UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Integración de un modelo de referencia descrito en C en un entorno de verificación UVM

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención: Sistemas Electrónicos
Autor: Alejandro Piñan Roescher
Tutor 1: Valentín de Armas Sosa
Tutor 2: Félix Bernardo Tobajas Guerrero

Fecha: Julio, 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Integración de un modelo de referencia descrito en C en un entorno de verificación UVM

HOJA DE FIRMAS

Alumno

Fdo.: Alejandro Piñan Roescher

Tutor

Fdo.: Valentín de Armas Sosa

Tutor

Fdo.: Félix B. Tobajas Guerrero

Fecha: Julio 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Integración de un modelo de referencia descrito en C en un entorno de verificación UVM

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario**

Fdo.: Fdo.:

Fecha: Julio 2017

Agradecimientos

En primer lugar quiero agradecer a mis padres, mi hermana, mi novia y familia, pues siempre han estado conmigo en los momentos difíciles, apoyándome para poder seguir adelante y enseñándome a no rendirme hasta alcanzar mis objetivos. Gracias por saber comprenderme y apoyarme.

En segundo lugar quiero agradecer a todos los profesores que han contribuido en esta etapa tan importante de mi vida, aportanto los distintos conceptos que me han formado como Ingieniero en la titulación de Grado en Tecnologias de las Telecomunicaciones. Destacando en especial, a mis tutores Valentín de Armas Sosa y Félix B. Tobajas Guerrero, quienes en todo momento me han ayuda y aconsejado de la mejor manera posible, resaltando que sin ellos esto no hubiera sido posible, gracias.

Finalmente, y no menos importante, agradecer a mis compañeros de grado quienes han hecho más llevadera esta estapa de mi vida. Pues, nos hemos ayudado y apoyo formando un equipo de compañeros y amigos.

Resumen

Durante las últimas décadas los circuitos electrónicos han crecido en complejidad y en coste de producción, lo que ha obligado a los ingenieros a investigar y desarrollar nuevos métodos para la verificación de diseños electrónicos de manera más completa, detallada y eficiente. En este contexto, uno de los resultados más importantes de la creciente necesidad de verificación de los sistemas digital actuales, es la metodología UVM (*Universal Verification Methodology*). Esta metodología, proporciona todos los elementos y mecanismos necesarios para llevar a cabo el proceso de la verificación mediante la construcción de bancos de prueba formados por componentes descritos en lenguaje *SystemVerilog*.

No obstante, estos componentes pueden estar descritos en diferentes lenguajes y los integradores y diseñadores deben combinar estos componentes para proporcionar una solución para las necesidades de cada entorno o caso de verificación.

En determinados ámbitos los algoritmos se desarrollan y se evalúan previamente utilizando modelos descritos en lenguaje C antes de su implementación RTL (*Register Transfer Level*). Una vez finalizado el algoritmo, estos modelos en C se pueden reutilizar como modelos de referencia (*Golden Reference*) para la verificación de módulos y/o sistemas. Dado que tanto la metodología UVM como la capacidad de las herramientas EDA soportan múltiples lenguajes de descripción, es posible plantear la integración de un modelo de referencia descrito en el lenguaje C para generar la respuesta esperada, y a partir de ésta compararla con la respuesta obtenida del dispositivo bajo verificación (DUV).

Abstract

During the last decades electronic circuits have grown in complexity and cost of production, which has forced the engineers to investigate and develop new methods for the verification of electronic designs in a more complete, detailed and efficient way. In this context, one of the most important results of the growing need for verification of current digital systems is the UVM (Universal Verification Methodology). This methodology provides all the necessary elements and mechanisms to carry out the verification process by implementing testbenches made up of components described in the SystemVerilog language.

However, these components may be described in different languages and integrators and designers must combine these components to provide a solution to the needs of each environment or verification case.

In certain domains, the algorithms are developed and evaluated previously using the models described in C language before their RTL (Register Transfer Level) implementation. Once the algorithm is complete, these C models can be reused as reference models (Golden Reference) for the verification of modules and/or systems. Since both the UVM methodology and the EDA tools capability support multiple description languages, it is possible to propose the integration of a reference model described in the C language to generate the expected responses, and from this to compare them with the responses obtained from the Device Under Verification (DUV).

Índice de contenidos

Índice de figuras V
Índice de códigos VII
Índice de tablas IX
Índice de ecuaciones XI
Acrónimos XIII
Parte I Memoria Descriptiva 1
Capítulo 1 Introducción 3
1.1. Introducción y antecedentes 3
1.2. Objetivos 6
1.3. Peticionario 7
1.4. Organización de la memoria 7
Capítulo 2 *Universal Verification Methodology* 11
2.1. Introducción 11
2.2. Introducción a la verificación funcional de sistemas digitales 12
2.3. Lenguajes de verificación hardware 14
2.4. Evolución de las metodologías de verificación 17
2.5. La metodología UVM 18
2.6. Librería UVM 20
2.6.1. Clases de la librería UVM 20
2.6.2. Las macros en UVM 22
2.6.3. El mecanismo de *Factory* 22
2.6.4. El mecanismo de mensajes 24
2.6.5. El mecanismo de configuración 26
2.6.6. Las fases de UVM 27
2.6.7. *Transaction Level Modeling* (TLM) 30
2.7. Transacciones, secuencias y componentes de UVM 33
2.7.1. Transacciones 34
2.7.2. Secuencias 35
2.7.3. Componente *sequencer* 37

2.7.4. Componentes *driver*, *monitor* y la interfaz *SystemVerilog*.....38
2.7.4.1. Componente *driver*.....39
2.7.4.2. Componente *monitor*.....41
2.7.5. Componente *agent*.....43
2.7.6. Componente *scoreboard*.....45
2.7.7. Componente *enviroment*.....48
2.7.8. El componente *test*.....50
2.7.9. El modulo *top*.....53
Capítulo 3 *SystemVerilog Direct Programming Interface*.....57
3.1. Introducción a los entornos multilenguaje.....57
3.2. Reutilización de código descrito en C en un entrono UVM.....58
3.3. La interfaz *SystemVerilog DPI*.....59
3.3.1. Los dominios de la interfaz *SystemVerilog* DIP.....60
3.3.2. Funciones/tareas importadas y exportadas.....61
3.3.2.1. Funciones/tareas importadas.....61
3.3.2.2. Funciones/tareas exportadas.....63
3.3.3. Funciones/tareas *pure* y *context*.....64
3.3.4. Restricciones en los tipos de datos.....65
3.3.5. Ejemplos de uso de la DPI.....67
Capítulo 4 Integración de un modelo de referencia descrito en C en un entorno UVM.....71
4.1. Introducción.....71
4.2. Arquitectura del componente *scoreboard*.....71
4.3. El Cifrado *Hill Cipher*.....74
4.3.1 Implementación en C del algoritmo *Hill Cipher*.....80
4.4. Entorno UVM generado para la integración del código C en el componente *predictor*.....85
4.4.1. Módulo DUV del *Hill Cipher*.....85
4.4.2. Transacciones.....86
4.4.3. Secuencias del módulo *Hill Cipher*.....88
4.4.4. La interfaz hillcipher_if.....91
4.4.5. El componente hillcipher_driver.....92
4.4.6. El componente hillcipher_monitor.....95
4.4.7. El componente hillcipher_agent.....100

4.4.8. El componente hillcipher_env 102
4.4.9. Arquitectura implementada del componente *scoreboard* 103
4.4.9.1. El componente hillcipher_scoreboard 105
4.4.9.2. El componente sb_predictor y modelo de referencia descrito en C 106
4.4.9.2.1. Modelo de referencia del algoritmo de *Hill Cipher* 107
4.4.9.2.2. El componente sb_predictor 113
4.4.9.2.3. El componente sb_comparator 116
4.4.10. Componente duv_env 122
4.4.11. Objeto de configuración hillcipher_sequence_config 124
4.4.12. La librería test_library 125
4.4.13. El módulo *top* 127
Capítulo 5 Resultados 129
5.1. Introducción 129
5.2. La interfaz DPI con la herramienta *QuestaSim* 129
5.2.1. Flujo de uso DPI en la herramienta *QuestaSim* 130
5.3. Resultados de la simulación 133
5.3.1. Resultados de la simulación por ventana de comandos 133
5.4.2. Resultados de la simulación mediante cronogramas 140
Capítulo 6 Conclusiones y líneas futuras 147
6.1. Conclusiones generales 147
6.2. Líneas futuras 149
Bibliografía 151
Parte II Pliego de condiciones 155
PC1. Recursos generales 157
PC2. Recursos hardware 157
PC3. Recursos *software* 157
Parte III Presupuesto 159
Presupuesto 161
PR1. Recursos humanos 161
PR2. Recursos hardware 161
PR3. Recursos software 162
PR4. Material Fungible 163
PR5. Coste total del proyecto 163

Parte IV Anexos 165
Anexo I – hillcipher_sequencer.sv 167
Anexo II – Resultados de la simulación por ventana de comandos 169
Anexo III – Contenido del CD-ROM 177

Índice de figuras

Figura 1.1: Porcentaje de tiempo y recursos dedicados a la verificación en proyectos ASIC / IC y FPGA, respectivamente. 3

Figura 2.1: Modelo básico de *testbench*. 12
Figura 2.2: Numero de ingenieros por proyecto ASIC/IC. 13
Figura 2.3: Adopción de los lenguajes de verificación. 15
Figura 2.4: Evolución de las metodologías de verificación y contribución de las empresas EDA. 17
Figura 2.5: Adopción de metodologías de verificación. 18
Figura 2.6: Modelo de un *testbench* basado en la metodología UVM. 19
Figura 2.7: Jerarquía parcial de clases UVM. 21
Figura 2.8: Fases de un componente UVM. 29
Figura 2.9 : Comunicacion tipo TLM *port-export*. 32
Figura 2.10 : Ejemplo de comunicación TLM *analysis port-analysis export.* ..33

Figura 4.1: Arquitectura del componente *scoreboard*, compuesta de un componente *predictor* y un componente *comparator.* 73
Figura 4.2: Cifrado simétrico. 75
Figura 4.3: Salida por consola del programa *Hill cipher* en C 84
Figura 4.4: Composición de una secuencia *Hill Cipher*. 88
Figura 4.5: Arquitectura implementada del componente *scoreboard*. 103
Figura 4.6: Ejemplo de solución para implementar dos uvm_analysys_imp_drv en un mismo componente. 104

Figura 5.1: Flujo de uso DPI en la herramienta *QuestaSim* de *Mentor Graphics*. 131
Figura 5.2: Archivos del entorno desarrollado empleados en la simulación. 133
Figura 5.3: Cronograma del resultado obtenido de la simulación, correspondiente a la primera secuencia de transacciones. 141
Figura 5.4: Cronograma del resultado obtenido de la simulación, correspondiente a la primera secuencia de transacciones. 142
Figura 5.5: Cronograma del resultado obtenido de la simulación, correspondiente a la segunda secuencia de valores. 143
Figura 5.6: Cronograma del resultado obtenido de la simulación, correspondiente a la segunda secuencia de valores. 144

Índice de códigos

Código 2.1: Código ejemplo de una transacción UVM. 34
Código 2.2: Código ejemplo de una secuencia UVM. 36
Código 2.3 : Código ejemplo de *sequencer* UVM. 37
Código 2.4 : Código ejemplo de creación de un *sequencer* mediante typedef. 38
Código 2.5 : Código ejemplo de una interfaz *SytemVerilog*. 39
Código 2.6 : Código ejemplo de un componente *driver* UVM. 40
Código 2.7 : Código ejemplo de un componente *monitor* UVM. 42
Código 2.8 : Código ejemplo de un componente *agent* UVM. 44
Código 2.9 : Código ejemplo de un componente *scoreboard* UVM. 46
Código 2.10: Código ejemplo de un *enviroment* UVM. 49
Código 2.11 : Código ejemplo de un clase *test* UVM, empleada como base de casos de *test*. 51
Código 2.12 : Código ejemplo de un caso de *test* UVM. 52
Código 2.13 : Código ejemplo de un módulo *top* de UVM. 54

Código 3.1: Ejemplo de importación DPI en el código *SystemVerilog*. 67
Código 3.2: Ejemplo de importación DPI en el código C. 68
Código 3.3: Ejemplo de una exportación DPI en el código *SystemVerilog*. 68
Código 3.4: Ejemplo de una exportación DPI en el código C. 68

Código 4.1: Programa escrito en C del cifrado *Hill Cipher*. 82
Código 4.2: *Package* UVM con las clase que conforman el entorno implementado. 85
Código 4.3: Implementación del modulo hillcipher (DUV). 86
Código 4.4: Implementación de la clase data_packet. 87
Código 4.5: Fichero de configuración para la secuencia *Hill Cipher* 88
Código 4.6: Implementación de la clase hillcipher_sequence. 90
Código 4.7: Implementación de la interfaz hillcipher_if. 92
Código 4.8: Implementación del componente hillcipher_driver. 94
Código 4.9: Implementación del componente hillcipher_monitor. 97
Código 4.10: Implementación del componente hillcipher_agent. 101
Código 4.11: Implementación del componente hillcipher_env. 102
Código 4.12: Implementación del componente hillcipher_scoreboard. 105
Código 4.13: Implementación del modelo de referencia, descrito en C, empleado para realizar las predicciones. 111
Código 4.14: Implementación del componente sb_predictor. 115
Código 4.15: Implementación del componente sb_comparator. 119
Código 4.16: Implementación del componente duv_env. 123

Código 4.17: Implementación del objeto de configuración hillcipher_sequence_config. 124
Código 4.18: Implementación de la librería test_library. 126
Código 4.19: Implementación del modulo *top.* 127

Código 5.1: Resultado de la simulación en dominio del codigo *SystemVerilog* del entorno UVM. 135
Código 5.2: Resultado de la simulación para la predicción de la primera secuencia en el modelo de referencia descrito en C. 136
Código 5.3: Comparaciones llevadas acabo en el componente *comparator* de la arquitectura *scoreboard* para la primera secuencia. 137
Código 5.4: Resultado de la simulación para la predicción de la segunda secuencia en el modelo de referencia descrito en C. 138
Código 5.5: Comparaciones llevadas acabo en el componente *comparator* de la arquitectura *scoreboard* para la segunda secuencia. 139
Código 5.6: Resultado final de la simulación. 140
Código 5.7: Salida por la ventana de comandos perteneciente a los mensajes del componente sb_predictor. 143
Código 5.8: Salida por la ventana de comando de los mensajes con los valores de las transacciones de la segunda secuencia. 144

Índice de tablas

Tabla 2.1: Valores de verbosity.....25
Tabla 2.2: Tipos de conexiones TLM, punto a punto, en UVM. 32

Tabla 3.1: Tipos de datos directamente compatibles entre SystemVerilog y C. 66

Tabla 4.1: Alfabeto a emplear formado por 26 letras para la aritmética modular 26.....76
Tabla 4.2: Equivalencia de los tipos empleados en la interfaz DPI.....108

Tabla PC1: Condiciones *hardware*. 157
Tabla PC2: Condiciones *software*.....157
Tabla PC3: Variables de entorno.....158

Tabla PR1: Costes asociados a los recursos hardware.....162
Tabla PR2: Costes asociados a los recursos software.....162
Tabla PR3: Costes asociados a los materiales fungibles. 163
Tabla PR5: Coste total del TFG.....163

Índice de ecuaciones

Ecuación 4.1. 77
Ecuación 4.2. 77
Ecuación 4.3. 78
Ecuación 4.4. 78
Ecuación 4.5. 79
Ecuación 4.6. 79
Ecuación 4.7. 79
Ecuación 4.8. 80
Ecuación 4.9. 80

Acrónimos

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AVM	Advance Verification Methodology
AVM	Advanced Verification Methodology
CD-ROM	Compact Disc-Read Only Memory
DPI	Direct Programming Interface
DUV	Device Under Verification
EDA	Electronic Design Automation
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
ERM	e Reuse Methodology
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
HVL	Hardware Verification Language
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IGIC	Impuesto General Indirecto Canario
IP	Intellectual property
ISP	Image Signal Processing
OOP	Object-Oriented Programming
OVM	Open Verification Methodology
PLI	Programming Language Interface

RAL	Register Abstraction Layer
RTL	Register Transaction Logic
SoC	Systems on Chip
SV	SystemVerilog
SV DPI	SystemVerilog Direct Programming Interface
TFG	Trabajo Fin de Grado
TLM	Transaction-Level Modeling
ULPGC	Universidad de Las Palmas de Gran Canaria
URM	Universal Reuse Methodology
UVC	Universal Verification Components
UVM	Universal Verification Methodology
VIP	Verification Intellectual Property
VMM	Verification Methodology Manual
VP	Verification Plan
VPI	Verilog Procedural Interface

Parte I

Memoria Descriptiva

Capítulo 1 Introducción

1.1. Introducción y antecedentes

Durante las últimas décadas los circuitos electrónicos han crecido en complejidad y en coste de producción, lo que ha obligado a los ingenieros a investigar y desarrollar nuevos métodos para la verificación de diseños electrónicos de manera más completa, detallada y eficiente. Las fases de verificación funcional están tomando cada vez más relevancia, debido al ahorro en costes-tiempos de la realización de cualquier diseño electrónico. Como muestra la Figura 1.1, en la actualidad las fases de verificación funcional pueden suponer hasta el 70% del tiempo de desarrollo y de los recursos asociados a un proyecto. Este dato pone de manifiesto la necesidad de usar metodologías avanzadas en el ámbito de la verificación funcional [1], [2].

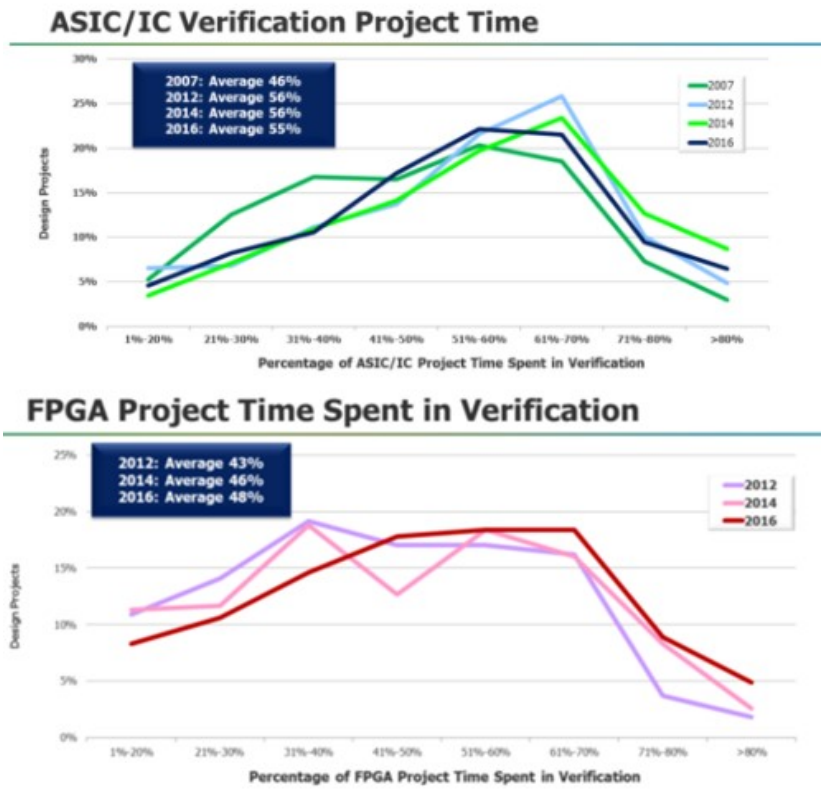

Figura 1.1: Porcentaje de tiempo y recursos dedicados a la verificación en proyectos ASIC / IC y FPGA, respectivamente.

Tradicionalmente, la verificación funcional de un sistema se ha realizado mediante un proceso *ad-hoc*, en el cual se obtenía el plan de verificación (VP, *Verification Plan*) a partir de las especificaciones funcionales del dispositivo bajo verificación (DUV, *Device Under Verification*). Cuando cada uno de los casos definidos en el VP ha sido verificado, confiando normalmente la toma de decisión en una inspección manual pero detallada de las formas de onda y de la selección del conjunto de vectores de entrada, el proceso de verificación funcional se da por completado. Los bancos de verificación, tradicionalmente descritos en leguajes HDL (*Hardware Description Language*), podían contener procesos capaces de leer vectores o comandos desde un archivo, emplearlos para modificar los valores de las líneas conectadas al DUV, y almacenar las señales de salida generadas por éste, así como volcarlos a otro archivo. No obstante, este proceso no se puede aplicar en sistemas complejos y, por tanto, no proporciona una verificación funcional segura [3].

Para hacer frente a la creciente complejidad de los sistemas actuales, la investigación en el ámbito de la verificación funcional ha logrado importantes progresos en el desarrollo de metodologías y herramientas. En cuanto a metodologías, si bien han habido aportaciones particulares por parte de diferentes empresas EDA (*Electronic Design Automation*) del sector (todas ellas de código cerrado), el principal avance en los últimos años es la creación de la metodología UVM.

UVM (*Universal Verification Methodology*) es una metodología estándar creada por Accellera, e introducida en la comunidad de usuarios en mayo de 2010, con el fin de fomentar la interoperabilidad de IPs de verificación (VIP, *Verification Intellectual Property*). Defendida, apoyada y soportada por las principales compañías EDA del sector, la metodología UVM permite incrementar la productividad eliminando la costosa interconexión que generalmente ralentiza la reutilización de los modelos de verificación. Esta metodología representa los últimos avances en cuanto a tecnologías de verificación, y está diseñada para facilitar una verificación funcional eficiente y exhaustiva [4].

UVM presenta las siguientes características y beneficios [5]:

- Es una tecnología madura, derivada de la metodología OVM (*Open Verification Methodology*), ofreciendo ahorro de tiempo y ampliando su cobertura gracias a la reusabilidad de los componentes de verificación.

- Proporciona librerías y metodologías de clases unificadas y abiertas para IPs de verificación (VIP) interoperables.

- Elimina la necesidad de interoperabilidad entre múltiples bibliotecas de verificación.

- Incorpora la capacidad de emplear entornos de verificación y sistemas de automatización.

- Es compatible con cualquier simulador que soporte el estándar IEEE 1800.

- Permite el uso de múltiples lenguajes de descripción y soporta VIP *plug and play*.

- Propicia la reusabilidad y escalabilidad de componentes y de entornos más complejos.

Además, la librería de la metodología UVM, está basada en el lenguaje *SystemVerilog* (SV), el cual soporta métodos de verificación avanzados, incluyendo el soporte a la programación orientada a objetos (OOP, *Object-Oriented Programming*), que proporciona un nivel de abstracción y una capacidad de reutilización superior en la etapa de verificación [3].

En muchos ámbitos, como puede ser el de procesamiento de imágenes, los algoritmos se desarrollan y se evalúan previamente utilizando modelos descritos en C antes de su implementación RTL (*Register Transfer Level*) [6]. Una vez finalizado el algoritmo, estos modelos en C se pueden reutilizar como modelos de referencia (*Golden Reference*) para la verificación de módulos y/o

sistemas. Dado que tanto la metodología UVM como la capacidad de las herramientas EDA soportan múltiples lenguajes de descripción, es posible plantear la integración de un modelo de referencia descrito en el lenguaje C para generar la respuesta esperada, y a partir de ésta compararla con la respuesta obtenida del DUV, implementado por lo general en lenguajes HDL [7].

1.2. Objetivos

El objetivo fundamental de este Trabajo Fin de Grado (TFG) consiste en la integración de un modelo de referencia de alto nivel, descrito en lenguaje C, en un entorno basado en la metodología UVM para la verificación continua de módulos o sistemas, haciendo uso de la interfaz SystemVerilog DPI (*Direct Programming Interface*).

