseguir que las señales se consiguieran mostrar de forma correcta.

8.3.- Conclusiones

Aparte de los problemas con las herramientas y con el funcionamiento del sistema general a la hora de asentarlo sobre el modelo de módulos que se creó previamente, no han surgido problemas a la hora de realizar pruebas ni se han obtenido resultados inesperados, las pruebas han resultado exitosas.

Por tanto, se ha conseguido crear un modelo funcional en SystemC con la funcionalidad repartida en dos módulos, comunicándose entre sí los datos necesarios para que el codificador H.264 realice su tarea sobre el video de entrada. Con las pruebas aportadas en este apartado, inicialmente se pretendía demostrar que SystemC es un lenguaje útil hoy en día. Con los resultados positivos que hemos obtenido, este hecho queda constatado.

Las pruebas han resultado bastante interesantes, sobretodo la segunda en que se muestra que una gran parte del proceso de codificación está concentrada en una misma función (+27%). Tras una pequeña búsqueda se detecta que la segunda función que más tiempo consume (en torno al 9%), es una función que es invocada dentro de la anterior, por lo que su tiempo de proceso pertenece al anterior. Esta es la función objetivo de un futuro trabajo, ya que es la que más consume con diferencia y de la que se podría extraer el speedup más grande. Por la misma razón, también se convierte en nuestra función más limitante en la segunda prueba.

Además de este tenemos otras 3 funciones que son también importantes (tomada la medida a partir del 5%), que pertenecen también al codificador y se podría trabajar sobre ellas también.

Tras estas pruebas y análisis se puede entrever que es una herramienta potente a la hora de establecer mejoras en el sistema y comprobar el comportamiento correcto del mismo.

Capítulo 9

Conclusiones generales y trabajos futuros

Con la realización de este proyecto se pretendía demostrar que este tipo de sistemas, utilizados en una empresa real en un desarrollo real, suponen una reducción de coste a lo largo del ciclo de vida del proyecto. Se ha propuesto una explicación sencilla pero lo más completa posible sobre este tipo de sistemas y de cómo y por qué se implementan para introducir al lector en el co-diseño de estos. Además se ha realizado una serie de pruebas sobre el mismo, tanto a nivel de módulo como el sistema en conjunto.

Sobre estas últimas pruebas será sobre las que se basen estas conclusiones, ya que es, en parte, uno de los dos objetivos principales del proyecto en sí. Como se ha demostrado, este tipo de lenguajes van a permitir realizar implementaciones de código C/C++ en un plano de diseño orientado al Hardware futuro, y sobre éste realizar pruebas tanto de comportamiento como de rendimiento. Sobrentendiendo "medidas de rendimiento" de una forma cuidadosa. Con estas, se demuestra que este tipo de implementación es útil a una empresa para reducir tiempo y coste en cuanto al que supone no disponer de esta herramienta y realizar pruebas sobre sistemas físicos directamente.

Además, en un segundo plano, pero no menos importante, tenemos el aprendizaje y formación del alumno en el campo del modelado en SystemC. El hecho de que el proyecto en sí haya concluido y que el sistema funcione correctamente y responda favorablemente a todos los teses, supone en sí mismo una prueba de que el aprendizaje ha sido positivo. Hay partes que han resultado más duras que otras en cuanto a una realización exitosa y a que hayan surgido más cantidad de problemas o errores, pero un aprendizaje práctico es siempre positivo. A pesar de que también haya habido ese aprendizaje previo por parte del alumno con el manual que ha sido referente en todo el proceso de desarrollo.

Una conclusión también interesante es el hecho de que, al estudiar el estado del arte en estos sistemas, se haya comprobado que efectivamente es una tecnología en uso actualmente y con bastantes posibilidades de expansión como pueden ser SystemC-AMS.

Como trabajo futuro queda lo que supondría una ventaja más sobre un sistema creado sin SystemC y que consiste en la optimización de este sistema. Como hemos visto a lo largo de las pruebas, hay partes del código que consumen más tiempo que otras. Su división en módulos, al igual que hicimos con el sistema completo, permitiendo un procesado en paralelo, supondría un speedup con respecto al modelo que ya poseemos. Este sería un primer punto sobre el que fijar el rumbo de este trabajo y el más inmediato.

"¿Acabar? El viaje no acaba aquí, la Muerte es solo otro camino que todos tenemos que tomar."
Gandalf, El señor de los anillos: El retorno del Rey.

Bibliografía

[1] D.C. Black, J. Donovan, and B. Bunton. *SystemC: From the Ground Up, Second Edition: From the Ground Up*. NATO ASI series: Cell biology. Springer, 2009.

[2] T. Grötker. *The Developer's Guide to Debugging*. SpringerLink: Springer e-Books. Springer, 2008.

[3] H.de J. Ochoa-Domínguez, J. Mireles-García, and J. de D. Cota-Ruíz. Descripción del nuevo estándar de video H.264 y comparación de su eficiencia de codificación con otros estándares. *Ingeniería, investigación y tecnología*, 8:157 – 180, 09 2007.

[4] J. Russell and R. Cohn. *Yuv*. Book on Demand, 2012.

[5] W. von Hagen. *The Definitive Guide to GCC*. Apress, 2006.

[6] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560 –576, july 2003.