Para lograr alcanzar este objetivo fundamental, se han planificado los siguientes objetivos operativos o parciales:

- O1. Objetivo 1: Aprendizaje autónomo de la metodología de verificación UVM a partir de la guía de usuario del estándar.

- O2. Objetivo 2: Estudio básico del lenguaje *SystemVerilog* con el fin de entender la estructura de los componentes usados en la metodología UVM.

- O3. Objetivo 3: Estudio de la interfaz *SystemVerilog* DPI (*Direct Programming Interface*) para la integración de modelos de referencia descritos en C.

- O4. Objetivo 4: Estudio básico del entorno de simulación *QuestaSim de Mentor Graphics*.

- O5. Objetivo 5: Estudio del modelo de referencia descrito en lenguaje C y de las especificaciones funcionales del DUV, así como sus interfaces de comunicación.

- O6. Objetivo 6: Creación y/o adaptación de los componentes básicos del entorno de verificación y su integración.

- O7. Objetivo 7: Estudio y creación de estructuras básicas para el cálculo de la cobertura de verificación.

- O8. Objetivo 8: Creación de un entorno completo de verificación que incluya los bloques desarrollados en los puntos O6 y O7.

1.3. Peticionario

Tras haber superado las asignaturas especificadas en el Plan de Estudios 2010 de la titulación Grado en Ingeniería en Tecnologías de la Telecomunicación, impartida por la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), se solicita, como requisito indispensable para la obtención del título de Graduado en Ingeniería en Tecnologías de la Telecomunicación, el desarrollo, la redacción, la exposición y la defensa de un Trabajo Fin de Grado.

1.4. Organización de la memoria

El presente documento consta de cuatro apartados bien diferenciados: Memoria, Pliego de condiciones, Presupuesto y Anexos. A su vez, la memoria se estructura en 6 capítulos, y termina con la bibliografía empleada. El contenido de estos capítulos es el que se resume a continuación:

- *Capítulo1. Introducción.* Este capítulo recoge los antecedentes que han dado lugar a la realización de este TFG, así como los objetivos del mismo, el peticionario, y la estructura del documento.

- *Capítulo 2. Universal Verification Methodology.* Este capítulo presenta una introducción a la verificación funcional de sistemas digitales y a la investigación y desarrollo, de herramientas y metodologías que han dado lugar a la metodología UVM. Finalmente, se tratan los conceptos necesarios para la generación de entornos de verificación basados en esta metodología.

- *Capítulo 3*. *SystemVerilog Direct Programing Interface* (DPI). Este capítulo comienza con una introducción sobre los entornos de verificación multilenguaje. En segundo lugar introduce la interfaz *SystemVerilog* DPI, la cual permite interactuar código descrito en SV con código descrito en lenguaje C. Esta herramienta será fundamental para llevar a cabo la integración de código descrito en C en un entorno de verificación basado en la metodología UVM y descrito en SV.

- *Capítulo 4*. *Integración de un modelo de referencia descrito en C en un entorno de verificación UVM*. En este capitulo se procederá con una explicación del modelo de referencia empleado para la integración, correspondiente al cifrado de *Hill Cipher*. Seguidamente se detallará la arquitectura del componente *scoreboard*, el cual será el componente en el que se llevará a cabo la integración del modelo de referencia. Finalmente, se muestra el entorno generado en el que se ha integrado el código descrito en C para realizar las predicciones para la verificación funcional.

- *Capítulo 5. Resultados*. Este capítulo muestra los resultados obtenidos tras la ejecución de un *test* empleando la herramienta de simulación *QuestaSim de Mentor Graphics*, para mostrar los resultados de la

integración del modelo de referencia descrito en C en un entorno UVM. Además, se muestra el flujo DPI en el entorno *QuestaSim*.

- *Capítulo 6. Conclusiones y líneas futuras*. Tras haber completado los objetivos propuestos para este Trabajo Fin de Grado, en este capítulo se incluyen las conclusiones obtenidas, además de las líneas futuras asociadas a la realización del presente TFG.

Capítulo 2 *Universal Verification Methodology*

2.1. Introducción

En este capítulo se realiza inicialmente una introducción a la verificación funcional de sistemas digitales. Seguidamente se presenta una breve descripción de la evolución de las herramientas utilizadas en este ámbito. Finalmente, se abordará la metodología de verificación UVM (*Universal Verification Methodology*). Esta metodología es muy extensa, y hace uso de múltiples conceptos: programación orientada a objetos, *SystemVerilog*, etc. Todos estos aspectos dificultan su explicación de forma ordenada y completa. Además, alcanzar un grado de entendimiento mínimo para su uso básico requiere de un esfuerzo inicial considerable, principalmente por dos motivos:

- En primer lugar, debido a las novedades que aporta la verificación funcional basada en la metodología UVM.

- En segundo lugar, por las dificultades que introduce el lenguaje *SystemVerilog*, en el que se fundamenta esta metodología.

En la organización del presente capítulo se ha procurado realizar una explicación lo más secuencial posible de la metodología. Sin embargo, y debido a las dependencias transversales de los conceptos utilizados, en determinados momentos las explicaciones se deberán realizar por capas, dado que algunos mecanismos pueden interactuar con otros, dificultando su explicación ordenada.

2.2. Introducción a la verificación funcional de sistemas digitales

La verificación funcional es uno de los procesos más importantes en el desarrollo de los sistemas digitales. El proceso de verificación funcional es aquel que determina el correcto funcionamiento de un sistema, asegurando que éste cumple con las especificaciones de diseño y opera adecuadamente, cumpliendo así con éxito el objetivo para el que ha sido diseñado [8].

El término *testbench* hace referencia al código de simulación empleado para generar un escenario, en el cual poder llevar a cabo el proceso de verificación funcional. Es decir, conforma el entorno que alberga todos los componentes necesarios para verificar el DUV (*Design Under Verification*). En dicho escenario, mostrado en su forma más simple en la Figura 2.1, se generan estímulos mediante vectores de test, que estarán dirigidos al DUV, para posteriormente realizar comparativas con las salidas esperadas asegurando de esta forma que su funcionamiento es el esperado, y que por tanto cumple con las especificaciones de diseño [9].

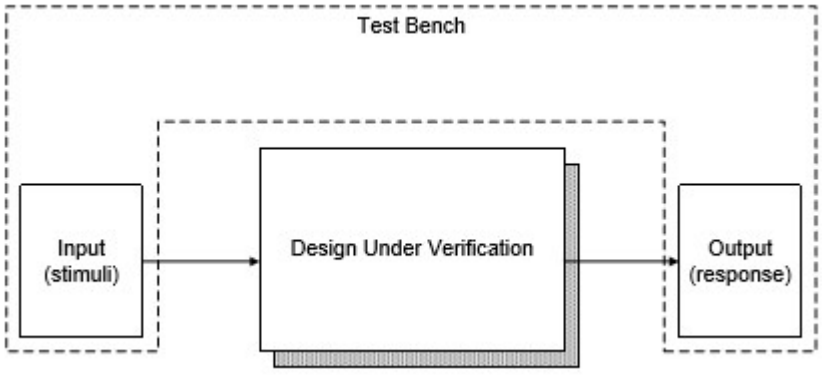

Figura 2.1: Modelo básico de *testbench*.

Hoy en día la capacidad de verificación continúa siendo inferior a la capacidad de diseño. Esta diferencia puede deberse a la curva de aprendizaje que supone la formación para el proceso de verificación. Esta complejidad exige diversas habilidades como la comprensión de diferentes arquitecturas, el análisis, la depuración y el aprendizaje de diversos lenguajes de programación, lenguajes HDL y lenguajes HVL (*Hardware Verification Language*), así como

metodologías y procedimientos de verificación, todas ellas necesarias para codificar las mejores prácticas para una verificación eficaz y exhaustiva mediante la creación de componentes IP de verificación (VIP) y *testbenches* modulares, reutilizables e interoperables.

La Figura 2.2 muestra el aumento en la necesidad de ingenieros de verificación frente a los ingenieros de diseño para proyectos ASIC/IC. Se puede observar cómo en la actualidad el número de ingenieros de verificación ha superado a los de diseño [2].

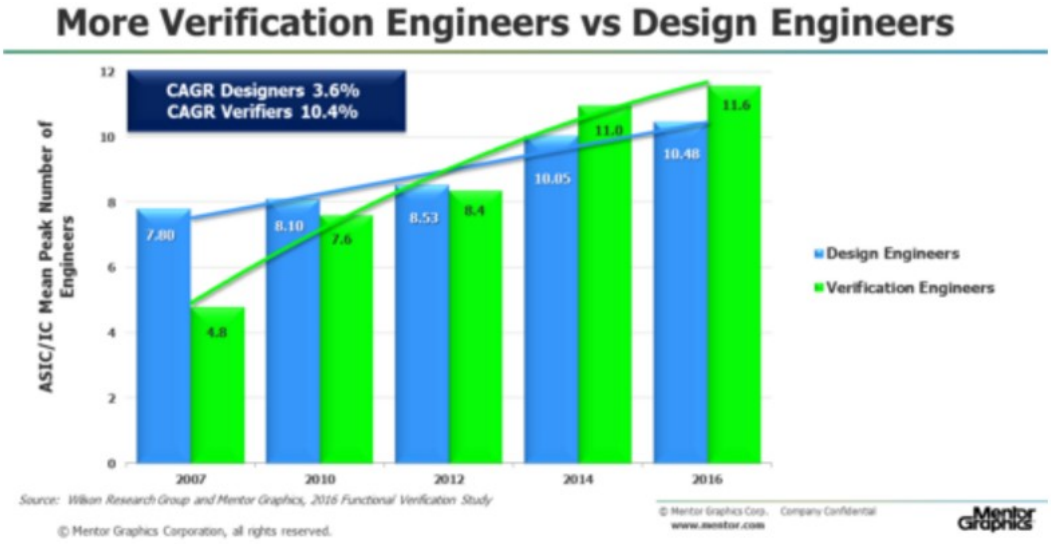

Figura 2.2: Numero de ingenieros por proyecto ASIC/IC.

Este dato pone de manifiesto la importancia que supone la verificación funcional, la cual reduce dos de los factores más importantes en un proyecto, como son, el coste y el tiempo, permitiendo obtener diseños que cumplen con sus especificaciones funcionales. Dos de los aspectos más importantes para este propósito son, la interoperabilidad y la reutilización. Los entornos de verificación se diseñan de forma que sean modulares y reutilizables, lo que permite su reutilización entre proyectos. La falta de entornos de verificación flexibles que permitieran la reutilización de los componentes de verificación a través de los proyectos de diseño electrónico, ha sido uno de los principales causantes del incremento en el coste en verificación.

Las mejoras en el proceso de la verificación funcional de los sistemas actuales, tratan de:

- Acelerar la verificación y la obtención de resultados.
- Reducir la relación coste-tiempo del proyecto.
- Obtener diseños de verificación reutilizables.
- Poder realizar el seguimiento del progreso de verificación (cobertura funcional).
- Desarrollar *testbenches* auto comprobables.

2.3. Lenguajes de verificación hardware

Los lenguajes HVL proporcionan el conjunto de herramientas orientadas a llevar a cabo el proceso de verificación funcional. Estos lenguajes aparecieron para mejorar las aportaciones que permitían realizar los lenguajes HDL, que como se comentó son aplicables a diseños relativamente sencillos y conforme el diseño aumente en complejidad se vuelven inviables. Los primeros lenguajes HVL como *Vera* y *e*, eran lenguajes orientados a objetos diseñado para la creación de *testbenches*. Cabe destacar que estos lenguajes eran propiedad de las empresas, y cada empresa tenía su propio lenguaje, por lo que la reutilización en cuanto a conocimientos, trabajo y metodologías era escaso. Las empresas de este ámbito, conscientes del problema, se involucraron haciendo libres varios de estos lenguajes y añadiendo sus características sobre el lenguaje *Verilog*, dando lugar a un nuevo lenguaje, denominado *SystemVerilog*. *SystemVerilog* soporta una variedad de operadores y estructuras de datos, así como variables aleatorias restringidas, cobertura funcional y aserciones temporales. En la actualidad, las herramientas que soportan *SystemVerilog* se han convertido en una de las opciones más atractivas a la hora de decantarse por un lenguaje de verificación o HVL. En la gráfica de la Figura 2.3 se muestra la evolución en la

adopción de *SystemVerilog* durante los últimos años, destacando como la opción de mayor incremento de adopción [2].

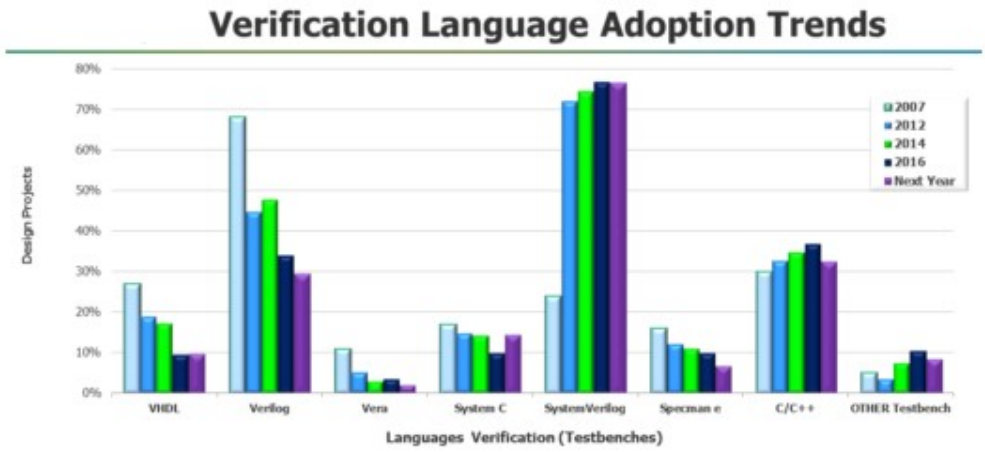

Figura 2.3: Adopción de los lenguajes de verificación.

2.3.1 El lenguaje *SystemVerilog*

SystemVerilog se consolidó como un estándar en el año 2013 bajo el nombre IEEE 1800-2012 y es una actualización de *Verilog*-2001, si bien amplía el ámbito de éste, originalmente enfocado únicamente en la descripción de hardware, hacia la verificación y la escritura de *testbenches*. El propósito era el de definir un lenguaje unificado para diseño y verificación, pudiendo usar solo una herramienta de simulación [10].

Esta actualización introduce 97 nuevas palabras reservadas y 31 secciones del Manual de Referencia del Lenguaje, segmentado principalmente en:

Diseño:

- Constructores de lenguaje.

- Tipos de datos estáticos.
- Interfaces de puertos.
- Dominios de reloj.
- Mejoras del lenguaje para síntesis.

Verificación:

- Procesos y datos dinámicos.
- Aserciones.
- Aleatorización y restricciones.
- Cobertura.

Orientación a objetos:

- Clases y clases parametrizadas.
- Herencia.
- Encapsulación.
- Agregación.
- Polimorfismo.

Interfaz con C

- *Direct Programming Interface* (DPI).

2.4. Evolución de las metodologías de verificación

La adopción de lenguajes HVL facilitó el proceso de verificación, pero no hubo consenso en el uso apropiado de un lenguaje de verificación y/o una metodología unificada de verificación. La Figura 2.4 representa la evolución en el uso de diferentes metodologías de verificación. Esta evolución ha sido posible gracias al esfuerzo y contribución de la experiencia de las principales empresas EDA del sector [4].

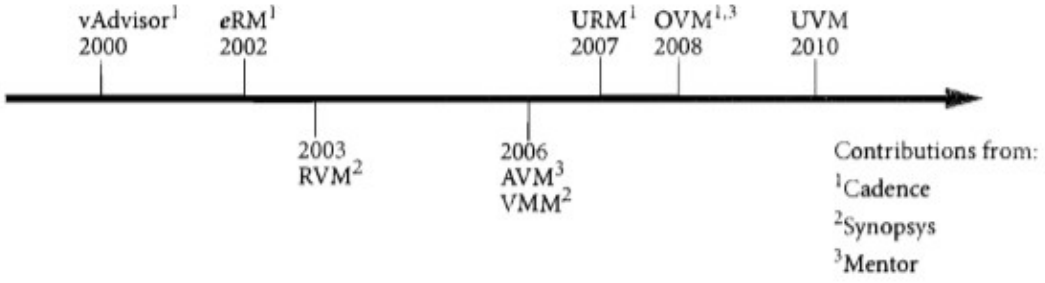

Figura 2.4: Evolución de las metodologías de verificación y contribución de las empresas EDA.

La metodología de verificación universal se desarrolló por la necesidad de los ingenieros de verificación de disponer de un marco de verificación unificado para permitir el desarrollo y la reutilización eficientes de los entornos de verificación y de las IP de verificación (VIP). Los desarrolladores de herramientas de simulación tenían sus propias metodologías, que no eran directamente compatibles entre sí. Para solventar todos estos inconvenientes, la organización de estándares para la automatización del diseño electrónico (EDA) y la fabricación de IC, *Accellera Systems Initiative*, decidió en 2010 establecer la metodología *Universal Verification Methodology* (UVM). Las metodologías de verificación de *Cadence* y *Mentor* contribuyeron como base para el desarrollo y generación de la metodología UVM. UVM se estableció en 2010 y continúa desarrollándose con la cooperación entre varias empresas de la industria de la verificación, como son: *Mentor Graphics*, *Synopsys*, *Cadence* y *Aldec*.

La metodología UVM se basa en la OVM (OVM, *Open Verification Methodology*) que es la combinación de la AVM *(AVM, Advance Verification*

Methodology) y la URM (URM, *Universal Reuse Methodology*). UVM tiene también algunos conceptos de la ERM (ERM, *e Reuse Methodology*) y algunos conceptos y código del VMM (VMM, *Verification Methodology*). Todas estas aportaciones lo han convertido en un estándar universal y, desde su aparición, es la metodología de verificación más empleada para la realización de *testbenches*. La Figura 2.5 muestra el incremento de adopción y uso de la metodología UVM [2].

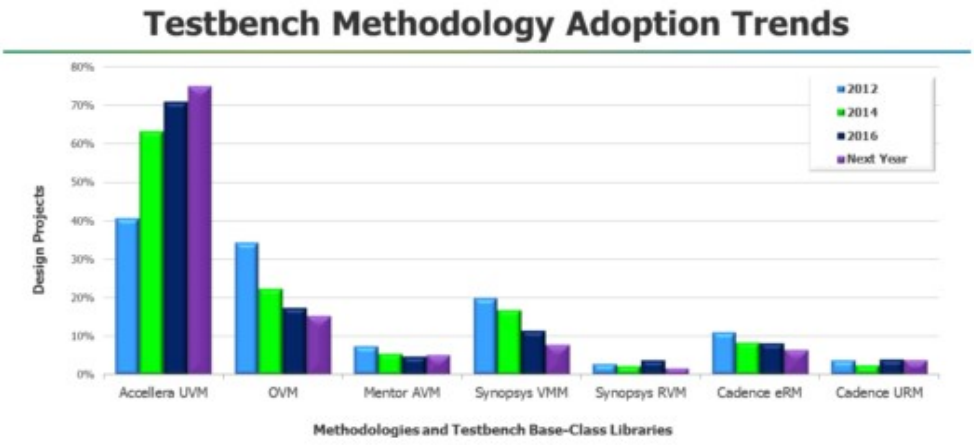

Figura 2.5: Adopción de metodologías de verificación.

2.5. La metodología UVM

La arquitectura de un *testbench* basado en la metodología UVM aporta todos los conocimientos y mejoras en el ámbito de la verificación funcional. Para ello se diseñan y construyen de forma modular para permitir la reutilización de un conjunto de componentes de verificación en diferentes proyectos, relacionado con la reutilización horizontal o, en un nivel más alto de integración dentro de un mismo proyecto, propiciando una reutilización vertical.

La Figura 2.6 representa un modelo de referencia de un *testbench* UVM para realizar la verificación funcional de un DUV [3].

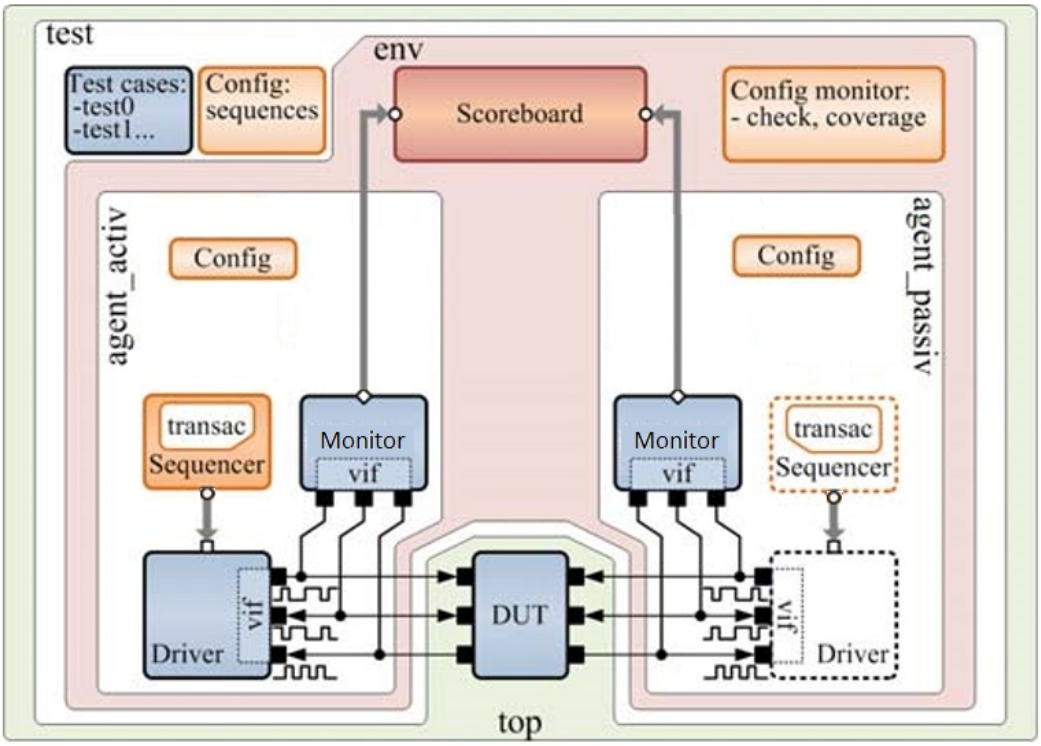

Figura 2.6: Modelo de un *testbench* basado en la metodología UVM.

La Figura 2.6 muestra diferentes niveles de jerarquía, empezando desde el módulo *top*, correspondiente al nivel de mayor jerarquía, el cual contiene un objeto de configuración, una librería de clases, *test cases* y el componente *enviroment*. El segundo nivel, considerado como el nivel superior de reutilización, el *enviroment*, contiene a su vez varios componentes *agent*, elementos de configuración y el componente *scoreboard*. Finalmente, se encuentra el último nivel, representado por el componente *agent*, también conocido como Componente de Verificación Universal (UVC, *Universal Verification Component*). Un UVC, es un componente de verificación encapsulado y configurado orientado a interactuar con un protocolo de interfaz específico. Cada UVC está formado por una arquitectura consistente y alberga todos los componentes (*driver*, *monitor* y *sequencer*) necesarios para interactuar con el DUV. Su relación consiste en:

- Generar y enviar estímulos.
- Recolectar información para realizar la verificación funcional.

Además, mediante un elemento de configuración, un UVC puede configurarse de acuerdo a su función dentro del entorno, generando de esta forma topologías de *testbenches* reutilizables y escalables.

2.6. Librería UVM

La metodología UVM proporciona todos los mecanismos y facilidades necesarias para agilizar el proceso de verificación funcional, proporcionando un flujo de trabajo común que aporta consistencia [4]. Para ello, la metodología UVM proporciona una librería de software libre descrita en *SystemVerilog.* Esta librería contiene una descripción de clases bases, utilidades y macros.

2.6.1. Clases de la librería UVM

La metodología UVM proporciona una librería de clases base descritas en *SystemVerilog*, la cual proporciona todos los bloques que se necesitan para un desarrollo ágil de componentes de verificación y entornos de test, bien construidos y reutilizables [4]. Estas clases, proporcionan automatización y uniformidad, y por tanto reutilización. Estas características se consiguen mediante el uso de métodos y campos predefinidos que el diseñador puede aprovechar para extender y personalizar mediante los mecanismos de herencia, o empleando el recurso de la *Factory*. En resumen, aprovechando las ventajas de la programación orientada a objetos (OOP) que soporta SV.

En la Figura 2.7 se muestra una vista parcial de la jerarquía de clases UVM. Las clases representadas en color gris son las clases base de las que el diseñador hace uso para obtener las ventajas comentadas. Seguidamente se resumen las clases base empleadas para modelar la jerarquía y la estructura de datos.

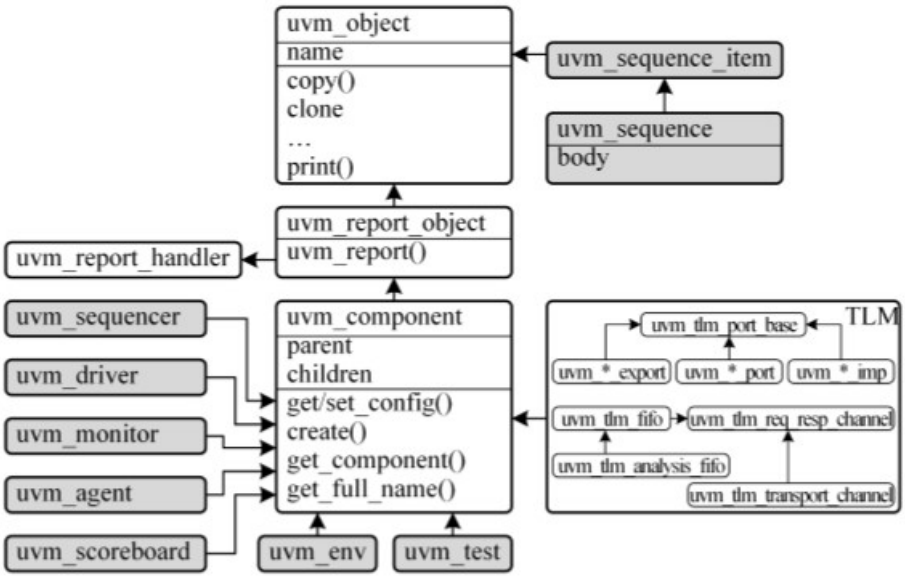

Figura 2.7: Jerarquía parcial de clases UVM.

- La clase `uvm_object` es una clase abstracta empleada para definir y automatizar un conjunto de métodos de operaciones de propósito común al resto de clases, clases para estructuras de datos, y clases jerárquicas.

- La clase `uvm_sequence_item` deriva directamente de la clase *`uvm_object`* y no representa un componente, se utiliza para la generación y análisis de estímulos. Esta clase está relacionada con los datos, elementos dinámicos del *tesbench*.

- La clase `uvm_report_object` proporciona un mecanismo de mensajes para la generación de informes, errores, etc. Todos los componentes `uvm_component` se derivan de esta clase heredando dicho mecanismo de mensajes.

- La clase *`uvm_component`* es la clase base común para los componentes, y está orientada a construir un *tesbench* jerárquico. Los componentes del *testbench* son elementos casi-estáticos, es decir, son creados dinámicamente pero permanecen estáticos durante la

simulación [4], permitiendo así establecer la jerarquía estructural a través de un conjunto definido de fases para la simulación.

- La clases `TLM` se emplean para representar las comunicación entre los componentes que necesitan intercambiar estructuras de datos, con mayor nivel de abstracción, en el entorno de verificación. La comunicación TLM se aborda en el apartado 2.6.7 de este capítulo.

2.6.2. Las macros en UVM

Las macros en UVM permiten la generación automatizada de código. Esto facilita las tareas rutinarias de codificación que deben realizarse al crear un elemento o componente a partir de las clases de UVM. Estas macros proporcionan automatización y son necesarias para poder hacer uso del mecanismo de *Factory*.

2.6.3. El mecanismo de *Factory*

La *Factory* es un patrón clásico de diseño software usado para crear código genérico. La metodología UVM hace uso del soporte OOP que proporciona SV para añadir un patrón de diseño *Factory* a su librería, permitiendo realizar entornos de verificación dinámicos y flexibles [12]. Para poder hacer uso de las utilidades y automatización de campos de las clases que aporta la *Factory*, UVM facilita macros parametrizables que permiten registrar un objeto o un componente mediante un identificador de tipo de objeto en la *Factory*. Las macros de utilidades son las siguientes:

Registro de un objeto en la *Factory*:

```
`uvm_object _utils(object_name)
```

Registro de un componente en la *Factory*:

```
`uvm_component_utils(compt_name)
```

Si, además de las utilidades, se desea disponer de un procedimiento para la automatización de campos, en tal caso se debe registrar la clase y sus campos, empleando las siguientes macros:

Macros para registrar objetos y sus campos en la *Factory*:

```
`uvm_object_utils_begin(object_name)
`uvm_field_*(field_name, UVM_flag)
`uvm_object_utils_end
```

Macros para registrar componentes y sus campos en la *Factory*:

```
`uvm_component_utils_begin(component_name)
`uvm_field_*(field_name, UVM_flag)
`uvm_component_utils_end
```

La macro `uvm_field_field_*` proporciona la implementación de métodos de uso común, evitando así tener que reescribirlos. El grado de automatización se lleva a cabo en función del valor de la variable `UVM_flag`, que se pasa como parámetro a la macro. El registro del tipo de objeto en la *Factoy* permite hacer uso del método `create()`, para reemplazar el constructor `new()` con el nuevo acceso`::type id::create()`. Este acceso permite realizar sustituciones de tipo en tiempo de ejecución, proporcionando una potente característica para la realización de test dinámicos [12]. Esto es así porque la principal diferencia entre el constructor `new()`y la función `::type id::create()` es que el primero crea componentes o transacciones de tipo fijo, mientras que el segundo crea objetos de `type id` permanentes almacenados en la *Factory*. Por último, la *Factory*, aprovechando

que tiene los objetos registrados, permite que a todo objeto del *testbench* se le pueda cambiar o sobrescribir el tipo sin necesidad de realizar modificaciones sobre su estructura. Para ello se emplean los métodos que se exponen a continuación. El primero se emplea para reemplazar un tipo de objeto (o componente) y el segundo para reemplazar un componente específico.

```
set_type_override_by_type(orig_type, override_type, bit
replace =1);
```

```
set_inst_override_by_type(String inst_path, orig_type,
override_type);
```

2.6.4. El mecanismo de mensajes

A la hora de desarrollar o utilizar entornos de verificación, se plantea la necesidad de disponer de un mecanismo de mensajes de información y/o de error. Para ello UVM proporciona un mecanismo que permite mostrar información y observar diferentes trazas de mensajes en cada una de las fases, tanto de construcción como de ejecución del entorno. El mecanismo de mensajes está disponible en todos los componentes pues, como se comentó, para este propósito heredan de la clase `uvm_report_object`. Cabe destacar que también se soporta en módulos e interfaces, descritos en *SystemVerilog*. El mecanismo de mensajes se gestiona con el uso de las cuatro macros siguientes:

```
`uvm info(string id, string message, int verbosity)
```

```
`uvm warning(string id, string message)
```

```
`uvm error(string id, string message)
```

```
`uvm fatal(string id, string message)
```

El primer argumento de las macros, `string id`, es una etiqueta empleada como filtro. Por norma general, se le asigna el nombre del componente que imprime el mensaje y permite, de este modo, que el diseñador o usuario del *testbench* pueda observar cuántos mensajes han sido enviados con dicha etiqueta una vez finalizado el test. El segundo argumento corresponde al mensaje con la información a imprimir, muy útil para realizar las fases de depuración, tanto del entorno como del código. Por último, la macro `` `uvm_info ``, a diferencia de las otras 3 macros, tiene un parámetro, denominado `verbosity`. Este parámetro indica el nivel de relevancia del mensaje y dependiendo del valor asignado a la variable global, `verbosity`, del entorno de verificación, el mensaje, se mostrará o no. Por ejemplo, si se le asigna a la variable `verbosity` el valor 100, todos los mensajes a los que se le haya asignado de manera independiente un valor por encima de 100 no serán mostrados [13]. Los valores que puede tomar esta variable se representan en la Tabla 2.1.

Verbosity	Valor
UVM_NONE	0
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

Tabla 2.1: Valores de `verbosity`.

De esta forma, si se asigna a la variable global `verbosity` el valor `UVM_DEBUG`, se mostrarán todos los mensajes introducidos en el entorno, mientras que si se le asigna el valor `UVM_NONE`, solo se mostrarán aquellos mensajes cuya variable `verbosity` sea `UVM_NONE`. UVM permite poder cambiar el nivel de relevancia de los mensajes de dos formas diferentes:

- Mediante la llamada a la función:

  `set_report_verbosity_level(verbosity).`

- Mediante la línea de comandos, usando el argumento `+UVM_VERBOSITY=UVM_Verbosity`.

Cabe destacar que la modificación del nivel de `verbosity` mediante línea de comandos no implica volver a compilar el diseño. Esto permite gestionar, de forma agilizada, el nivel de importancia deseado en el proceso de depuración.

2.6.5. El mecanismo de configuración

Un componente de verificación, para poder ser reutilizable y genérico, debe disponer de un mecanismo que permita su configuración para un modo de operación específico. Con tal propósito, la librería de la metodología UVM proporciona un mecanismo flexible que permite configurar los atributos y conformar la topologías del componente de forma dinámica, sin hacer uso de la *Factory*. Además, la metodología UVM permite que la configuración se pueda realizar desde diferentes niveles de la jerarquía, lo que aumenta la reusabilidad a cualquier nivel de la estructura de un *testbench* [14].

UVM proporciona la clase `uvm_config_db`, que contiene una interfaz simplificada de acceso a los objetos de configuración creados en una base de datos. Mediante los métodos `set()` y `get()`, el usuario puede leer o escribir los parámetros de configuración de un objeto determinado. Las funciones definidas en la clase `uvm_config_db` son estáticas y se llaman haciendo uso del operador "`::`", como se muestra a continuación.

La llamada al método `set()` permite definir un elemento en la base de datos:

```
uvm config db#(T)::set(context,inst name,field,value)
```

La llamada al método `get()` permite recuperar el elemento de la base de datos.

```
uvm config db#(T)::get(context,inst name,field,value)
```

El parámetro `T` indica el tipo de dato con el que se va a operar en la base de datos (`integer`, `string`, `object`). Los argumentos de los métodos son:

- `Context`, generalmente se utiliza el identificador `this` (operador común en la programación orientada a objetos).
- La ruta jerárquica desde el contexto hasta el nombre que se le dio al componente (`inst name`) sobre el que se desea actuar. Normalmente se especifica usando la sentencia: `context.inst_name`.
- Campo a configurar (`field`).
- Valor que se desea aplicar o variable sobre la que se va a leer el valor (`value`).

Esta estructura permite que los valores de configuración, de cualquier tipo de datos, incluidos los tipos definidos por el usuario, se almacenen en una tabla accesible globalmente y se recuperen posteriormente por componentes de nivel inferior de jerarquía. De esta manera, cada componente puede construir sus clases derivadas conforme a los datos de configuración especificados.

2.6.6. Las fases de UVM

Las clases en la jerarquía de UVM se dividen en dos categorías distintas: datos y componentes, cada una con un propósito diferente:

- Los objetos de datos, objetos dinámicos, pueden crearse en cualquier momento. Su objetivo es modelar el estímulo y las transacciones observadas, es decir, los datos que fluyen en el *testbench*. Los objetos

de datos, por lo tanto, no se organizan en estructuras predeterminadas por las fases.

- A diferencia de los objetos, los componentes están diseñados para modelar partes permanentes y estructurales del *testbench*.

Para tal propósito, y para obtener un flujo de ejecución consistente cuando se diseña un entorno de verificación, UVM aporta un conjunto de fases que permiten ordenar los principales pasos que tienen lugar durante la simulación. Estas fases se dividen en tres grandes grupos [11]:

- *Built-Phases*. En estas fases se configuran, construyen y conectan los componentes y los puertos implementados en éstos.

- *Run-Phases*. Estas fases son las únicas que consumen tiempo, por lo tanto se implementan como tareas, y es donde se lleva a cabo la ejecución del *testcase*.

- *CleanUp_Phases*. Estas fases recogen los resultados de la ejecución del *testcase* y generan informes de verificación.

Los diferentes grupos de fases se ilustran en el Figura 2.8, y seguidamente se proporciona una explicación más detallada del proceso que se lleva a cabo en cada una de estas fases.

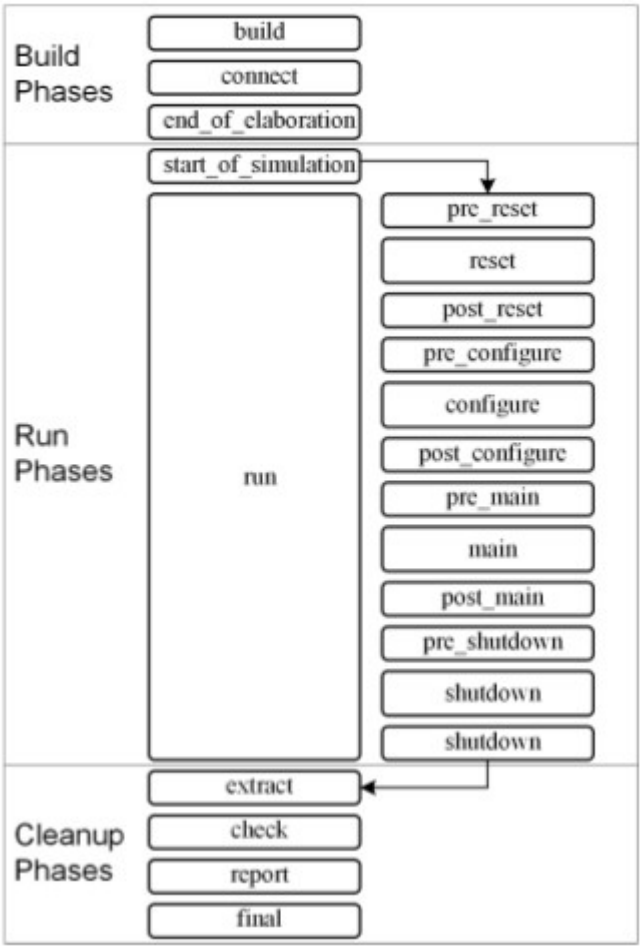

Figura 2.8: Fases de un componente UVM.

En primer lugar, la fase *build_phase* construye la jerarquía de componentes del *tesbench* de forma descendente, *top-down*, permitiendo que los componentes hijos o derivados, sean configurados por sus clases padres. La siguiente fase, denominada *connect_phase,* realiza la conexión de los componentes empleando puertos TLM. La fase *end_of_elaboration_phase* permite al usuario imprimir la estructura del *testbench*. Estas fases aseguran que el sistema se construya antes de la fase *run-phase*.

En la Figura 2.8 se puede observar que la fase *run_phase*, a su vez, está subdividida en otras fases, todas ellas involucradas principalmente en componentes orientados a la transacción. Dado que consumen tiempo de simulación, pueden hacer uso de un mecanismo de sincronización denominado *objections*. El mecanismo de *objections* se utiliza para determinar

cuándo ha finalizado un test. Permite a cada componente indicar la finalización de su fase de ejecución. La simulación sólo finalizará si los componentes han completado su fase *run_phase*. Este mecanismo se implementa mediante dos métodos:

- `raise_objection()`, empleado para señalizar el momento en que se inicia la ejecución de la fase *run_phase* en un componente, y.

- `drop_objection()`, empleado para señalizar el momento en que finaliza la fase *run_phase* en un componente.

La fase *run_phase* se da por finalizada cuando termina el método `drop_objection()` de todos los componentes que lo implementen y comienzan las fases de *clean_up: extract_phase, check_phase, report_phase* y *final_phase,* utilizadas para extraer los resultados de la verificación, determinando y mostrando el resultado general de la misma.

2.6.7. *Transaction Level Modeling* (TLM)

TLM responde a las siglas de *Transaction Level Modeling*, estándar de comunicaciones definido en SystemC y recogido en el documento IEEE 1666-2011. Actualmente coexisten dos estándares [15]:

- El estándar TLM-1.0. Es una API TLM de propósito general que está diseñada para modelar el paso de mensajes. En el paso de mensajes, los objetos (en este caso las transacciones) se pasan entre componentes de una forma similar a los paquetes que se envían en una red de datos. En el paso de mensajes, como en la transmisión de transacciones, no hay ningún estado compartido entre el remitente y el receptor, la única comunicación está contenida dentro de los mensajes que se pasan.

- El estándar TLM-2.0. Este estándar está diseñado para permitir el desarrollo de modelos de plataforma virtual de alta velocidad en *SystemC*. Está específicamente orientado al modelado de buses mapeados en memoria, y contiene muchas características que permiten la integración y reutilización de componentes que se conectan a los buses.

Los componentes de UVM se comunican a través de un estándar de conexiones TLM que les proporciona interoperabilidad. Esto es debido a que la semántica de paso de mensajes TLM está bien adaptada para el modelado a nivel de transacción, aportando un mayor nivel de abstracción para la comunicación entre los componentes del entono de verificación. Empleando una implementación de la comunicación TLM un componente puede comunicarse con cualquier otro componente que implemente dicha conexión. Cada interfaz TLM consiste en uno o más métodos para el intercambio de transacciones, y es la semántica común de las comunicaciones TLM la que permite intercambiar elementos sin afectar al resto del entorno.

Para las conexiones tipo TLM entre componentes la metodología UVM proporciona los puertos *port* y *export*:

- *Port*: un puerto tipo TLM *port* define un conjunto de métodos que se utilizarán para una conexión determinada.

- *Export*: un puerto tipo TLM *export* proporciona la implementación de los métodos llamados desde un puerto TLM *port*.

En la Figura 2.9 se muestra un ejemplo de conexión entre dos componentes, un productor y un consumidor, que utilizan una conexión *port-export*. Como se muestra en la imagen, el puerto *port* se representa con un cuadrado y el puerto *export* se representa con un círculo.

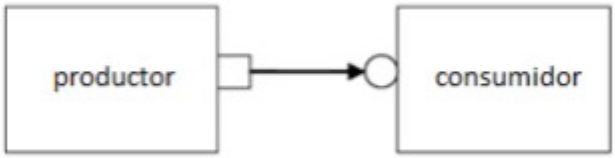

Figura 2.9 : Comunicacion tipo TLM *port-export*.

Para interconectar componentes mediante los puertos *port* y *export* se hace uso de la función `connect()`. En la Tabla 2.2 se pueden observar los tipos de conexión punto a punto y su implementación empleando los puertos *port/export.*

Tipo de conexión	Implementación
Port-to-export	`comp1.port.connect(comp2.export);`
Port-to-port	`subcomponent.port.connect(port);`
Export-to-export	`export.connect(subcomponent.export);`

Tabla 2.2: Tipos de conexiones TLM, punto a punto, en UVM.

Además, de las conexiones punto a punto, la metodología UVM proporciona puertos tipo TLM que permiten coneciones punto a multipunto. Estos puertos se denominan *analysis port* y *analysis export*:

- *Analysis port*: un puerto TLM *analysis port* es un puerto que se implementa mediante la función `write()`. Es decir, un componente llama a la función `write()`, empleando la sentencia `analysis_port.write()`. Esta función es una función `void` y su llamada siempre se completa en el mismo ciclo, es decir, sin consumir tiempo e independientemente de los componente que estén conectados al mismo.

- *Analysis export*: un puerto TLM *analysis export* proporciona la implementación del método `write()`. Este método, como se comentó, se llama desde el puerto tipo *analysis port*.

En la Figura 2.10 se muestra un ejemplo en el cual se emplea puertos tipo TLM, *analysis port* y *analysis export*. Estos puertos se representa con un rombo, en el caso del *analysis port* y con un circulo en el caso del *analysis export*.

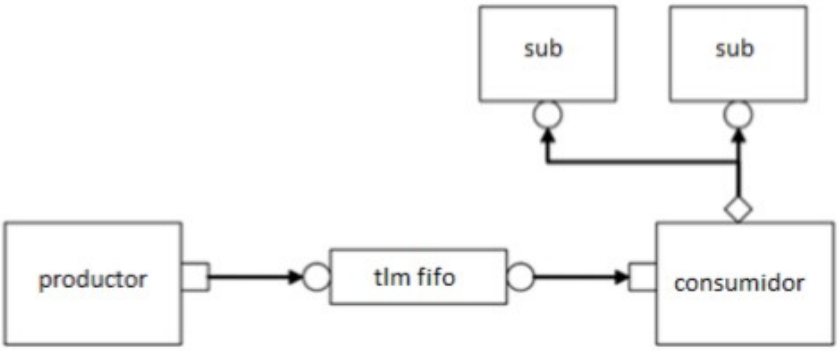

Figura 2.10 : Ejemplo de comunicación TLM *analysis port-analysis export.*

Por último cabe destacar que en determindas ocasiones se puede disponer de la necesidad de implementar dos funciones `write()` en un mismo componente *subscriber*. Para ello se deben declarar dos macros, una para cada implementación de cada funcion `write()`. Este caso se tratará en el componente *scoreboard* tratado en el Capítulo 4.

2.7. Transacciones, secuencias y componentes de UVM

A continuación, se procederá a dar una explicación de las principales clases, mostradas en gris en la Figura 2.7 de la jerarquía de clase UVM.

2.7.1. Transacciones

Los elementos tipo `sequence_items`, también conocidos como transacciones, son la unidad básica de comunicación en un *testbench*. Se crean a partir de la clase `uvm_sequence_item`, y encapsulan los campos y métodos necesarios para pasar información entre los componentes del entorno de verificación, empleándose para representar la comunicación con el DUV en un nivel de abstracción, permitiendo el uso de conexiones tipo TLM. En el Código 2.1 se muestra un ejemplo de la descripción de una transacción UVM, compuesta por los diferentes campos necesarios para modelar una transacción.

```
1  class my_transaction extends uvm_sequence_item;
2
3     rand bit [15:0] data_in;
4     rand bit [15:0] data_out;
5     rand bit [1:0]  cf;
6          bit        enable;
7     rand int        delay;
8
9     constraint timing {delay inside {[0:5]};}
10
11    `uvm_object_utils_begin(my_sequence_item)
12       `uvm_field_int(data_in, UVM_DEFAULT)
13       `uvm_field_int(data_out, UVM_DEFAULT)
14       `uvm_field_int(delay, UVM_DEFAULT)
15       `uvm_field_int(cf, UVM_DEFAULT)
16       `uvm_field_int(enable, UVM_DEFAULT)
17    `uvm_object_utils_end
18
19    function new(string name = "my_transaction");
20       super.new(name);
21    endfunction: new
22
23 endclass: my_transaction
```

Código 2.1: Código ejemplo de una transacción UVM.

Del código anterior y haciendo referencia a lo comentado en los apartados anteriores, cabe destacar:

- La línea 1 muestra cómo se crea la clase a partir de la clase `uvm_sequence_item`, aprovechando macros y métodos declarados en esta última.

- En la línea 9 se muestra un ejemplo del uso de una restricción de tiempo (*SystemVerilog*) `{delay inside {[0:5]};}`. En este ejemplo en concreto, se trata de restringir el valor aleatorio del campo `delay` dentro del rango deseado (entre 0 y 5).

- Entre las líneas 11 y 17 se realiza el registro en la *Factory* de la clase y de sus campos, empleando las macros comentadas anteriormente. Además, al registrar los campos en la *Factory* se obtiene directamente acceso a métodos tales como `copy()` y `clone()`, entre otros.

- Por último, las líneas entre la 19 y la 21 describen el constructor de la clase. Es un código estándar de la metodología que, obligatoriamente, se repite en el resto de las clases.

Una vez especificada la unidad básica de transferencia, `data_item`, se puede plantear la generación de tramas más complejas compuestas por un conjunto de ellas para formar una secuencia, siendo éste el rol de la clase `uvm_sequence`.

2.7.2. Secuencias

Las secuencias son múltiples transacciones generadas para representar un protocolo de comunicación que viene determinado por el DUV. La clase `sequence` se crea a partir de la clase base `uvm_sequence` para generar estímulos, es decir, proporciona la interfaz básica para crear un flujo de `data_items` y/u otras secuencias. Son objetos dinámicos y por lo tanto se puede generar una librería de secuencias y lanzarlas como estímulos durante toda la simulación. Esta práctica facilita la ejecución de varios *testbenches*, formado cada uno de ellos por una o un conjunto de secuencias. Este aspecto facilita enormemente la ejecución de un Plan de Verificación. El comportamiento de una secuencia se implementa dentro de la tarea `body()`, la cual es una tarea principal y propia de esta clase en la que se crean

transacciones y se les asignan valores a sus diferentes campos. El Código 2.2 describe un ejemplo de una secuencia.

```
1  class my_sequence extends uvm_sequence #( my_tx);
2
3     `uvm_object_utils( my_sequence)
4
5   //constructor
6     virtual task body( );
7        repeat(5) begin
8          req = my_tx::type_id::create("req");
9          start_item(req);
10          assert(req.randomize( ) enable { == 'b1;});
11         finish_item(req);
12       end
13    endtask: body
14 endclass: my _sequence
```

Código 2.2: Código ejemplo de una secuencia UVM.

Del Código 2.2 cabe destacar los siguientes aspectos:

- La línea 1 muestra la cabecera de la clase, creada a partir de la clase `uvm_sequence`. Esta clase está parametrizada con el tipo de dato, `sequence_item`, a emplear para generar las secuencias. Además del objeto `sequence`, todos los componentes del entorno que necesiten datos del tipo `sequence_item` deben ser parametrizados con este tipo.

- La línea 3 emplea la macro para registrar objetos en la *Factory*.

- Las línea entre la 6 y la 13 describen la tarea `body ()` que constituye la tarea principal de una secuencia.

- La línea 7 emplea un bucle `repeat` para crear 5 transacciones. En este caso en particular, la secuencia descrita está formada por 5 transacciones.

- La línea 8 referencia una transacción empleando el método `create()` proporcionado por la *Factory*.

- La línea 9 junto con la línea 11 son funciones heredadas de la clase `uvm_sequence` que permiten realizar un control de los protocolos de *handshake* utilizados para enviar las transacciones.

- La línea 10 asigna valores a los campos de la transacción que se enviarán al componente *driver*.

Cabe destacar que UVM proporciona dos macros encargadas de crear, aleatorizar y enviar las transacciones de una secuencia de forma automatizada, implementando implícitamente los métodos `start_item()` y `finish_item()`. Estas macros son:

`uvm do (crea, aleatoriza y envía)

`uvm do with (crea, aleatoriza, restringe campos y envía)

2.7.3. Componente *sequencer*

Un componente *sequencer* ejecuta secuencias y las envía al componente *driver*. Creado a partir de la clase base *uvm_sequencer*, proporciona un puerto de comunicación llamado `seq_item_export`, que permite su conexión con el componente *driver* mediante conexiones tipo TLM. El Código 2.3 muestra el ejemplo de un *sequencer* denominado `my_sequencer` creado a partir de la clase *uvm_sequencer* y parametrizada para trabajar con transacciones del tipo `my_tx`.

```
1  class my_sequencer extends uvm_sequencer #(my_tx);
2
3     `uvm_sequencer_utils(my_tx)
4
5     function new(string name, uvm_component parent);
6        super.new(name, parent);
7     endfunction: new
8  endclass: my_sequencer
```

Código 2.3 : Código ejemplo de *sequencer* UVM.

Como muestra el Código 2.3, un componente *sequencer* simplemente declara la clase y la parametriza con el tipo de transacción a gestionar, línea1 del código. En segundo lugar registra el componente en la *Factory*, línea 3, y entre las líneas 5 y 6 describe su constructor de una manera estándar. Sin embargo, en este código se introducen novedades no vistas hasta ahora:

- En la línea 3 se utiliza por primera vez la macro para registrar un componente `` `uvm_sequencer_utils ``.

- Aunque puede crearse como una clase derivada a partir de la clase `uvm_sequence`, la mayoría de las veces se puede crear simplemente mediante un `typedef`, como se describe en el Código 2.4.

```
typedef uvm_sequencer #(my_tx) my_sequencer;
```

Código 2.4 : Código ejemplo de creación de un *sequencer* mediante `typedef`.

La asociación de la secuencia con el componente *sequencer* se lleva a cabo en la clase `test`. Esta asociación se detallará más adelante en el apartado 2.7.8.

2.7.4. Componentes *driver*, *monitor* y la interfaz *SystemVerilog*

El *Bus Functional Model* (BFM) consiste en una abstracción para interactuar con el DUV. Representa un mecanismo para encapsular todas las señales del DUV de manera que los componentes del entorno de verificación, encargados de esta función, pueden comunicarse con el mismo. La complejidad de un BFM varía en función del diseño, dependiendo del protocolo de bus. Los componentes que necesitan implementar un BFM deben tener acceso a una interfaz de *SystemVerilog*. La solución propuesta en la metodología UVM es hacer uso de la *virtual interface* o interfaz virtual. Una *virtual interface* permite separar los modelos abstractos, basados en clases, de

aquellos componentes físicos y sus señales de interfaz. Este mecanismo permite que componentes como *driver* y *monitor* puedan acceder a las señales físicas del DUV a través de un conjunto de señales virtuales, definidas dentro de la `virtual interface`. El Código 2.5 muestra un ejemplo de interfaz *SystemVerilog*.

```
1  interface my_if(input logic clk, rst_n);
2
3     logic [1:0]  cf;
4     logic [15:0] data_in;
5     logic [15:0] data_out;
6     logic        enable;
7
8  endinterface: my_if
```

Código 2.5 : Código ejemplo de una interfaz *SytemVerilog*.

2.7.4.1. Componente *driver*

El componente *driver* es el encargado de recibir datos del componente *sequencer* en forma de transacción y transformarlos en señales RTL que el DUV pueda interpretar y gestionar. Además, como se comentó, un componente *driver* debe contener una interfaz virtual SV para conectarse con el DUV y poder estimularlo. Por lo tanto, la clase necesita tener un puntero a dicho objeto mediante una interfaz virtual. El componente *driver* configura la interfaz virtual utilizando el método estático `uvm_config_db::get()`. Este método fue comentado en el capítulo del mecanismo de configuración. Finalmente, el puerto tipo TLM heredado de la clase base `uvm_driver`, denominado `seq_item_port`, se emplea para comunicarse con el componente *sequencer*. El protocolo *Handshake* entre el componente *driver* y el componente *sequencer* se implementa mediante los métodos `get_next_item()` e `item_done()`, junto con los métodos `start_item()` y `finish_item()` definidos en el objeto *sequence*. El Código 2.5 muestra un ejemplo de descripción de un componente *driver*. Este componente se crea a partir de la clase base `uvm_driver` y se parametriza con el tipo de transacción a gestionar.

```
1  class my_driver extends uvm_driver #(my_tx);
2     virtual my_if  vif;
3
4     `uvm_component_utils(my_driver)
5
6     // constructor UVM
7
8    function void build_phase(uvm_phase phase);
9      super.build_phase(phase);
10    if(!uvm_config_db#(virtual my_if)::get(this, "", "in_intf", vif))
11    `uvm_fatal("NOVIF", {"vir_if must be set for: ", get_full_name( ),
12 ".vif"})
13       `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
14   endfunction
15
16   virtual task run_phase(uvm_phase phase);
17      forever begin
18         seq_item_port.get_next_item(req);
19         drive_packet(req);
20         seq_item_port.item_done( );
21      end
22   endtask
23   virtual task drive_packet(my_secuence_item tx);
24       // Depende del DUV
25   endtask
26
27 endclass: my_driver
```

Código 2.6 : Código ejemplo de un componente *driver* UVM.

Las principales características a destacar en el Código 2.6 son las siguientes:

- En la línea 2 se declara una interfaz virtual, necesaria para el acceso a la interfaz del DUV.

- La línea 4 realiza el registro del componente en la *Factory*.

- Las líneas entre la 8 y la 14 describe la fase `built_phase`. En esta fase, como se comentó anteriormente, se configura el componente *driver* mediante el uso del método `get()` de la base de datos. Además, usando el mecanismo de mensaje se genera un mensaje `` `uvm_fatal ``, indicando, que en caso de no poder acceder a la interfaza virtual, la existencia de un fallo grave o dar un mensaje simple `` `uvm_info ``, confirmando la configuración de la interfaz.

- Las líneas entre la 16 y 22 implementan la fase `run_phase` en su forma más básica. Ésta consiste de un bucle `forever` que solicita una transacción al componente *sequencer*, línea 11 del código, con la llamada a la función `get_next_item`. Una vez recibida ésta, la envía al DUV empleando la interfaz virtual y, una vez finalizada, avisa al componente *sequencer* que ha terminado de transmitir esa secuencia, usando la función `item_done`.

2.7.4.2. Componente *monitor*

El componente *monitor* se crea a partir de la clase base `uvm_monitor` y es un componente pasivo, es decir no conduce ninguna señal al DUV. Su propósito se basa en recolectar las transacciones a través de la interfaz virtual y exportalas a los componentes que implementen la función `write()` del puerto de análisis TLM declarado en el componente *monitor*. En IPs simples, la cobertura y la verificación se pueden realizar a este nivel, pero conforme la complejidad de IP aumente, la cobertura y la verificación se delegan a otros componentes diseñados para este propósito, como son el componente *scoreboard* y el componente *coverage*. El Código 2.7 describe un ejemplo de componente *monitor*.

```
1  class my_monitor extends uvm_monitor;
2     virtual my_if vif;
3     int num_pkts;
4
5     uvm_analysis_port #(my_tx) item_collected_port;
6     my_tx data_collected, data_clone;
7
8     `uvm_component_utils(my_monitor)
9
10    //constructor
11
12    function void build_phase(uvm_phase phase);
13      super.build_phase(phase);
14   if(!uvm_config_db#(virtual  pipe_if)::get(this,  "",  monitor_intf,
15 vif))
16    `uvm_fatal("NOVIF", {"VInterface must be set for: ", get_full_name( ),
17 ".vif"})
18
19     item_collected_port = new("item_collected_port", this);
```

```
20      data_collected = my_tx::type_id::create("data_collected");
21      data_clone = my_tx::type_id::create("data_clone");
22    endfunction: build_phase
23
24     virtual task run_phase(uvm_phase phase);
25        forever begin
26        // se recogen las entradas o salidas, conforme al protocolo
27        end
28     endtask: run_phase
29
30     virtual function void report_phase(uvm_phase phase);
31   `uvm_info(get_type_name(   ),   $psprintf("REPORT:C_PACKETS   =   %0d",
32  num_pkts), UVM_LOW)
33     endfunction: report_phase
34
35  endclass: my_monitor
```

Código 2.7 : Código ejemplo de un componente *monitor* UVM.

Las principales características a destacar en el Código 2.7 son las siguientes:

- La línea 1 muestra la cabecera de la clase, creada a partir de la clase `uvm_monitor`. Cabe destacar que esta clase no está parametrizada, y por lo tanto, no redefine ningún parámetro.

- La línea 2 declara una interfaz virtual, con el propósito de tener acceso a la interfaz del DUV.

- La línea 3 declara una variable, `num_pack`, que en este caso se emplea en la fase `report_phase` permitiendo mostrar el número de transacciones recolectadas por el componente *monitor*.

- La línea 5 declara un puerto tipo TLM, `uvm_analisis_port`, para comunicar con los componentes que implementen la función `write()`.

- La línea 6 declara dos transacciones de tipo `my_tx` empleadas para recolectar las señales de interés que debe obtener el componente *monitor* de la interfaz virtual para su posterior conversión en una transacción y su envío.

- La línea 8 registra el componente en la *Factory*.
- Las líneas entre la 12 y la 20 implementan la fase `built_fase`. En esta fase se configura la interfaz empleando el mismo mecanismo de configuración que se comentó en el componente *driver*. Además, se construye el puerto TLM y las dos transacciones empleadas para la recolección de datos.

- Las líneas entre la 22 y la 26 implementan la tarea `run_phase`. Esta fase modela el comportamiento del componente *monitor* y depende de la interfaz y protocolo del DUV.

- Finalmente, las línea 28 y la 31 implementan la fase `report_fase`, la cual emplea la variable `num_packet` y el mecanismo de mensajes para mostrar el número de transacciones recolectadas por el componente *monitor* al finalizar la simulación.

2.7.5. Componente *agent*

Un componente *agent*, también conocido como UVC, es un componente que se deriva de la clase `uvm_agent`, y está diseñado para interactuar con la interfaz especificada. Este componente constituye un bloque jerárquico de nivel, configurable y listo para usar, destinado a un protocolo que viene especificado por el DUV. Es decir, realiza el encapsulado de los elementos más básicos de la jerarquía de componentes, considerándose el primer componente reutilizable de un *testbench*. Su estructura en un *testbench* depende de la configuración. En este sentido, la clase `uvm_agent` tiene un atributo de configuración *build*, llamado `is_active`, que puede ser utilizado por el diseñador para construir un componente *agent* activo o pasivo. Estos componentes *agent* se diseñarán activos o pasivos según generen o reciban señales, respectivamente. El primero es el más apropiado para conectarlo a una interfaz de entrada del DUV, y el segundo se adapta mejor a las interfaces

de salida. Los diferentes componentes de los que consta un componente *agent* son:

- Componente *sequencer.*
- Componente *driver.*
- Componente *monitor.*

El Código 2.8 describe el ejemplo de un componente *agent* creado a partir de la clase base `uvm_agent`.

```
1   class my_agent extends uvm_agent;
2
3     uvm_active_passive_enum is_active = UVM_ACTIVE;
4
5     my_sequencer sequencer;
6     my_driver    driver;
7     my_monitor  monitor;
8
9     `uvm_component_utils_begin(my_agent)
10     `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_ALL_ON)
11    `uvm_component_utils_end
12
13    //constructor
14
15     function void build_phase(uvm_phase phase);
16       super.build_phase(phase);
17       if(is_active == UVM_ACTIVE) begin
18        sequencer= my_sequencer::type_id::create("sequencer",this);
19        driver = my_driver::type_id::create("driver", this);
20       end
21         monitor = pipe_monitor::type_id::create("monitor", this);
22        `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
23     endfunction: build_phase
24
25     function void connect_phase(uvm_phase phase);
26        if(is_active == UVM_ACTIVE)
27          driver.seq_item_port.connect(sequencer.seq_item_export);
28         `uvm_info(get_full_name(  ),  "Connect  stage  complete.",
29  UVM_LOW)
30     endfunction: connect_phase
31  endclass: my_agent
```

Código 2.8 : Código ejemplo de un componente *agent* UVM.

De la estructura del ejemplo descrito en el Código 2.8 se destaca:

- En la línea 3 se declara la variable `is_active` empleada en la configuración del componente. Como muestran las líneas 17 y 26, según el valor de la variable, se genera un agente activo o pasivo.
- Las líneas entre la 5 y 7 declaran los componentes básicos que conforman el componente *agent*. La inclusión de los componentes *sequencer* y *driver* dependerá del valor que tenga la variable `is_active`. El componente *monitor* siempre se incluye.

- Las líneas entre la 9 y la 11 registran en la *Factory* el componente `my_agent` , y su variable de configuración `is_active`.

- Las líneas entre la 15 y la 23 describen la fase `build_phase` y en ella se realiza la declaración de componentes que conforman el componente *agent*, dependiendo del valor asignado a la variable `is_active`.

- La fase `connect_phase` se implenta entre las líneas 25 y 29 y en ésta se realiza el conexionado tipo TLM , dependiendo nuevamente de la variable `is_active`. Es decir si el componente *agent* es activo se conecta el puerto `port` del componente *driver* con el puerto *`export`* del componente *sequencer*. Esto se aprecia en la línea 27 del código.

2.7.6. Componente *scoreboard*

El componente *scoreboard* es un elemento vital para proporcionar la propiedad de auto chequeo al entorno de verificación. Se encarga de verificar el DUV a un nivel funcional, es decir, su función depende de la implementación. Un componente *scoreboard* se encarga de verificar que las salidas actuales del DUV coinciden con los valores de salida previstos. Su función se basa en pronosticar los resultados esperados, para comparar éstos con las salidas DUV, verificando así su funcionamiento. Para recibir las transacciones que el componente *monitor* envío por su puerto TLM `analisis_port` estos

componentes implementan la función `write()`. El Código 2.9 describe un ejemplo de componente *scoreboard* creado a partir de la clase `uvm_scoreboard`. Cabe destacar que en el ejemplo en lugar de implementar la función `write()`, se implementan dos recursos `uvm_analisis_fifo`. Esto representa otra solución, pues las FIFOs implementan los puertos tipo `analisis_port` implícitamente y, además, proporcionan sincronismo para las transacciones almacenadas en la misma.

```
1  class my_scoreboard extends uvm_scoreboard;
2
3     uvm_tlm_analysis_fifo #(my_item) input_packets_collected;
4     uvm_tlm_analysis_fifo #(my_item) output_packets_collected;
5
6     my_item input_item;
7     my_item output_item;
8
9     `uvm_component_utils(my_scoreboard)
10
11    function new(string name, uvm_component parent);
12       super.new(name, parent);
13    endfunction: new
14
15    function void build_phase(uvm_phase phase);
16       super.build_phase(phase);
17       input_item_collected = new("input_item_collected", this);
18       output_item_collected = new("output_item_collected", this);
19       input_item = my_item::type_id::create("input_item");
20       output_item = my_item::type_id::create("output_item");
21      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
22    endfunction: build_phase
23
24    virtual task run_phase(uvm_phase phase);
25       super.run_phase(phase);
26        forever begin
27          input_packets_collected.get(input_item);
28          output_packets_collected.get(output_item);
29          compare_data( );
30       end
31    endtask: run_phase
32
33    virtual task compare_data( );
34   //Se comparan los datos y se imprimen mensajes de los resultados
35    endtask:compare_data
36
37 endclass: my_scoreboard
```

Código 2.9 : Código ejemplo de un componente *scoreboard* UVM.

Del ejemplo descrito en el Código 2.9 se destacan los siguientes puntos:

- En la línea 1 se crea la clase a partir de la clase base `uvm_scoreboard`. Cabe destacar que esta cabecera no está parametrizada.

- En la línea 3 y en la línea 4 se declaran las dos FIFOs comentadas anteriormente. La primera se emplea para recibir las transacciones que representan los estímulos de entrada al DUV y son enviadas por el agente activo (a través del componente *monitor*). La segunda se crea para recibir las transacciones que representan las salidas actuales del DUV, enviadas por el agente pasivo.

- Las líneas 5 y 6 declaran dos transacciones del tipo `my_tx` utilizadas como parámetro para indicar a las FIFOs el tipo de transacción a gestionar.

- Entre las líneas 15 y la 22 describen la fase `build_phase`. En esta fase se crean las FIFOs y las transacciones declaradas anteriormente.

- Las líneas entre la 24 y la 31 describen la tarea `run_phase`. En este ejemplo en concreto se utiliza un bucle `forever`, y mediante las FIFOs se solicitan, empleando el método `get()`, las transacciones que representan las entras y salidas del DUV para su posterior comparación. Es decir, donde se lleva a cabo la verificación funcional.

Para comparar los resultados previstos con los resultados reales del DUV, el componente *scoreboard* también puede implementar una jerarquía en su interior. Los algoritmos de predicción y los algoritmos de verificación se pueden definir en clases que son referenciadas por el componente *scoreboard*. Estos sub-bloques configurables permiten reutilizar el componente *scoreboard* y configurarlo conforme a la predicción (*predictor*) y verificación (*comparator*) necesarias. El componente *scoreboard* y las clases que lo componen,

predictor y *comparator,* se tratan con mayor detenimiento en el siguiente capítulo, pues este componente es empleado para alcanzar el objetivo principal del presente TFG, permitiendo integrar algoritmos en C para realizar las predicciones.

2.7.7. Componente *enviroment*

Un componente *enviroment* es un contenedor, de nivel superior, de componentes reutilizables. Su función se basa en referenciar y configurar los componentes que lo conforman. La mayor parte de la reutilización del proceso de verificación ocurre en este nivel, en el cual el diseñador o usuario puede referenciar un entorno y configurar sus componentes *agent* y demás componentes para tareas de verificación específicas. Esto permite cambiar y configurar el número de componentes presentes en el entorno a implementar. Los principales componentes que encapsula un entorno son:

- Uno o más componentes *agent*: encargados de conducir los estímulos de entrada al DUV y monitorizar sus entradas y salidas de acuerdo con la interfaz especificada.

- Un componente *scoreboard*: el cual se encarga de verificar las respuestas ante los estímulos del DUV.

- Opcionalmente, un componente *coverage*: que registra la información de la transacción para el análisis de la cobertura funcional.

- Un componente de configuración, que permite que el test configure el entorno, incluyendo sus componentes, de acuerdo a las necesidades de diseño.

A modo de ejemplo, el Código 2.10 describe un componente *enviroment* que se crea a partir de la clase `uvm_env`. Este componente referencia dos componentes *agent*, uno pasivo y uno activo, un componente *scoreboard* y un componente *coverage*.

```
1  class my_env extends uvm_env;
2
3     my_agent        active_agent;
4     my_agent        pasive_agent;
5     my_scoreboard    sb;
6     my_coverage     cov;
7
8     `uvm_component_utils( my_env)
9
10    //constructor
11
12    function void build_phase(uvm_phase phase);
13       super.build_phase(phase);
14    uvm_config_db#(int)::set(this,"active_agent","is_active",
15 UVM_ACTIVE);
16    uvm_config_db#(int)::set(this,"pasive_agent","is_active",
17 UVM_PASSIVE);
18
19       active_agent   =  my_agent::type_id::create("active_agent",
20 this);
21       pasive_agent   =  my_agent::type_id::create("pasive_agent",
22 this);
23       sb = my_scoreboard::type_id::create("sb", this);
24       cov = my_coverage::type_id::create("cov", this);
25
26     `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
27    endfunction: build_phase
28
29  function void connect_phase(uvm_phase phase);
30    pasive_agent.monitor.item_collected_port.
31       connect(sb.input_packets_collected.analysis_export);
32    active_agent.monitor.item_collected_port.
33       connect(sb.output_packets_collected.analysis_export);
34    active_agent.monitor.item_collected_port.
35       connect(pipe_cov.analysis_export);
36    pasive_agent.monitor.item_collected_port.
37       connect(pipe_cov.analysis_export);
38
39   `uvm_info(get_full_name( ), "Connect phase complete.", UVM_LOW)
40  endfunction: connect_phase
41
42 endclass: dut_env
```

Código 2.10: Código ejemplo de un *enviroment* UVM.

Del Código 2.10 se destacan los siguientes puntos de interés:

- La línea 1 declara la clase derivando su declaración a partir de la clase base `uvm_enviroment`.

- Las líneas entre la 3 y la 6 declaran los componentes que pueden formar parte del entorno. Cabe destacar que su inclusión dependerá

de la configuración de la clase modelada, como se comentó con anterioridad.

- La línea 8 registra el componente en la *Factory*.

- Las líneas de la 12 a la 24 implementan la fase `built_phase`. En esta fase primero se emplea el mecanismo de configuración para configuran los componentes que lo conforman, y seguidamente se referencian empleando el método `create()` proporcionado por la *Factory*.

- Finalmente, las líneas entre la 26 y 35 describen la fase `conect_phase` para conectar todos los componentes del entorno. En el ejemplo expuesto se conectan los puertos TLM `analisis_port` con el puerto `analisis_export`, pertenecientes a los componentes, al componente *monitor* y al componente *scoreboard* y el componente *coverage.*

2.7.8. El componente *test*

El componente *test* es el componente de mayor jerarquía encargado de la creación del *testbench*. Se deriva a partir de la clase base `uvm_test` y sus funciones son:

- Referenciar el componente *enviroment.*

- Referenciar e iniciar secuencias conforme al caso de *test* a ejecutar.

- Administrar los objetos `objection` de la fase `run_phase`, para asegurar que el test se complete correctamente.

El Código 2.11 muestra una clase base empleada para modelar los distintos casos de test. Esta clase contiene elementos comunes a cualquier

test y permite crear una librería con los diferentes casos de test, simplemente heredando de ella y adaptándola para el caso de test necesario.

```
1  class base_test extends uvm_test;
2
3     `uvm_component_utils(base_test)
4
5     my_env env;
6     uvm_table_printer printer;
7
8     // constructor
9
10    function void build_phase(uvm_phase phase);
11       super.build_phase(phase);
12       env = my_env::type_id::create("env", this);
13       printer = new( );
14       printer.knobs.depth = 5;
15    endfunction:build_phase
16
17    virtual function void end_of_elaboration_phase(uvm_phase phase);
18       `uvm_info(get_type_name(  ),  $sformatf("Printing  the  test
19 topology :\n%s", this.sprint(printer)), UVM_LOW)
20    endfunction: end_of_elaboration_phase
21
22    virtual task run_phase(uvm_phase phase);
23       phase.phase_done.set_drain_time(this, 1500);
24    endtask: run_phase
25
26 endclass: base_test
```

Código 2.11 : Código ejemplo de un clase *test* UVM, empleada como base de casos de *test*.

De la clase `base_test` del Código 2.11 se destaca:

- La línea 3 registra la clase en la *Factory*.
- La línea 4 declara el componente *enviroment* a emplear en el caso de test.
- La línea 5 declara un objeto del tipo `uvm_table_printer` que permite mostrar, en la ventana de comandos, la topología del test. Esto se implementa en la fase `end_of_elaboration_phase` y proporciona una herramienta útil para comprobar la correcta dependencia en la jerarquía.

- Las líneas entre la 10 y la 15 describen la fase `built_phase`. En esta fase se referencia el componente *enviroment* y el objeto `uvm_table_printer`.

- La línea 14 define el nivel de profundidad que se quiere mostrar.

- Las líneas entre la 17 y la 20 implementan la fase `end_of_elaboration_phase`. Esta fase hace uso del mecanismo de mensajes y del objeto `uvm_table_printer` para mostrar la topología del entorno para un caso de test.

- Las líneas entre la 22 y la 23 definen la tarea `run_phase` en la que se emplea el método `set_drain_time()` para añadir tiempo a la simulación y permitir a todos los elementos completar sus mecanismos de `objection`.

El Código 2.12 muestra un ejemplo de clase *test* extendida de la clase base del Código 2.11.

```
1  class test_1 extends base_test;
2
3    `uvm_component_utils(test_1)
4
5   //Constructor UVM
6
7    virtual function void build_phase(uvm_phase phase);
8      super.build(phase);
9    endfunction: build_phase
10
11   virtual task run_phase(uvm_phase phase);
12    super.run_phase(phase);
13     phase.raise_objection(this);
14       seq = my_sequence::type_id::create("seq");
15       seq.star(my_env.active_agent.sequencer);
16     phase.drop_objection(this);
17   endtask: run_phase
18
19 endclass:test_1
```

Código 2.12 : Código ejemplo de un caso de *test* UVM.

Del Código 2.12 se pueden destacar los siguientes puntos:

- En la línea 1 se muestra cómo se crea la clase a partir de la clase `base_test`. Esto, como se comentó anteriormente, permite modelar distintos casos de test a partir de la clase `base_test`.

- La línea 3 realiza el registro del test en la *Factory*.

- Las líneas entre la 7 y la 9 describen la fase `built_phase`. En esta fase se llama al método `built` de la clase `super` para iniciar la fase `built_phase` de la clase padre.

- Las líneas entre la 11 y la 14 describen la tarea `run_phase`. En esta tarea se emplean los mecanismos de `objection` comentados. Se crea una secuencia, de forma dinámica, haciendo uso de la *Factory* y se emplea el método `start()` para lanzar la secuencia y asociarla con su correspondiente al componente *sequencer*.

2.7.9. El modulo *top*

Como se mostró en la Figura 2.6, el mayor nivel de la jerarquía de componentes UVM es un módulo que se conoce con el nombre de módulo *top*. La función del módulo *top* es la de conectar el *testbench* de UVM, basado en clases, con el DUV basado en módulos. Para ello el módulo *top* se construye empleando una descripción HDL y debe contener la interfaz SV, el DUV, la lógica de control necesaria para la generación de señales tales como el reset y el reloj, y el método `run_test()`. En este módulo se referencia la interfaz SV, el DUV y se conectan. Seguidamente, antes de ejecutar el test mediante el método `run_test()`, se asocia la interfaz virtual que se emplea en el *testbench* a la interfaz del DUV. Para ello, se utiliza el método `set()` del mecanismo de configuración que permite insertar la interfaz virtual para que los componentes que necesiten configurar su interfaz virtual puedan tener acceso a la misma. Esto se comentó en el ejemplo del componente *driver* y el componente *monitor* los cuales empleaban el método `get()` para configurar su

interfaz virtual. En el Código 2.13 se describe un ejemplo de modulo *top* que muestra lo comentado.

```
1  module top;
2     import uvm_pkg::*;
3     import my_pkg::*;
4
5     bit clk;
6     bit rst_n;
7
8     my_if  ivif(.clk(clk), .rst_n(rst_n));
9     my_if  ovif(.clk(clk), .rst_n(rst_n));
10
11    my_duv  duv_top(.clk(clk),
12                    .rst_n(rst_n),
13                    .i_cf(ivif.cf),
14                    .i_en(ivif.enable),
15                    .i_data(ivif.data_in0),
16                    .o_data(ovif.data_out1)
17                   );
18
19    always #5 clk = ~clk;
20
21    initial begin
22        #5 rst_n = 1'b0;
23       #25 rst_n = 1'b1;
24    end
25
26    initial begin
27     uvm_config_db#(virtual my_if)::set(uvm_root::get( ), "*.agent.*" ,
28 "in_int", ivif);
29     uvm_config_db#(virtual my_if)::set(uvm_root::get( ), "*.monitor" ,
30 "out_int", ovif);
31
32      run_test( );
33
34    end
35 endmodule
```

Código 2.13 : Código ejemplo de un módulo *top* de UVM.

Del ejemplo de código anterior se destacan los siguientes puntos de interés:

- En la línea 1 se declara el módulo *top*.
- En la línea 2 se importa el paquete `uvm_pkg::*`,y en la línea 3 el paquete `my_pkg::*`. Estos paquetes contienen todas las declaraciones necesarias para dar soporte a la metodología UVM.

- Las líneas 5 y 6 declaran dos variables para generar la lógica de control, correspondiente a la señal de reloj y de reset. La señal de reloj se genera en la línea 19 y la de reset en un proceso tipo `initial` entre la línea 21 y la línea 24.

- Las líneas 8 y 9 declaran dos interfaces SV que sirven de punto de acceso para las interfaces virtuales declaradas en los componentes que necesitan tener acceso a la interfaz del DUV. La primera es para un componente *agent* activo y la segunda para el componente *agent* pasivo, es decir para las entradas y salidas del DUV.

- Entre las líneas 11 y 17 se referencia el DUV y se conecta éste con la interfaz.

- Las líneas 17 y 19 hacen uso del método `set()` del mecanismo de configuración para insertar la interfaz virtual en la base de datos.

- En la línea 32 se llama a la función `run_test( )`.Esta función construye el *testbench* de forma *topdown* comenzando desde el modulo *top*. Esta función se encarga de inicializar las fases de todos los componentes y construir el *testcase* conforme al caso de prueba que se le pase. Como se comentó, el nombre del *test* que se va a ejecutar se puede pasar como argumento o puede emplearse directamente mediante la línea de comandos.

Capítulo 3 *SystemVerilog Direct Programming Interface*

3.1. Introducción a los entornos multilenguaje

La metodología UVM proporciona los elementos y mecanismos necesarios para la automatización y reutilización de entornos de verificación que permiten generar entornos potentes y flexibles, con la finalidad de construir *testbenches* descritos en *SystemVerilog*. Sin embargo, por lo general los proyectos en los que se requiere verificar diversos bloques IP que conforman un sistema complejo, como en el caso de los SoC (*System On Chip*), normalmente se construyen con componentes implementados en diferentes lenguajes y diferentes niveles de abstracción. Además, los diseñadores pueden disponer de modelos o algoritmos de verificación, escritos en otros lenguajes. En consecuencia, la reutilización de códigos o VIPs implementados en diferentes lenguajes, es razón suficiente para justificar la necesidad de contar con entornos de verificación multilenguaje. Los *testbenches* actuales puedes estar compuestos por componentes escritos en diferentes lenguajes (*SystemVerilog*, *e*, *SystemC*, *C/C++,etc*) y los integradores y diseñadores deben combinar estos componentes para proporcionar una solución para las necesidades de cada entorno o caso de verificación. Si bien la metodología UVM proporciona soporte multilenguaje y existen herramientas de simulación que dan soporte también a los entornos multilenguaje, todavía hay mucho trabajo pendiente para tratar de alcanzar la solución óptima. Esta solución deberá tratar cada lenguaje como una contribución para alcanzar la solución deseada, tratando siempre de mantener la interoperabilidad de los VIPs [16].

3.2. Reutilización de código descrito en C en un entrono UVM

La decisión de emplear el lenguaje C para modelar un componente de un sistema depende en gran medida de la exactitud detallada de la temporización requerida en el modelo. Los modelos se describen en el lenguaje de programación C para realizar pruebas de alto nivel y gran parte del modelado se realiza en C, a menos que existan problemas de sincronización. Estos modelos en C tienen la ventaja de ser portátiles y de ser utilizados comúnmente para la descripción funcional, resultando fáciles de depurar y de ejecutar. Además, a medida que avanza el diseño, y por tanto la verificación, será importante que los modelos con abstracciones de tiempo variable y los modelos sin especificaciones de temporización de alto nivel funcionen conjuntamente.

En determinados ámbitos, como puede ser el procesado de señales de imágenes (ISP) los algoritmos se desarrollan y evalúan empleando modelos implementados en C antes de la implementación RTL. Una vez finalizada la descripción de los algoritmos, los modelos en C se pueden reutilizar como modelos de referencia para el desarrollo de VIPs y los correspondientes entornos de verificación [7]. Esto aporta dos ventajas para la verificación funcional en un entorno basado en la metodología UVM:

- En primer lugar, se pueden aprovechar estos modelos para realizar la verificación funcional del DUV. Es decir, se pueden realizar las predicciones de las salidas esperadas empleando estos modelos de referencia, y comparar éstas con las salidas actuales del DUV, verificando de esta forma su funcionamiento.

- En segundo lugar, el disponer de un modelo de referencias antes de la implementación RTL del DUV, permite adelantar el diseño del *tesbench* utilizando dicho modelo de referencia. Esto permite ir planificando la generación de los vectores de prueba requeridos para las pruebas del entorno de verificación, sin esperar a que el diseño RTL del DUV esté disponible.

En este contexto, se plantea con frecuencia la cuestión de cómo llamar a un modelo de referencia descrito en C/C++ desde un *tesbench* basado en la metodología UVM y descrito en *SystemVerilog*. En los siguientes apartados del presente capítulo se introduce el uso básico de *SystemVerilog* DPI. Esta interfaz, que pertenece al estándar *Accellera SystemVerilog,* permite integrar código escrito en C en un entorno de verificación basado en la metodología UVM mediante un conjunto de directrices. Cabe destacar que, aunque cada uno de los lenguajes antes mencionado está definido por un estándar formal, la interfaz entre los lenguajes no está estandarizada con la misma precisión que los lenguajes. Esto conlleva que en la práctica las diferencias entre las implementaciones y la ausencia de soporte de estándar para las llamadas entre ambos lenguajes suela requerir una solución dependiente de la herramienta. Sin embargo, la naturaleza simple y directa de *SystemVerilog* DPI lo hace ideal para llamar a funciones de bibliotecas C estándar, como la biblioteca matemática del estándar de programación C, o de bibliotecas definidas por el usuario, proporcionando un mecanismo directo para representar partes de un diseño como modelos abstractos escritos en C, y modelos de comportamiento *SystemVerilog* más detallados, en un nivel RTL.

3.3. La interfaz *SystemVerilog DPI*

La Interfaz de Programación Directa de *SystemVerilog* (SV DPI) tiene su origen en dos interfaces propietarias, la interfaz *VCS DirectC* de *Synopsys* y la interfaz *SystemSim Cblend* de *Co-Design Automation* (actualmente Synopsys). Estas dos interfaces, inicialmente propietarias, fueron soluciones desarrolladas para trabajar con sus respectivos simuladores, tratando de ampliar sus capacidades. Fue el comité de estándares *Accellera SystemVerilog* el que fusionó las capacidades de las dos tecnologías y definió la semántica de la interfaz DPI con el objetivo de ser compatible con cualquier simulador *SystemVerilog*. La motivación para desarrollar esta interfaz es doble. En primer lugar, la interfaz debe permitir la construcción de un sistema heterogéneo (un diseño, un *testbench*) en el que algunos componentes descritos en otros lenguajes puedan interactuar con componentes descritos en

SystemVerilog. Por otro lado, existe también la necesidad práctica de disponer de una forma directa y eficiente de conectar el código existente, normalmente descrito en C/C++, sin el conocimiento y la sobrecarga de las interfaces *Verilog PLI (Programming Language Interface) y VPI (Verilog Procedural Interface)* [17].

La interfaz DPI sigue el principio de caja negra, lo cual implica que la especificación y la implementación de un componente está claramente separada, y la implementación real es transparente para el resto del sistema. Por lo tanto, el lenguaje de descripción empleado para la implementación también es transparente, aunque cabe destacar que el estándar define sólo la semántica de enlace con el lenguaje C/C++. La separación entre el código *SystemVerilog* y el código C se basa en el uso de funciones/tareas como la unidad de encapsulación. En general, cualquier función se puede tratar como una caja negra y se implementa en *SystemVerilog* o en C de una manera transparente, es decir como si el código fuera nativo en el lado de cada lenguaje. Por lo tanto, la principal característica de la interfaz DPI es que permite que el código descrito en el lenguaje *SystemVerilog* pueda llamar directamente a funciones/tareas descritas en el lenguaje de programación C y viceversa. De este modo, la DPI permite pasar valores directamente entre las funciones/tareas de *SystemVerilog* y C, permitiendo de esta forma que ambos códigos se comuniquen e interactúen con un propósito común. Para la implementación de la interfaz SV DPI, se requiere el uso de cabeceras de importación y exportación de funciones/tareas, así como una concordancia entre los lenguajes para el intercambio en el formato de los tipos de datos.

3.3.1. Los dominios de la interfaz *SystemVerilog* DIP

La DPI como interfaz contiene dos dominios bien determinados, el dominio del lenguaje *SystemVerilog* y el dominio del lenguaje de programación. Como se mencionó anteriormente, en teoría es posible interconectar *SystemVerilog* con cualquier lenguaje de programación, con una semántica similar a la del

lenguaje C, empleando DPI. Sin embargo, por el momento, la semántica para la interfaz de lenguaje sólo ha sido definida para C/C++.

Ambos dominios de la interfaz permanecen independientes. Es decir, la interpretación de datos o cualquier interactuación entre los dos dominios de la interfaz es independiente y depende únicamente de la implementación. Los usuarios de interfaz DPI son responsables de especificar en su código C los tipos nativos equivalentes a los tipos *SystemVerilog* utilizados en las declaraciones de importación o exportación. Mediante las importaciones y exportaciones, la interfaz DPI permite la transferencia de datos entre dos dominios a través de argumentos de función y retorno [18].

3.3.2. Funciones/tareas importadas y exportadas

En el lado *SystemVerilog* una llamada DPI es una importación o una exportación. Estas definiciones son desde el punto de vista del código *SystemVerilog*. Es decir, una función/tarea declarada como una importación hace referencia al código C que se importa, para que el código *SystemVerilog* lo llame. Mientras que una función/tarea declarada como exportación en el lado de *SystemVerilog* es el código *SystemVerilog* exportado para ser llamado desde el código escrito en C.

3.3.2.1. Funciones/tareas importadas

Los métodos implementados en C, y declarados como importados en *SystemVerilog,* se pueden llamar desde el código *SystemVerilog* como si fueran funciones/tareas nativas del propio lenguaje. Estas tareas/funciones se denominan métodos importados. La característica simple y directa de *SystemVerilog* DPI permite llamar a funciones de bibliotecas C estándar, como la biblioteca matemática del estándar de programación C, o bibliotecas y funciones/tareas definidas por el usuario. La declaración de importación se

define por un prototipo del nombre de una función/tarea implementada en C precedido del comando `import "DPI"`, con sus argumentos y tipo de retorno de función [19]. Dos ejemplos de declaraciones de importación son:

- La función seno de la librería *math* de C:

```
import "DPI" function real sin(real in);
```

- Una tarea definida por el usuario en el lado de C:

```
import "DPI" task file_write(string data);
```

Cabe destacar que aunque las tareas y las funciones importadas se definen de forma similar, presentan algunas diferencias propias del comportamiento de una función o una tarea:

- Las tareas/funciones pueden tener argumentos de entrada (`input`), salida (`output`) o entrada/salida (`inout`).
    - Los argumentos declarados como `input`, valor por defecto, no se modifican. Es decir, si estos argumentos se cambian dentro de una función/tarea, los cambios no serán visibles fuera de la misma.
    - Los valores iniciales de los argumentos declarados como `ouput` son indeterminados y dependen de la implementación. Es decir, la función/tarea no asumirá nada sobre los valores iniciales.
    - La función/tarea puede acceder al valor inicial de un argumento `inout` y los cambios que la función/tarea realiza al argumento declarado como `inout`, serán visibles fuera de la función/tarea.
- Las funciones no consumen tiempo de simulación.
- Los tipos de resultados de las funciones están restringidos a los siguientes tipos de datos:

- `void, byte, shortint, int, logint, real, shortreal, chandle y string.`
- `packed bit arrays` de 32 bits como máximo.
- Valores escalares de tipo `bit` y `logic`.

- Las tareas pueden consumir tiempo de simulación.
- Las tareas importadas solo pueden devolver un resultado, indicando que ha finalizado su ejecución.

3.3.2.2. Funciones/tareas exportadas

Además de importar funciones/tareas desde C, la interfaz DPI permite que las tareas/funciones de *SystemVerilog* se exporten al código escrito en lenguaje C. Esto permite que el código dentro del lenguaje C ejecute código descrito en *SystemVerilog*. Una declaración de exportación es similar a una declaración de importación DPI, con la diferencia de que sólo se especifica el nombre de la tarea/función a exportar del código *SystemVerilog,* los argumentos de la tarea/función *SystemVerilog* no se especifican como parte de la declaración de exportación DPI [19]. Un ejemplo de declaración de exportación es el siguiente:

```
export "DPI" adder_function;
```

Una ventaja importante de exportar las tareas de *SystemVerilog* es que las tareas pueden consumir tiempo de simulación mediante el uso de asignaciones no bloqueantes, controles de eventos, demoras y declaraciones de espera. Esto proporciona una forma de que una función C basada en DPI sincronice la actividad con el tiempo de simulación. Cuando una función C llama a una tarea exportada de *SystemVerilog* que consume tiempo, la ejecución de la función C se detendrá hasta que la tarea de *SystemVerilog* complete la ejecución y regrese a la función C de llamada. Mediante una combinación de funciones/tareas importadas del código C y funciones/tareas

exportadas del código *SystemVerilog*, los datos se pueden pasar con facilidad entre los dos lenguajes aprovechando las ventajas que aporta cada uno y ampliando las capacidades del simulador. Sin embargo, cabe destacar, que no se pueden realizar las exportaciones en las clases *SystemVerilog,* pues no se permite exportar funciones/tareas miembros de clases al ser éstas dinámicas.

3.3.3. Funciones/tareas *pure* y *context*

Una función declarada como `pure` depende únicamente de los valores que se pasan a la función a través de sus argumentos. Una ventaja de estas funciones es que los simuladores pueden ser capaces de realizar optimizaciones que mejoren el rendimiento de la simulación. Sin embargo, se debe tener en cuenta que una función `pure` no puede utilizar variables globales o estáticas, no puede realizar operaciones de E/S de archivos, no puede tener acceso a variables de entorno del sistema operativo, y no puede llamar a funciones de las bibliotecas *Verilog* PLI. Sólo las funciones que no devuelven un tipo `void`, y con argumentos de entrada tipo `input` pueden ser especificadas como `pure`. Las tareas importadas no permiten ser declaradas como `pure`. A continuación se muestra un ejemplo de función C importada, declarada como `pure`. Este ejemplo corresponde a la función seno de la librería `math` del lenguaje de programación C. Esta función solo necesita el argumento para devolver el resultado y no depende de otras funciones, por lo que declarándola como `pure` se optimiza la simulación.

```
import "DPI" pure function real sin(real in);
```

Las funciones/tareas declaradas como `context` permiten que se conozca el contexto de su llamada y si se pueden comunicar con otras funciones/tareas. Esto hace posible que las funciones DPI aprovechen las funciones del lenguaje PLI como escribir en el archivo de registro del simulador y sobre los archivos abiertos desde el código *SystemVerilog*. Una declaración de ejemplo de una tarea dependiente del contexto es la siguiente:

```
import "DPI" context task print(input int file_id, input
bit [127:0] data);
```

Finalmente, cabe destacar que es responsabilidad del usuario declarar correctamente una función/tarea importada como `pure` o `context`. De forma predeterminada, se supone que las funciones DPI son genéricas. Es decir, si una función C accede a las bibliotecas PLI u otras bibliotecas API, y esa función no se declara como una función `context`, pueden producirse resultados de simulación impredecibles, e incluso bloquearse. De la misma forma, si una llamada no se declara como `pure` no obtiene optimización en la simulación.

3.3.4. Restricciones en los tipos de datos

La interfaz *SystemVerilog* DPI admite sólo tipos de datos *SystemVerilog*, que son los únicos tipos de datos que pueden cruzar el límite entre el lenguaje *SystemVerilog* y lenguaje de programación C en ambas direcciones. Un valor que se pasa a través de la DPI se especifica en el código *SystemVerilog* como un valor de tipo *SystemVerilog*, mientras que el mismo valor se especificará en código C como un valor de tipo C compatible. Por lo tanto, se requiere un par de definiciones de tipo coincidentes para pasar un valor a través de DPI, la definición *SystemVerilog* y la definición C [18]. En la Tabla 3.1 se muestran algunos tipos de *SystemVerilog* que son directamente compatibles con los tipos definidos en C.

Tipo de dato *SystemVerilog*	Tipo de dato C
bit	Unsigned char
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void*
string	char*

Tabla 3.1: Tipos de datos directamente compatibles entre SystemVerilog y C.

Sin embargo, hay tipos específicos del lenguaje *SystemVerilog*, como los tipo `packet` (`arrays`, `estructuras` y `uniones`), de 2 estados o de 4 estados, que no tienen representación directa en el lenguaje C. Aunque no se especifica una representación real de los tipos `packet`, el lado de la interfaz DPI del código C define una representación canónica para `arrays` del tipo `packet` de 2 y 4 estados. Para ello se debe incluir el archivo `svdpi.h`. Este archivo es independiente de la implementación y define todos los tipos básicos y todas las funciones de la interfaz, empleando para ello una biblioteca que proporciona la traducción entre la representación utilizada en un simulador y la representación canónica del `array` tipo `packet`. También se pueden pasar `arrays` de tipos directamente compatibles como argumentos en las llamadas a la DPI, como los mostrados en la Tabla 3.1. Cabe destacar que también se pueden especificar como `arrays` abiertos, pero únicamente en las declaraciones de importación, pues las exportaciones no soportan este tipo de argumentos. Un argumento de tipo `array` abierto se declara con uno o más rangos de sus dimensiones sin especificar, empleando corchetes vacíos (`[]`). Para los `arrays` tipo `packet` es diferente, en el lado del lenguaje C se utiliza rangos normalizados. Estos rangos normalizados se emplean para la representación canónica de los `arrays` tipo `packet` entre C y *SystemVerilog*. De tal forma que la normalización de `[n-1:0]` se aplica a la dimensión de la

parte `packet` del `array` (restringidos a una dimensión) y la normalización `[0:n-1]` se aplica a la dimensión en la parte `unpacket` del `array` [17].

Todos los tipos anteriores se pasan como argumentos de la función y se pueden pasan por referencia o por valor y, por lo tanto, son directamente accesibles en el código C. Los valores de los argumentos declarados como `input` se pasan siempre por valor, mientras que los argumentos declarados en *SystemVerilog* como `array` abierto se pasan por un identificador (tipo `svOpenArrayHandle`) y son accesibles a través de funciones de biblioteca.

Finalmente, se destaca que la DPI asigna la carga al usuario para que coincidan los tipos de datos definidos en el lenguaje *SystemVerilog* con los tipos de datos equivalentes en el lado del lenguaje C. Es decir, la interfaz DPI no proporciona un mecanismo para que las función/tareas comprueben el tipo de dato que se está empleando en el otro lado del lenguaje. Una explicación más detallada de la interfaz *SystemVerilog* DPI se puede encontrar en el documento *SystemVerilog Language Reference Manual* [17].

3.3.5. Ejemplos de uso de la DPI

En este apartado se muestran dos ejemplos simples, uno donde se importa una función desde C y otro donde se exporta una función *SystemVerilog*, que muestran el funcionamiento básico de la DPI.

El Código 3.1 y el Código 3.2 muestran un ejemplo del proceso importación DPI. Este ejemplo representa una de las primeras pruebas que se realizan para comprender y ver el funcionamiento de la interfaz DPI.

```
1  // En el código SystemVerilog
2  ....
3  import "DPI-C" function void my_func(string s);
4
5  initial
6    my_func("Hello world\n");
7  ....
```

Código 3.1: Ejemplo de importación DPI en el código *SystemVerilog.*

```
1  // En el código en C
2  ....
3   int my_func(const char* s);
4  .....
```

Código 3.2: Ejemplo de importación DPI en el código C.

Los Códigos 3.1 y 3.2 muestran los pasos a seguir para llevar a cabo una importación DPI. En primer lugar, se dispone de una función implementada en lenguaje C, línea 3 del Código 3.2. En segundo lugar, en la descripción del código *SystemVerilog* se declara la función a importar, línea 3 del Código 3.1, y seguidamente se puede llamar, línea 6 del Código 3.2, como si fuera una función nativa del lenguaje *SystemVerilog*.

En el Código 3.3 y el Código 3.4 se muestra un ejemplo de exportación. En este ejemplo se llama una tarea implementada en *SystemVerilog* desde el código C. Este ejemplo es interesante porque permite que modelos descritos en lenguaje C, los cuales son modelos sin tiempo, puedan ejecutar controles de temporización (retardos o esperas) para la sincronización.

```
1  // En el código SystemVerilog
2  .....
3  export "DPI-C" task my_task;
4  ....
5  task my_task;
6  #10;
7  endtask
8  ....
```

Código 3.3: Ejemplo de una exportación DPI en el código *SystemVerilog.*

```
1  // En el código en C
2  ....
3  { my_task(); }
4  ....
```

Código 3.4: Ejemplo de una exportación DPI en el código C.

El Código 3.3 y el Código 3.4 muestran los pasos a seguir para llamar a una tarea *SystemVerilog* desde el código C. En primer lugar se dispone de la función/tarea en *SystemVerilog*, la cual se puede observar entre las líneas 5 y 7 del Código 3.3. En segundo lugar se declara la función/tarea a exportar del código *SystemVerilog*, línea 3 del Código 3.4, y seguidamente se puede llamar desde el código C, como si fuera una función/tarea nativa del lenguaje C.

Capítulo 4 Integración de un modelo de referencia descrito en C en un entorno UVM

4.1. Introducción

En este capítulo se detallará cómo realizar la integración de un modelo de referencia descrito en C en un entorno de verificación basado en la metodología UVM. Para el desarrollo del entorno de verificación se hará uso de los conceptos explicados en el Capítulo 2. La integración del modelo de referencia en el entorno UVM se llevará a cabo en el componente *scoreboard*. Este componente estará formado por otros dos, el *predictor* y el *comparator*. Para la comunicación entre el código descrito en *SystemVerilog* y el código C se utilizará la interfaz *SystemVerilog* DPI descrita en el Capítulo 3.

4.2. Arquitectura del componente *scoreboard*

El componente *scoreboard* se implementa en un *testbench* con el objetivo de determinar si el DUV funciona de acuerdo a sus especificaciones. El componente *scoreboard* pronostica la respuesta esperada y luego la compara con la respuesta actual u observada del DUV [20]. Por lo tanto, el propósito fundamental de un componente *scoreboard* en un entorno de verificación UVM consiste en:

- Examinar las entradas del DUV obtenidas a través del componente *agent* activo, para predecir el resultado esperado.

- Examinar las salidas del DUV obtenidas del componente *agent* pasivo, para verificar las mismas.

- Comparar las salidas predichas o esperadas con las salidas actuales del DUV, con el objetivo de informar sobre la coincidencia o desigualdad de las comparaciones.

Es decir, un componente *scoreboard* debe incluir alguna forma de predecir los resultados esperados, comparar los resultados esperados con los resultados actuales u observados y hacer un informe de seguimiento de los resultados en el proceso de comparación. Básicamente, las funciones que debe incluir un componente *scoreboard* en un *testbench* son:

- Obtener una copia de la transacción de entrada al DUV.
- Extraer la información de entrada a emplear para la predicción.
- Utilizan la información de entrada extraída para predecir las salidas esperadas.
- Obtener una copia de la transacción de salida actual o real del DUV.
- Comparar las salidas de las transacciones esperadas con las salidas de las transacción actuales o reales.
- Generar un informe de seguimiento de las operaciones de comparación.
- Generar mensajes de error de comparación a medida que se detectan.
- Generar un informe final con éxito o fracaso al finalinalizar la simulación.

Por este motivo, en entornos de verificación más complejos, como ya se comentó, la arquitectura y el comportamiento del componente *scoreboard* se implementa a partir de dos clases que formarán la estructura del componente *scoreboard,* el componente *predictor* y el componente *comparator*. Es decir, la funcionalidad del componente *scoreboard* se divide en 2 tareas. Una para llevar a cabo las predicciones y la segunda para llevar a cabo las comparaciones. En la Figura 4.1 se muestra la arquitectura de un componente *scoreboard* conformado por un componente *predictor* y un componente *comparator*.

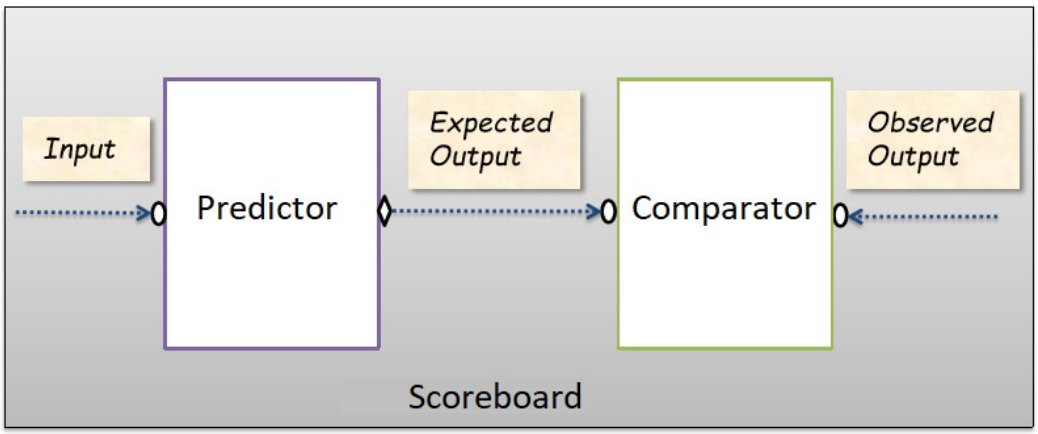

Figura 4.1: Arquitectura del componente *scoreboard*, compuesta de un componente *predictor* y un componente *comparator.*

Como se muestra en la Figura 4.1, la funcionalidad de predecir los resultados se implementada en el componente *predictor*, el cual realiza las siguientes funciones:

- Obtiene una copia de la transacción de entrada.
- Extrae la información de entrada a emplear para la tarea de predicción.
- Utiliza la información de entrada extraída para predecir las salidas esperadas.

- Convierte las entradas predichas o esperadas en transacciones, para poder enviarlas al componente *comparator*.

La funcionalidad implementada en el componente *comparator* realiza la comparación entre las salidas predichas o esperadas y las reales y además debe informar del progreso de las comparaciones. Para ello:

- Obtiene una copia de la transacción predicha y que ha sido enviada por el componente *predictor*.

- Obtiene una copia de la transacción de salida actual o real.

- Compara la transacción de salidas esperadas con las transacciones de salidas reales.

- Realiza el cómputo de las tasas de acierto y error de acuerdo con las operaciones de comparación.

- Genera informes de los fallos en las operaciones de comparacion a medida que se detectan.

- Genera un informe final con resultados de éxito y/o fracaso al finalizar la simulación.

4.3. El Cifrado *Hill Cipher*

La criptografía es un mecanismo que permite ocultar información de un mensaje a cualquier persona que no le esté permitido obtenerla. La palabra criptografía proviene del griego *krypto* (oculto), y *graphos* (escribir), que significa escritura oculta. Se divide en dos grandes ramas, la criptografía de clave privada o simétrica y la criptografía de clave pública o asimétrica [21].

La criptografía simétrica se ha implementado en diferentes tipos de dispositivos: manuales, mecánicos, eléctricos, hasta llegar a los ordenadores, donde se programan los algoritmos actuales. La Figura 4.2 representa la idea general de la criptografía simétrica, consistente en aplicar diferentes funciones al mensaje que se desea cifrar, de modo que sólo conociendo la clave pueda descifrarse.

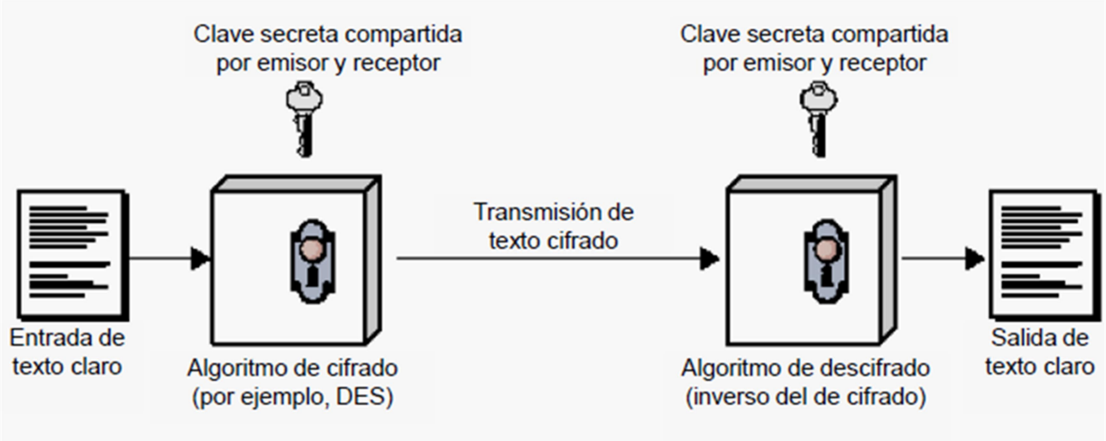

Figura 4.2: Cifrado simétrico.

El cifrado simétrico es una forma de criptosistema en la cual la encriptación y desencriptación se realiza usando la misma clave. El cifrado simétrico transforma un texto plano en un texto cifrado utilizando una clave secreta y un algoritmo de cifrado. A su vez, se utiliza la misma clave y un algoritmo de descifrado lo cual permite recuperar el texto cifrado. Las técnicas tradicionales de cifrado simétrico utilizan la sustitución o transposición, estas técnicas convierten elementos de texto sin formato (caracteres, bits) en elementos de texto cifrado.

El cifrado de *Hill Cipher* fue inventado por el matemático norteamericano Lester S. Hill en 1929, en su artículo *Cryptography in an Algebraic Alphabet*, publicado en *The American Mathematical Monthly*. Está basado en el álgebra lineal y fue el primer sistema criptográfico polialfabético que era práctico para trabajar con más de tres símbolos simultáneamente [22]. Este sistema es polialfabético pues puede darse que un mismo carácter en un mensaje a enviar se encripta en dos caracteres distintos en el mensaje encriptado. Este

algoritmo consiste en asociar cada letra del alfabeto con un número. La forma más sencilla de hacerlo es la asociación natural ordenada, aunque podrían realizarse otras asociaciones diferentes. Además, se pueden elegir otras letras del alfabeto, signos, números, etc. El alfabeto a emplear estará formado por *n* letras previamente definidas por el usuario, por ejemplo, para la aritmética modular 26 se proponen los siguientes caracteres: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*, *K*, *L*, *M*, *N*, *O*, *P*, *Q*, *R*, *S*, *T*, *U*, *V*, *W*, *X*, *Y*, *Z*, con el cifrado que se presenta en la Tabla 4.1.

Letra	Cifrado	Letra	Cifrado	Letra	Cifrado	Letra	Cifrado
A	0	B	1	C	2	D	3
E	4	F	5	G	6	H	7
I	8	J	9	K	10	L	11
M	12	N	13	0	14	P	15
Q	16	R	17	S	18	T	19
U	20	V	21	W	22	X	23
Y	24	Z	25				

Tabla 4.1: Alfabeto a emplear formado por 26 letras para la aritmética modular 26.

Como en la correspondencia anterior, entre letras/signos y números, aparecen 26 números, hay que trabajar con los números enteros módulo 26. Es decir, se consideran los números enteros 0, 1, 2,… , 25 y el resto se identifica con estos de forma cíclica. Así, el 26 es igual a 0, el 27 a 1, el 28 a 2, etcétera, y lo mismo ocurre con los números negativos, de forma que – 1 es igual 25, – 2 es igual 24, etcétera. Además, se reducen las operaciones aritméticas (suma, resta, multiplicación y división) al conjunto de los números enteros módulo 26 de forma natural, es decir, al operar dos números enteros (módulo 26) el resultado se considera también espresado en módulo 26. Cabe destacar que este algoritmo puede emplear otra aritmética modular (Módulo 27, 28,*etc.*) agregando los caracteres faltantes para completar la tabla de caracteres a utilizar.

Para encriptar un mensaje, cada bloque de *n* letras correspondiente al mensaje a transmitir (considerados como un *n-vector-componente*) está

multiplicado por una matriz invertible *n* × *n*, aplicando módulo *n*. Para la encriptación se emplea la fórmula matemática, mostrada en la Ecuación 4.1:

$$C = PK \bmod n$$

Ecuación 4.1.

Donde:

- *C* : es el texto encriptado.
- *P* : es el vector de texto a encriptar.
- *K* : es la matriz llave empleada para la encriptación.
- *Mod n*: es la operación modular, y *n* es el número de símbolos totales, los cuales conforman un alfabeto determinado.

Para desencriptar el mensaje, cada bloque es multiplicado por el inverso de la matriz llave empleada para la encriptación. Para la desencriptación se utiliza la fórmula mostrada en la Ecuación 4.2:

$$P = K^{-1}\ C \bmod n$$

Ecuación 4.2.

Donde:

- *P* : es el mensaje desencriptado.
- K^{-1} : es la llave inversa (transformación de la llave original) para realizar la operación de desencriptación.
- *C* : Es el mensaje cifrado.
- *Mod n* : es la operación modular, y *n* es el número de símbolos totales, los cuales conforman un alfabeto determinado.

Una restricción para que el método de encriptación *Hill Cipher* se pueda aplicar, es que la matriz clave privada tenga matriz inversa asociada. Esto, en aritmética modular, significa que:

- El determinante de la matriz clave privada tiene que ser distinto de cero.

- El máximo común divisor entre el determinante de la matriz, clave privada y la aritmética modular con la que se trabaja (Módulo 26,27,*etc*.) tiene que ser igual a 1.

A continuación se muestra un ejemplo donde se lleva a cabo una encriptación y desencriptación empleando el algoritmo de *Hill Cipher*. En primer lugar se obtiene una matriz llave de cifrado, empleada para la encriptación y tiene que ser escogida entre el conjunto de matrices invertibles *n* × *n* (Módulo 26 para este ejemplo). Para el ejemplo se ha considerado como mensaje la secuencia *ACT*, y como clave la clave mostrada en la matriz de la Ecuación 4.3:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}$$

Ecuación 4.3.

La matriz de la Ecuación 4.3 se corresponde con la cadena de caracteres *GYBNQKURP*. Empleando la tabla 4.1 para la aritmética modular mostrada en el ejemplo, donde *A* es 0, *C* es 2 y *T* es 19, el mensaje se corresponde con el vector mostrado en la Ecuación 4.4:

$$\begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix}$$

Ecuación 4.4.

Como se comentó, no todas las matrices tienen su matriz inversa. La matriz tendrá inversa si su determinante no es cero y no tiene ningún factor común con la base modular. Así, en el ejemplo, se ha elegido usar módulo 26, por lo cual el determinante tiene que ser distinto de 0, y no tiene que ser divisible por 2 o 13. Si consideramos la matriz de la Ecuación 4.5, tenemos que:

$$\begin{vmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{vmatrix} = 25$$

Ecuación 4.5.

El determinante de la matriz anterior es 25. Este determinante es distinto de 0 y no tiene factores comunes con 26, lo que implica que esta matriz se puede emplear para el cifrado de *Hill Cipher*. Por ello el vector cifrado se corresponde con lo mostrado en la Ecuación 4.6.

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} = \begin{pmatrix} 67 \\ 222 \\ 319 \end{pmatrix} = \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} \ mod\ 26$$

Ecuación 4.6.

Obteniéndose como texto cifrado el texto correspondiente al significado *POH*, el cual es mostrado en la Ecuación 4.7.

$$\begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix}$$

Ecuación 4.7.

Para desencriptar el mensaje anterior, se transforma el texto cifrado en un vector, que tendrá que multiplicarse por la matriz inversa de la matriz clave, que en este caso en particular se corresponde a la secuencia de letras *IFKVIVVMI*. Para ello se calcula la matriz inversa, en módulo 26, de la matriz empleada para el encriptado, como se puede observar en la Ecuación 4.8.

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}^{-1} = \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} \ mod\ 26$$

Ecuación 4.8.

Tomando el ejemplo anterior de texto cifrado *POH* y aplicando la fórmula de desencriptación se obtiene el mensaje original, que se muestra en la Ecuación 4.9.

$$\begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} = \begin{pmatrix} 260 \\ 574 \\ 539 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} \ mod\ 26$$

Ecuación 4.9.

El cual da como resultado la secuencia de caracteres *ACT*, correspondiente al mensaje original.

4.3.1 Implementación en C del algoritmo *Hill Cipher*

El Código 4.1 muestra un programa escrito en el lenguaje C correspondiente a la implementación del algoritmo *Hill Cipher* que se empleará como modelo de referencia para la verificación funcional. Este programa emplea los algoritmos del cifrado *Hill Cipher* comentados en el apartado anterior. En el programa se emplean cuatro funciones, cuyos prototipos están descritos entre las líneas 7 y la 10 del Código 4.1, para realizar la función de

encriptado y desencriptado de mensajes de tres caracteres empleando para ello una matriz llave de 3x3.

```
#include<stdio.h>
#include<math.h>

float encrypt[3][1], decrypt[3][1], a[3][3], b[3][3], mes[3][1],
c[3][3];

void encryption();    //encrypts the message
void decryption();    //decrypts the message
void getKeyMessage(); //gets key and message from user
void inverse();      //finds inverse of key matrix

void main() {
    getKeyMessage();
    encryption();
    decryption();
}

void encryption() {
    int i, j, k;

    for(i = 0; i < 3; i++)
        for(j = 0; j < 1; j++)
            for(k = 0; k < 3; k++)
             encrypt[i][j] = encrypt[i][j] + a[i][k] * mes[k][j];

    printf("\nEncrypted string is: ");
    for(i = 0; i < 3; i++)
        printf("%c", (char)(fmod(encrypt[i][0], 26) + 97));
}

void decryption() {
    int i, j, k;

    inverse();

   for(i = 0; i < 3; i++)
     for(j = 0; j < 1; j++)
         for(k = 0; k < 3; k++)
         decrypt[i][j] = decrypt[i][j] + b[i][k] * encrypt[k][j];

    printf("\nDecrypted string is: ");
    for(i = 0; i < 3; i++)
        printf("%c", (char)(fmod(decrypt[i][0], 26) + 97));

    printf("\n");
}

void getKeyMessage() {
    int i, j;
    char msg[3];

    printf("Enter 3x3 matrix for key (It should be inversible):\n");

    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++) {
            scanf("%f", &a[i][j]);
```

```
57              c[i][j] = a[i][j];
58          }
59
60      printf("\nEnter a 3 letter string: ");
61      scanf("%s", msg);
62
63      for(i = 0; i < 3; i++)
64          mes[i][0] = msg[i] - 97;
65  }
66
67  void inverse() {
68      int i, j, k;
69      float p, q;
70
71      for(i = 0; i < 3; i++)
72          for(j = 0; j < 3; j++) {
73              if(i == j)
73                  b[i][j]=1;
75              else
76                  b[i][j]=0;
77          }
78
79      for(k = 0; k < 3; k++) {
80          for(i = 0; i < 3; i++) {
81              p = c[i][k];
82              q = c[k][k];
83
84              for(j = 0; j < 3; j++) {
85                  if(i != k) {
86                      c[i][j] = c[i][j]*q - p*c[k][j];
87                      b[i][j] = b[i][j]*q - p*b[k][j];
88                  }
89              }
90          }
91      }
92
93      for(i = 0; i < 3; i++)
94          for(j = 0; j < 3; j++)
95              b[i][j] = b[i][j] / c[i][i];
96
97      printf("\n\nInverse Matrix is:\n");
98      for(i = 0; i < 3; i++) {
99          for(j = 0; j < 3; j++)
100             printf("%d ", b[i][j]);
101
102         printf("\n");
103     }
104 }
105
```

Código 4.1: Programa escrito en C del cifrado *Hill Cipher.*

El Código 4.1 corresponde a la implementación en C utilizada en este ejemplo, y tiene las siguientes caracteristicas:

- La línea 2 incluye el archivo `<math.h>`, empleado para utilizar la función `mod 26` de dicha librería.

- En la línea 4 se declaran las variables globales empleadas por las funciones del programa. Es importante destacar que para estas variables se deberá de buscar sus equivalentes con *SystemVerilog* para poder utilizar la interfaz DPI.

- En las líneas entre la 7 y la 10 se declaran los prototipos de las funciones que se emplean en el programa.

- Las líneas entre la 12 y la 16 implementan la función principal, función `main()`, del programa. Como se muestra en el Código 4.1 esta funcion se encarga de llamar a las funciones `getKeyMessage()`,`encryption()`,`decryption()`.

- Las líneas de la 48 a la 66 describen la función `getKeyMessage()`. Esta función emplea la función `scanf()` para recibir por consola la matriz llave para el cifrado y el mensaje a cifrar. La cadena de mensaje dada y la cadena de clave se representan en forma de *array*.

- Las líneas de la 18 a la 29 describen la función `encryption()`. La clave y la matriz de mensajes se multiplican y se toma el módulo 26 para cada elemento de la matriz obtenido en la multiplicación. La matriz clave, como se comentó, debe ser invertible y no debe tener múltiplos con el modulo empleado (mod 26 en este caso), de lo contrario no será posible descifrar el mensaje.

- Las líneas de la 31 a la 46 describen la función `decryption()`. La matriz de mensajes cifrados se multiplica por la inversa de la matriz clave y, finalmente, se toma su módulo 26 para obtener el mensaje original.

- Las líneas de la 68 a la 105 describen la función `inverse()`. Esta función se llama desde la función `decryption()` y se encarga de obtener la inversa de la matriz empleada para la encriptación, que servirá para realizar la desencriptación.

La ejecución del Código 4.1 usando los mismos datos del ejemplo del funcionamiento del *Hill Cipher* descrito en el apartado anterior, da como salida por consola los mensajes que se muestran en la Figura 4.3.

```
Enter 3x3 matrix for key (It should be inversible):
6 24 1
13 16 10
20 17 15

Enter a 3 letter string: act

Encrypted string is: poh

Inverse Matrix is:
0.158730 -0.777778 0.507937
0.011338 0.158730 -0.106576
-0.224490 0.857143 -0.489796

Decrypted string is: act
```

Figura 4.3: Salida por consola del programa *Hill cipher* en C

El programa anterior, se debe adaptar para su posterior integración en el componente *predictor*. Para ello en primer lugar se debe plantear la manera de conectar el código anterior y el código *SystemVerilog* empleando la interfaz DPI. Es decir, se debe buscar los diferentes tipos de variables utilizadas para el intercambio de datos y plantear para cada una de ellas un formato que permita el intercambio entre los dos dominios de la interfaz DPI. Por ello, se deben de desarrollar varias pruebas con el objetivo de verificar el intercambio de datos a través de la interfaz. El modelo de referencia adaptado para la integración se muestra en el apartado 4.4.9.2.1. de este capítulo.

4.4. Entorno UVM generado para la integración del código C en el componente *predictor*

El Código 4.2 muestra el contenido del fichero `hillcipher_pkg`. Este fichero contiene la implementación del paquete (*package)* de todas las clases que conforman el entorno generado. Este tipo de estructuras (*package*) en *SystemVerilog* son muy adecuadas ya que permiten su reusabilidad y, además, generan un *namespace* propio y sin posibilidades de cometer errores por duplicidad o igualdad de nombres en el entorno. En los siguientes apartados se mostrarán las clases necesarias para poder explicar la integración del modelo de referencia descrito en C en el componente *predictor* con el objetivo de realizar la verificación funcional mediante dicho modelo, a exepción de la clase `hillcipher_sequence` la cual se mostrará en el Anexo I.

```
1   package hillcipher_pkg;
2       import uvm_pkg::*;
3       `include "uvm_macros.svh"
4       `include "data_packet.sv"
5       `include "hillcipher_sequence_config.sv"
6       `include "hillcipher_sequence.sv"
7       `include "hillcipher_sequencer.sv"
8       `include "hillcipher_driver.sv"
9       `include "hillcipher_monitor.sv"
10      `include "hillcipher_agent.sv"
11      `include "sb_predictor.sv"
12      `include "sb_comparator.sv"
13      `include "hillcipher_scoreboard.sv"
14      `include "hillcipher_env.sv"
15      `include "dut_env.sv"
16      `include "test_lib.sv"
17  endpackage: hillcipher_pkg
```

Código 4.2: *Package* UVM con las clase que conforman el entorno implementado.

4.4.1. Módulo DUV del *Hill Cipher*

Cabe destacar que el diseño RTL del DUV no es propósito del presente TFG. Por tanto, el DUV se tratará como una caja negra con una interfaz de entrada a la que se le envía la información de entrada y una interfaz de salida por la que el DUV envía los datos procesados. En este caso particular, el DUV implementado saca, de forma retardada, los mismos datos que tiene en la

interfaz de entrada, por lo que se presupone que el DUV encripta y desencripta correctamente, pues las señales de entrada a encriptar son directamente las salidas que corresponden a los datos sin encriptar. El Código 4.3 muestra el `modulo hillcipher` que corresponde con el DUV usado en este ejemplo.

```
1   module hillcipher(input       clk,
2                     input       rst,
3                     input [7:0] adata,
4                     input       adata_en,
5                     input [7:0] msg,
6                     input       msg_en,
7                     input [1:0] msg_or_adata,
8                     output reg [7:0] odata,
9                     output reg       odata_en,
10                    output reg [7:0] omsg,
11                    output reg       omsg_en);
12
13     always @(posedge clk or rst)
14       if (rst) begin
15         odata <= 0;
16         odata_en <= 0;
17         omsg <= 0;
18         omsg_en <= 0;
19
20       end
21       else begin
22         odata <= adata;
23         odata_en <= adata_en;
24         omsg <= msg;
25         omsg_en <= msg_en;
26       end
27  endmodule:hillcipher
```

Código 4.3: Implementación del `modulo hillcipher (`DUV).

En el Código 4.3 se observa cómo el DUV recibe las señales de entrada y la señal de sincronismo con la señal de reloj. En cada flanco de subida de la señal `clk`, línea 13, en caso de no estar activa la señal de reset, línea 14, asigna estas entradas a las salidas, asignaciones correspondientes a las líneas 22 a 25.

4.4.2. Transacciones

Para el módulo `hillcipher` la implementación de las transacciones en este ejemplo se ha hecho definiendo una clase denominada `data_packet` y

empleada para modelar las transacciones del módulo `hillcipher`. El Código 4.4 muestra la clase `data_packet`, y sus campos:

- `elem_in`: este campo representa cada elemento de la matriz llave empleada para la encriptación.

- `msg_in`: este campo representa cada carácter a encriptar. Es decir, corresponde a cada uno de los caracteres del mensaje.

- `elem_out`: este campo representa los elementos de la matriz desencriptada.

- `msg_out`: este campo representa los caracteres del mensaje desencriptado.

- `msg_or_elem`: este campo se emplea para distinguir entre un elemento de la matriz o un carácter del mensaje.

```
1   class data_packet extends uvm_sequence_item;
2
3       rand  bit [7:0] elem_in;
4       rand  bit [7:0] msg_in;
5       rand  bit [7:0] elem_out;
6       rand  bit [7:0] msg_out;
7       rand  bit [1:0] msg_or_elem;
8
9      `uvm_object_utils_begin(data_packet)
10     `uvm_field_int(elem_in, UVM_DEFAULT)
11     `uvm_field_int(msg_in, UVM_DEFAULT)
12     `uvm_field_int(elem_out, UVM_DEFAULT)
13     `uvm_field_int(msg_out, UVM_DEFAULT)
14     `uvm_field_int(msg_or_elem, UVM_DEFAULT)
15     `uvm_object_utils_end
16
17     function new(string name = "data_packet");
18        super.new(name);
19     endfunction: new
20
21     virtual task displayAll( );
22     `uvm_info("DataPacket", $sformatf("elem_in= %0h msg_in= %0h
23      elem_out = %0h msg_out = %0h msg_or_elem= %0h", elem_in, msg_in,
24      elem_out, msg_out,   msg_or_elem), UVM_LOW)
25     endtask: displayAll
26  endclass: data_packet
```

Código 4.4: Implementación de la clase `data_packet`.

4.4.3. Secuencias del módulo *Hill Cipher*

Como se comentó en el apartado del cifrado de *Hill Cipher* para cifrar *n* caracteres se necesita una matriz de tamaño *nxn*. En este ejemplo en concreto, por simplificación, se implementó con una matriz de 3x3 para enviar 3 caracteres. Esto implica que una secuencia estará formada por los 9 elementos de la matriz (array bidimensional en el caso del código C y *SystemVerilog*) y los 3 caracteres del *array*, que representan la parte del mensaje a cifrar. Por lo tanto, una secuencia estará formada por los elementos y caracteres comentados, y sus respectivas señales de habilitación. La Figura 4.4 muestra las transacciones contenidas en una secuencia.

a1	a2	a3	a4	a5	a6	a7	a8	a9	msg1	msg2	msg3

Figura 4.4: Composición de una secuencia *Hill Cipher*

Los valores asignados a los elementos que conforman la secuencia se obtienen empleando un fichero cuyo nombre se ha parametrizado mediante un objeto de configuración. El Código 4.5 muestra el contenido del fichero de configuración utilizado en este ejemplo. Este fichero se compone de los limites que marcan el numero de caracteres y el número de elementos de la matriz, ademas de los valores de de los elementos y caracteres que forman la secuencia *Hill Cipher*.

```
1  3 9
2  a c t
3  6 24 1
4  13 16 10
5  20 17 15
```

Código 4.5: Fichero de configuración para la secuencia *Hill Cipher*

Como muestra el Código 4.5 el fichero de se compone de 5 líneas:

- La línea 1 representa los límites empleados en los bucles `for` para recorrer el fichero e ir asignando los valores a las transacciones que formarán la secuencia *Hill Cipher*.

- La línea 2 representa los caracteres del mensaje a cifrar. Este mensaje, se corresponde con el mismo ejemplo empleado para la explicación del cifrado de *Hill Cipher*.

- Las líneas entre la 3 y la 5 representa los 9 valores de los elementos de la matriz llave empleada para el cifrado. Destacando también, que se trata de la misma matriz del ejemplo.

El Código 4.6 muestra la implementación de la clase `hillcipher_secuence`. Esta clase se encarga de crear las transacciones que forman una secuencia. Para ello lee los valores de la matriz y de los caracteres desde el fichero de configuración y los envía mediante dos bucles `for`, uno para enviar los 3 caracteres a cifrar y otro para enviar los 9 elementos de la matriz llave.

```
1   class hillcipher_sequence extends uvm_sequence#(data_packet);
2
3       hillcipher_sequence_config seq_config;
4
5       integer c, file, c1, file1;
6       bit [7:0]  msg[2:0];
7       bit [7:0]  array[8:0];
8       int msg_lenght, array_lenght;
9
10      `uvm_object_utils(hillcipher_sequence)
11
12      function new(string name = "hillcipher_sequence");
13         super.new(name);
14      endfunction:new
15
16
17      virtual task body( );
18       seq_config  = hillcipher_sequence_config::type_id::create
19       ("hillcipher_sequence_config");
20
21       if(!uvm_config_db#(hillcipher_sequence_config)::get
22       (.cntxt( null ), .inst_name( "" ), .field_name( "seq_config" ),
23        .value( seq_config ) ) )
24       `uvm_error( "seq_config", " not found" )
25
26        file = $fopen( seq_config.file_name,"r");
```

```
27      c=$fscanf(file, "%d %d", msg_lenght,array_lenght);
28
29      for(int i=0; i< msg_lenght; i++)begin
30          c=$fscanf(file, "%s", msg[i]);
31         `uvm_do_with(req, {req.msg_or_elem == 2'b10;
32          req.msg_in == msg[i];})
33      end
34
35      for(int j=0; j< array_lenght; j++)begin
36          c=$fscanf(file, "%f",array[j]);
37         `uvm_do_with(req, {req.msg_or_elem == 2'b01;
38          req.elem_in == array[j];})
39      end
40
41        `uvm_do_with(req, {req.msg_or_elem == 2'b00;})
42
43         $fclose(file);
44    endtask: body
45
46  endclass: hillcipher_sequence
```

Código 4.6: Implementación de la clase `hillcipher_sequence`.

Del Código 4.6 se destaca:

- En la línea 3 se declara un objeto de configuración.

- La tarea `body()`, líneas 17 a 44, comienza con el proceso de configuración que se lleva a cabo mediante un objeto de configuración que da valor al fichero y a los límites de la longitud del mensaje y del *array*, con un doble objetivo. Primeramente, se configuraran las secuencias empleando el fichero que la clase *test* configura para el *testcase* insertando el fichero con el mensaje y la matriz, para la prueba en concreto. En segundo lugar, se obtienen las dimensiones del mensaje y de la longitud del *array*.

- En las líneas 18 y 19 se crea el objeto de configuración `hillcipher_sequence_config`.

- Las líneas entre la 21 y la 24 emplean el mecanismo de configuración UVM, comentado en el Capítulo 2, para configurar el objeto. Cabe destacar que, como se comentó, las secuencias son objetos dinámicos

y no componentes, por lo tanto en el mecanismo de configuración al emplear el metodo `get()` para el acceso a la base de datos se debe poner en el parametro contexto el valor `null`, `.cntxt(null)` en lugar del operador `this`, pues el objeto no tiene contexto.

- Las líneas 26 y 27 emplean los mecanismo de manejo de ficheros. La línea 26 abre el fichero empleando la función `$fopen()`, para abrir el fichero en modo lectura, esto se representa con el párametro `r` que se pasa a la función. La línea 27, mediante la función `$fscanf()`, obtiene los valores del fichero correspondientes al límite del mensaje y a la dimensión de la matriz. De forma similar, las líneas 30 y 36 emplean la función `$fscanf()`, para dar valores a la transacciones de la secuencia.

4.4.4. La interfaz `hillcipher_if`

La interfaz `hillcipher_if` permite al componente *driver* y el componente *monitor* comunicarse con el DUV. El Código 4.7 muestra la interfaz implementada para el entorno. Esta interfaz, además de las señales necesarias para enviar los caracteres y elementos para el cifrado, dispone de las señales de habilitación, `elem_in_en`, `elem_out_en`, `msg_in_en` y `msg_out_en`, necesarias para implementar el protocolo de las transacciones. Cabe destacar que todas las variables se declaran de tipo `logic`, este tipo de dato en *SystemVerilog* resuelve problemas que aparecen con las asignaciones entre variables tipo `reg` y puertos de entrada/salida.

```
1   interface hillcipher_if(input logic clk, rst);
2
3      logic [7:0] elem_in;
4      logic       elem_in_en;
5      logic [7:0] msg_in;
6      logic       msg_in_en;
7      logic [7:0] elem_out;
8      logic       elem_out_en;
9      logic [7:0] msg_out;
10     logic       msg_out_en;
11     logic [1:0] msg_or_elem;
12
```

```
13 endinterface: hillcipher_if
```

Código 4.7: Implementación de la interfaz `hillcipher_if`.

Del Código 4.7 cabe destacar que en el entorno generado se crearán dos interfaces a partir de esta clase. Una interfaz de entrada para enviar los estimulos al DUV, y otra interfaz de salida para las respuestas de salida ante los estímulos de entrada del DUV.

4.4.5. El componente `hillcipher_driver`

El componente `hillcipher_driver` se encarga de convertir las transacciones en estímulos RTL para estimular al DUV. Para ello, como se comentó en el Capítulo 2, en la fase `built_phase`, líneas entre la 10 a la 17 del Código 4.8, se configura la interfaz virtual. Seguidamente en la fase `run_phase`, líneas 19 a 24, se define el comportamiento de la clase. En esta tarea se ejecutan dos tareas en paralelo mediante las sentencias `fork` y `join`. Estas dos tareas, la primera llamada `reset()`, se encarga de inicializar las señales del DUV y la segunda, `get_and_drive()`, solicita transacciones al componente *sequencer* mediante el protocolo de *hanshake*, comentado en el Capítulo 2, y los envía mediante otra tarea denominada `drive_packet(data_packet pkt)` que se encarga de activar las señales de comunicación con el DUV conforme al protocolo definido. El protocolo implementado comprueba en cada ciclo de reloj el valor del campo `msg_or_elem` definido en la clase `data_packet`. Es decir, mediante este campo el componente *driver* puede distinguir si la transacción contiene un elemento de la matriz o un carácter del mensaje. En función del valor de este campo se activa la correspondiente señal de habilitación, `msg_in_en` o `elem_in_en`.

```
1  class hillcipher_driver extends uvm_driver #(data_packet);
2     virtual hillcipher_if vif;
3
4     `uvm_component_utils(hillcipher_driver)
5
6     function new(string name, uvm_component parent);
7        super.new(name, parent);
```

```
8     endfunction: new
9
10    function void build_phase(uvm_phase phase);
11       super.build_phase(phase);
12       if(!uvm_config_db#(virtual hillcipher_if)::get(this, "",
13       "in_intf", vif))
14      `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
15       get_full_name( ), ".vif"})
16      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
17    endfunction
18
19    virtual task run_phase(uvm_phase phase);
20       fork
21          reset( );
22          get_and_drive( );
23       join
24    endtask: run_phase
25
26    virtual task reset( );
27      `uvm_info(get_type_name( ), "Resetting signals ... ", UVM_LOW)
28       forever begin
29          @(posedge vif.rst);
30          vif.elem_out <= 8'b0;
31          vif.elem_out_en <= 1'b0;
32          vif.msg_out <= 8'b0;
33          vif.msg_out_en <= 1'b0;
34       end
35    endtask: reset
36
37    virtual task get_and_drive( );
38      `uvm_info(get_type_name( ), "Reciving signals ... ", UVM_LOW)
39       forever begin
40          @(posedge vif.clk);
41          while(vif.rst!= 1'b1) begin
42             seq_item_port.get_next_item(req);
43             drive_packet(req);
44             seq_item_port.item_done( );
45          end
46       end
47    endtask: get_and_drive
48
49    virtual task drive_packet(data_packet pkt);
50       @(posedge vif.clk);
51        if(pkt.msg_or_elem == 2'b10) begin
52          vif.msg_in_en <= 1'b1;
53          vif.elem_in_en <= 1'b0;
54          vif.msg_in <= pkt.msg_in;
55          vif.msg_or_elem <= pkt.msg_or_elem;
56        end
57        else if (pkt.msg_or_elem == 2'b01)begin
58          vif.elem_in_en <= 1'b1;
59          vif.msg_in_en <= 1'b0;
60          vif.elem_in <= pkt.elem_in;
61          vif.msg_or_elem <= pkt.msg_or_elem;
62          end
63          else begin
64          vif.elem_in_en <= 1'b0;
65          vif.msg_in_en <= 1'b0;
66          vif.msg_or_elem <= pkt.msg_or_elem;
67          end
68
```

```
69      endtask
70  endclass:hillcipher_driver
```

Código 4.8: Implementación del componente `hillcipher_driver`.

Del Código 4.8 se destaca lo comentado en los siguientes puntos:

- Las líneas de la 10 a la 17 implementan la fase `built_phase`. En esta fase, como se comentó, se configura la *interfaz virtual*.

- Las líneas entre la 19 y 24 describen la fase `run_phase` y como se comentó se implementa con un `fork` y `join` para ejecutar dos tareas en paralelo, la tarea `reset( )` y la tarea `get_and_drive( )`.

- Las líneas de la 26 a la 35 describen la tarea `reset()`, empleada para inicializar las señales.

- Las líneas entre la 37 y 47 describen la tarea `get_and_drive( )`. En esta tarea se implementa el protocolo de *hanshake* con el componente secuencer para ir solicitándole transacciones que seguidamente se envían a la tarea `drive_packet(data_packet pkt)`.

- Las líneas entre la 49 y la 69 implementan la tarea `drive_packet(data_packet pkt)`, esta tarea está parametrizada con el tipo de transacción a manejar por el componente *driver* y se encarga de extraer los valores de las tranzacciones para asignárselos a las señales de la interfaz virtual que comunican con el DUV conforme al protocolo establecido.

4.4.6. El componente `hillcipher_monitor`

El componente *hillcipher_monitor* va a realizar la función inversa a la del componente *driver*. Es decir, se encargar de transformar las señales que muestrea de la interfaz conforme al protocolo y las transforma en transacciones para enviarlas al componente *scoreboard,* donde se realizará la verificación funcional. Por ello, en el entorno implementado se deben disponer de dos componentes *monitor*, uno ubicado en el componente *agent* activo y encargado de recibir las transacciones que el componente *driver* envía al DUV para, posteriormente enviarlas al componente *scoreboard*. El componente *scoreboard*, haciendo uso de la clase *predictor* realizará las predicciones mediante llamadas al modelo de referencia comentado anteriormente. Por otra parte, se debe disponer de un componente *monitor* ubicado en el componente *agent* pasivo con el propósito de monitorizar las salidas actuales del DUV, las cuales son el objetivo del proceso de verificación. Para ello, la clase *monitor* se diseña de forma parametrizada permitiendo crear o hacer referencia a un componente *monitor* activo o pasivo, según el propósito comentado anteriormente. Esta parametrización se ha implementado a través del mecanismo de configuración de la metología UVM, en concreto empleando la variable `uvm_active_passive_enum`, línea 4 del Código 4.9. Mediante esta variable durante la fase `run_phase` se configura el componente *monitor* activo de entrada, línea 49 en este ejemplo, siendo activo y por tanto activa las llamadas a las tareas `collect_msg_in( )`, `collect_arrayb_in()`, `collect_idle()`, líneas 51 a 53. Por el contrario, si se crea como componente un monitor pasivo, este tendrá como función el monitorizar la interfaz de salida del DUV empleando las tareas `collect_msg_out( )`, `collect_arrayb_out()`, líneas 58 y 59.

```
1   class hillcipher_monitor extends uvm_monitor;
2      virtual hillcipher_if vif;
3      string monitor_intf;
4      protected uvm_active_passive_enum  is_monitor_active =
5      UVM_ACTIVE;
6      int num_pkts;
7
8      uvm_analysis_port #(data_packet) item_collected_port;
9      data_packet data_collected;
10     data_packet data_clone;
```

```
11
12    `uvm_component_utils_begin(hillcipher_monitor)
13    `uvm_field_enum(uvm_active_passive_enum, is_monitor_active,
14     UVM_ALL_ON)
15    `uvm_component_utils_end
16
17     function new(string name, uvm_component parent);
18        super.new(name, parent);
19     endfunction: new
20
21     function void build_phase(uvm_phase phase);
22        super.build_phase(phase);
23        if(!uvm_config_db#(uvm_active_passive_enum)::get(this, "",
24          "is_monitor_active", is_monitor_active))
25          `uvm_info(get_full_name( ), "Didn't get the is_monitor_active
26          field.", UVM_LOW)
27        if(!uvm_config_db#(string)::get(this, "", "monitor_intf",
28           monitor_intf))
29          `uvm_fatal("NOSTRING", {"Need interface name for: ",
30          get_full_name( ), ".monitor_intf"})
31          `uvm_info(get_type_name( ), $sformatf("INTERFACE USED = %0s",
32          monitor_intf), UVM_LOW)
33        if(!uvm_config_db#(virtual hillcipher_if)::get(this, "",
34          monitor_intf, vif))
35          `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
36          get_full_name( ), ".vif"})
37        item_collected_port = new("item_collected_port", this);
38        data_collected =
39         data_packet::type_id::create("data_collected");
40        data_clone = data_packet::type_id::create("data_clone");
41        `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
42     endfunction: build_phase
43
44     virtual task run_phase(uvm_phase phase);
45      if(is_monitor_active == UVM_ACTIVE) begin
46        fork
47         collect_msg_in( );
48         collect_arrayb_in();
49         collect_idle();
50        join
51      end
52      else begin
53        fork
54         collect_msg_out( );
55         collect_arrayb_out();
56        join
57      end
58     endtask: run_phase
59
60    virtual task collect_msg_in( );
61        forever begin
62             @(posedge vif.clk);
63             if (vif.msg_in_en) begin
64                data_collected.msg_in = vif.msg_in;
65            data_collected.msg_or_elem = vif.msg_or_elem;
66                num_pkts++;
67                $cast(data_clone, data_collected.clone( ));
68                item_collected_port.write(data_clone);
69            end
70        end
71    endtask: collect_msg_in
```

```
72
73    virtual task collect_arrayb_in( );
74        forever begin
75              @(posedge vif.clk);
76           if (vif.elem_in_en) begin
77             data_collected.elem_in = vif.elem_in;
78             data_collected.msg_or_elem = vif.msg_or_elem;
79             num_pkts++;
80             $cast(data_clone, data_collected.clone( ));
81             item_collected_port.write(data_clone);
82           end
83        end
84    endtask: collect_arrayb_in
85
86    virtual task collect_idle();
87     forever begin
88              wait(vif.msg_in_en)
89              wait((!vif.elem_in_en) && (!vif.msg_in_en))
90              data_collected.msg_or_elem = vif.msg_or_elem;
91              num_pkts++;
92              $cast(data_clone, data_collected.clone( ));
93        `uvm_info(get_type_name( ), $psprintf("REPORT: DATA IDLE = %0d
94         REPORT:TIME = %0d", data_clone.msg_or_elem, $time), UVM_LOW)
95              item_collected_port.write(data_clone);
96          end
97    endtask
98
99    virtual task collect_msg_out( );
100      forever begin
101        @(posedge vif.clk);
102          if (vif.msg_out_en) begin
103            data_collected.msg_out = vif.msg_out;
104            data_collected.msg_or_elem = 2'b10;
105            num_pkts++;
106            $cast(data_clone, data_collected.clone( ));
107            item_collected_port.write(data_clone);
108         end
109       end
110   endtask: collect_msg_out
111
112   virtual task collect_arrayb_out( );
113     forever begin
114        @(posedge vif.clk);
115          if (vif.elem_out_en) begin
116            data_collected.elem_out = vif.elem_out;
117            data_collected.msg_or_elem = 2'b01;
118            num_pkts++;
119            $cast(data_clone, data_collected.clone( ));
120            item_collected_port.write(data_clone);
121          end
122       end
123    endtask: collect_arrayb_out
124
125    virtual function void report_phase(uvm_phase phase);
126     `uvm_info(get_type_name( ), $psprintf("REPORT: COLLECTED
127      PACKETS = %0d", num_pkts), UVM_LOW)
128    endfunction: report_phase
129
130 endclass: hillcipher_monitor
```

Código 4.9: Implementación del componente `hillcipher_monitor`.

Del Código 4.9 se destacan los siguientes puntos de interés, que resaltan lo comentado anteriormente:

- En la línea 2 se declara una variable de tipo `string` con nombre `monitor_intf`. Esta variable se emplea en la fase `built_phase` para obtener la interfaz virtual, según se cree un componente monitor activo o pasivo. Es decir, como se comentó, en el entorno generado se necesitan dos interfazes una de entrada al DUV y una de salida, y por lo tanto, se debe asociar al componente monitor con su interfaz, monitor activo con la interfaz de entrada y monitor pasivo con la de salida.

- La línea 4 declara la variable de configuración `uvm_active_passive_enum is_monitor_active`. Esta variable, como se comentó, se emplea para generar un componente *monitor* activo o pasivo según su propósito en el componente *agent.*

- La línea 8 declara un puerto tipo `uvm_analysis_port` empleado para enviar las transacciones al componente *scoreboard*.

- Las líneas 9 y 10 declaran dos transacciones del tipo `data_packet`, comentadas en el apartado de las transaciones. Estas transacciones se emplean en el componente *scoreboard* para extraer, recolectar y enviar la información de interés necesaria para la verificación funcional.

- Las líneas entre la 21 y la 42 implementan la fase `built_phase`. En esta fase, como se comentó, según el valor de la variable `monitor_intf` se configura el componente *monitor* con la interfaz virtual de entrada (componente *monitor* del *agent* activo) o salida (componente *monitor* del *agent* pasivo) al DUV.

- Las líneas entre la 44 y la 58 implementan la fase `run_phase`. En esta fase se define la funcionalidad del componente monitor según el valor de la variable de configuración `is_monitor_active`, línea 45 . Es decir, si la variable adquiere el valor `UVM_ACTIVE`, se ejecutan en paralelo las tareas correspondientes al componente *monitor* ubicado en el *agent* activo, `collect_msg_in( )`, `collect_arrayb_in()`, `collect_idle()`. Mientras que si la variable en el mecanismo de configuración adquiere el valor `UVM_PASIVE` se ejecutan en paralelo las tareas correspondientes al componente *monitor* ubicado en el *agent* pasivo, `collect_msg_out()`, `collect_arrayb_out()`.

- En las líneas de la 60 a la 71 se implementa la tarea `collect_msg_in( )`. Esta tarea se encarga de recoger la información correspondiente a los caracteres, que representan el mensaje a cifrar, a partir de los estímulos enviados al DUV, para finalmente transformar éstos en transacciones para enviarlas al componente *scoreboard.* Para ello emplea la función `write()` que implementa el puerto `analisis_export` del componente *scoreboard*.

- En las líneas de la 73 a la 84 se implementa la tarea `collect_arrayb_in()`. Esta tarea, de forma similar a la tarea definida anteriormente, se encarga de recoger la información correspondiente a la matriz empleada para el cifrado.

- Las líneas entre la 86 y 97 implementan la tarea `collect_idle()`. Esta tarea se encarga de esperar por la última transacción que indica que es el final de la secuencia.

- Las líneas entre la 99 y la 110 implementan la tarea `collect_msg_out( )`. Esta tarea se encarga de recoger los caracteres correspondientes a la interfaz de salida del DUV.

- Las líneas entre la 112 y la 123 implementan la tarea `collect_arrayb_out()`. Esta tarea se encarga de recoger los elementos correspondientes a la interfaz de salida del DUV.

- Por ultimo, en las líneas entre la 125 y 128 se implementa la fase `report_phase`. Esta fase emplea el mecanismo de configuración y una variable de tipo `int`, llamada `num_pakts`, para informar del número de transacciones recolectadas por cada componente *monitor*, debiendo de coincidir en ambos.

4.4.7. El componente `hillcipher_agent`

El componente *agent* de la metodología UVM, como se comentó en el Capítulo 2, es un contenedor de componentes para interactuar con el DUV conforme a un determinado protocolo. El componente *agent* implementado en el Código 4.10 utiliza el mecanismo de configuración de forma que este mecanismo le permite generar un componente *agent* activo, formado por los componentes, *sequencer, driver y monitor,* o un componente *agent* pasivo compuesto únicamente por un componente *monitor.*

```
1   class hillcipher_agent extends uvm_agent;
2
3      protected uvm_active_passive_enum is_agent_active = UVM_ACTIVE;
4
5      hillcipher_sequencer sequencer;
6      hillcipher_driver    driver;
7      hillcipher_monitor   monitor;
8
9      `uvm_component_utils_begin(hillcipher_agent)
10     `uvm_field_enum(uvm_active_passive_enum, is_agent_active,
11      UVM_ALL_ON)
12     `uvm_component_utils_end
13
14     function new(string name, uvm_component parent);
15        super.new(name, parent);
16     endfunction
17
18     function void build_phase(uvm_phase phase);
19        super.build_phase(phase);
20        if(!uvm_config_db#(uvm_active_passive_enum)::get(this, "",
21          "is_agent_active", is_agent_active)) begin
```

```
22          `uvm_info(get_full_name( ), "Didn't get the is_agent_active
23           field.", UVM_LOW)
24        end
25        if(is_agent_active == UVM_ACTIVE) begin
26           sequencer = hillcipher_sequencer::type_id::create(
27           "sequencer", this);
28           driver = hillcipher_driver::type_id::create("driver", this);
29        end
30
31        monitor = hillcipher_monitor::type_id::create("monitor", this);
32
33      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
34     endfunction: build_phase
35
36     function void connect_phase(uvm_phase phase);
37        if(is_agent_active == UVM_ACTIVE)
38          driver.seq_item_port.connect(sequencer.seq_item_export);
39         `uvm_info(get_full_name( ), "Connect stage complete.", UVM_LOW)
40     endfunction: connect_phase
41
42  endclass: hillcipher_agent
```

Código 4.10: Implementación del componente `hillcipher_agent`.

Del Código 4.10 se destacan los siguientes puntos de interés:

- La línea 3 declara la variable de configuración, `uvm_active_passive_enum is_agent_active`. Esta variable se empleará tanto en la fase `built_phase` como en la fase `conect_phase` para la configuración del componente de acuerdo a las necesidades del entorno.

- En la líneas 5, 6 y 7 se declaran los componentes que pueden forman parte de del componente *agent* en función de la variable de configuración comentada anteriormente.

- Las líneas entre la 18 y 34 implementan la fase `built_phase`. En esta fase se configura la variable `uvm_active_passive_enum is_agent_active`, empleando el mecanismo de configuración para obtener el valor de la variable que se introdujo en la base de datos, que permita generar el componente *agent* activo o pasivo, según corresponda.

- Finalmente, en las líneas entre la 36 y la 40 se implementa la fase `connect_phase`. Esta fase se encarga de conectar a los componentes que conformarán el componente *agent*, con las conexiones tipo TLM comentadas en el Capítulo 2.

4.4.8. El componente `hillcipher_env`

En cuanto al entorno, se ha optado por generar un entorno formado, en primer lugar, por un solo *agent*, activo o pasivo según la configuración del mismo. Esto se ha hecho con el objetivo de generar dos entornos, uno encargado de manejar las entradas del DUV y otro de manejar las salida del mismo. Estos entornos (o sub-entornos) se referencian en un entorno global que contendrá a estos entornos y al componente *scoreboard*, conformando el entorno completo del DUV. La clase `hillcipher_env` mostrada en el Código 4.11 simplemente crea o referencia, haciendo uso de la *Factory*, un componente *agent*.

```
1   class hillcipher_env extends uvm_env;
2
3      hillcipher_agent agent;
4
5     `uvm_component_utils(hillcipher_env)
6
7     function new(string name, uvm_component parent);
8        super.new(name, parent);
9     endfunction
10
11    function void build_phase(uvm_phase phase);
12      super.build_phase(phase);
13      agent = hillcipher_agent::type_id::create("agent",this);
14     `uvm_info(get_full_name( ),"Build stage complete",UVM_LOW)
15    endfunction: build_phase
16
17  endclass: hillcipher_env
```

Código 4.11: Implementación del componente `hillcipher_env`.

4.4.9. Arquitectura implementada del componente *scoreboard*

Como se comentó, la arquitectura empleada está separada en dos tarea, la tarea de predicción (componente *predictor*) y la tarea de comparación y evaluación (componente *comparador*). La tarea de predicción está implementada por el componente *predictor* cuya respuesta se compara en el componente *comparator*, respuesta real de DUV, para verificar la funcionalidad del diseño. La arquitectura implementada para llevar a cabo la integración del modelo de referencia descrito en C se muestra en la Figura 4.4. Cabe destacar que sobre la arquitectura se han realizado algunos cambios en el componente *sb_comparator*, pues en la arquitectura mostrada el componente *sb_comparator* implementa las funciones `write()` implícitamente mediante las *FIFOs*, que como se comentó en el Capítulo 2, es una de las formas de implementar un puerto `uvm_analisis_export`.

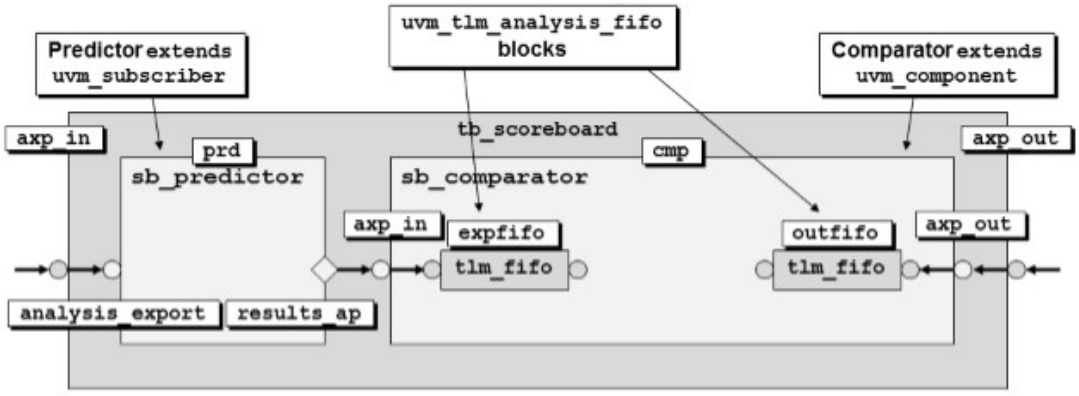

Figura 4.5: Arquitectura implementada del componente *scoreboard*.

En la Figura 4.5 se muestra un ejemplo de la implementación del componente `sb_comparator` que se corresponde con la solucion empleada en este TFG para implementar dos funciones `write()` en un mismo componente.

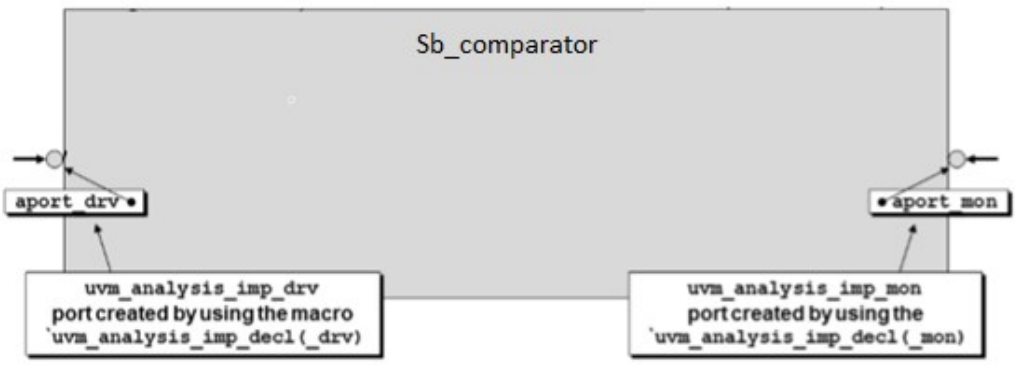

Figura 4.6: Ejemplo de solución para implementar dos `uvm_analysys_imp_drv` en un mismo componente.

Como se comentó en el Capítulo 2, la implementación de un puerto `uvm_analisys_export` se lleva a cabo implementando el método `write()`. Sin embargo el componente `sb_comparator` de la arquitectura *scoreboard,* mostrada en la Figura 4.6, utiliza dos puertos de implementación. En teoría cada puerto de implementación debe tener su propio método `write()`, pero si hay dos o más puertos de implementación, se requiere que el componente que implemente dichos puertos, tenga dos métodos `write()`, lo cual no está permitido. Para resolver este problema, la metodología UVM proporciona la macro `` `uvm_analysis_imp_decl(_suffix) `` que se utiliza para declarar cada puerto `uvm_analysis_imp` con nombre único que se pasa como parámetro a la macro (`_suffix)`. Los nombres de sufijo de macro son necesarios en varios lugares del código, los lugares requeridos son:

- En las declaraciones del puerto `uvm_analysis_imp_suffix`.
- Como parte del nombre de la función `write_suffix`.
- En el nombre de la declaración del puerto.
- En el nombre, cuando los puertos declarados se construyen en la fase `build_phase`.

De esta forma el uso de la macro `uvm_analysis_imp_decl() permite la construcción de dos puertos de implementación de análisis con sus dos funciones write() correspondientes (cada una con su nombre asociado).

4.4.9.1. El componente hillcipher_scoreboard

En el Código 4.12 se muestra la clase implementada del componente hillcipher_scoreboard correspondiente a la arquitectura de la Figura 4.5.

```
1   class hillcipher_scoreboard extends uvm_scoreboard;
2
3      uvm_analysis_export #(data_packet) exp_in;
4      uvm_analysis_export #(data_packet) exp_out;
5      sb_predictor                       prd;
6      sb_comparator                      cmp;
7
8     `uvm_component_utils(hillcipher_scoreboard)
9
10    function new(string name, uvm_component parent);
11      super.new( name, parent );
12    endfunction
13
14    function void build_phase(uvm_phase phase);
15      super.build_phase(phase);
16      exp_in  = new("exp_in",  this);
17      exp_out = new("exp_out", this);
18      prd     = sb_predictor::type_id::create("prd", this);
19      cmp     = sb_comparator::type_id::create("cmp", this);
20    endfunction
21
22    function void connect_phase( uvm_phase phase );
23       super.connect_phase(phase);
24      // Connect predictor & comparator to respective analysis exports
25      exp_in.connect(prd.analysis_export);
26      exp_out.connect(cmp.exp_out);
27      // Connect predictor to comparator
28      prd.results_ap.connect(cmp.exp_in);
29    endfunction
30
31  endclass: hillcipher_scoreboard
```

Código 4.12: Implementación del componente hillcipher_scoreboard.

Del Código 4.12 se detallan los siguientes puntos:

- Las líneas 3 y 4 declaran dos puertos tipo `uvm_analysis_export`. Estos puertos se emplean para recibir las transacciones del componente *agent* (componente *monitor* que alberga). El primer puerto, línea 3 `(exp_in)`, se asocia con el componente *predictor*. Y el segundo, línea 4 `(exp_out)`, se corresponde con el componente *comparator*.

- La línea 5 declara el componente `sb_predictor` y la línea 6 declara el componente `sb_comparator`.

- La líneas entre la 14 y la 20 implementan la fase `built_phase`. En esta fase se crean o referencian los componentes `sb_predictor` y `sb_comparator` y los puertos tipo `uvm_analysis_export` comentados.

- Por último, las líneas entre la 22 y la 29 describen la fase `connect_phase`. En esta fase se lleva a cabo la conexión de los puertos de los componentes que conforman el componente `hillcipher_scoreboard`. Es decir, se conectan los puertos de los componentes `sb_predictor` y `sb_comparator` con sus respectivos puertos *analysis export* (líneas 25 y 26) y, además, se conectan el componente `sb_predictor` con el componente `sb_comparator`, línea 28.

4.4.9.2. El componente `sb_predictor` y modelo de referencia descrito en C

El componente *predictor* recibirá la secuencia de *Hill Cipher* transacción a transacción y las almacena y gestiona empleando ficheros y haciendo uso de las funciones de manejo de ficheros de *SystemVerilog*. Una vez se han

obtenido todas las transacciones que conforman la secuencia y se han almacenado en el fichero, se procede a recorrer el fichero e ir asignando los valores en dos *arrays* empleados como argumento para realizar la llamada a la función implementada en C, la cual representa el modelo de referencia. Una vez se dispone de todos los caracteres y elementos de la secuencia se llama al modelo descrito en C. Este modelo se ejecuta en tiempo cero de simulación y retorna los valores mediante los argumentos de la función importada. Como se comentó en el Capítulo 3, estos argumentos pueden ser de tipo `input`, `ouput` o `inout`. Declarado dos argumentos `ouput`, se ha aprovechado la misma llamada para obtener de vuelta los valores de las predicciones. Seguidamente, se asignan a transaciones del tipo `data_packet` y nuevamente se hace uso de las comunicaciones TLM para pasar las predicciones al componente `sb_comparator`.

4.4.9.2.1. Modelo de referencia del algoritmo de *Hill Cipher*

Cabe destacar que el modelo de referencia C original no está estructurado para recibir transacciones una a una, por lo que la interfaz de DPI en la llamada al modelo de referencia consistirá en una sola llamada de función que lleva consigo un conjunto de datos completo. Por lo tanto, el componente `sb_predictor` recibirá el conjunto de transacciones entrantes y luego las pasará al modelo de referencia para el procesamiento por lotes. Es decir, en el código *SystemVerilog* del componente `sb_predictor` se almacenan las transacciones en un fichero y cuando se dispone de la secuencia completa se llama al modelo de referencia descrito en C.

Este modelo se integra en el componente `sb_predictor` y forma lo que se conoce como un *predictor*, el cual representa un modelo de referencia del DUV. Para su uso correcto, es necesario enviarle los mismo estímulos de entrada que se envían al DUV. De esta manera, el modelo de referencia proporcionará los datos de respuesta esperados que, por definición, son los datos correctos. En este caso el modelo de referencia utilizado está escrito en

lenguaje C, pero también podría estar descrito en otro lenguajes tales como C++ o *SystemC*.

El modelo de predicción para la verificación del diseño del módulo *Hill Cipher* se escribe en C, y *SystemVerilog* proporciona la Interfaz DPI para interactuar con dicho modelo. Este proceso requiere realizar definiciones de tipos coincidentes para pasar un valor a través de la interfaz DPI, la definición *SystemVerilog* y la definición C. El análisis del programa en C que implementa los algoritmos de *Hill Cipher* realizados en el punto anterior, desveló que se debía adaptar para su posterior integración en el componente *predictor*. Para ello, en primer lugar y tras varias pruebas, se definieron los tipos de datos que permiten un intercambio adecuado empleando la interfaz *SystemVerilog* DPI. En la Tabla 4.2 se muestran las equivalencias en los tipos de datos emplados en ambos dominios de la interfaz DPI.

Tipo de dato *SystemVerilog*	Tipo de dato C
`input shortreal dyn_arr[][]`	`const svOpenArrayHandle array_in`
`input byte s[]`	`const svOpenArrayHandle str`
`output shortreal arg[3][3]`	`float* arg`
`output byte men[3]`	`char* msg`

Tabla 4.2: Equivalencia de los tipos empleados en la interfaz DPI.

En segundo lugar, una vez definidos los tipos de datos a emplear, se ha implementado una función global, a la que se ha llamado `hillcipher()`. Esta función será la función a importar desde el componente *predictor* descrito en *SystemVerilog*. En la función se pasan por parámetro dos *arrays* con los valores a predecir y dos arrays para obtener los valores predichos. Dicha función pasa los parametros de entrada a la funcion `getKeyMessage()`, que obtiene de este modo un *array* bidimensional de 3x3 que representa la matriz llave, y un array de 3 caracteres que representa el mensaje a cifrar, enviados a través de la interfaz DPI desde el código descrito en *SystemVerilog* al código descrito en el lenguaje C. El Código 4.13 muestra la implementación del modelo de referencia, descrito en C, empleado para realizar las predicciones.

```
#include<stdio.h>
#include<math.h>
#include<svdpi.h>
#include<dpihill.h>

float encrypt[3][1], decrypt[3][1], a[3][3], b[3][3], mes[3][1],
c[3][3];
char men1[3], men2[3];

void hillcipher(const svOpenArrayHandle array_in, const
svOpenArrayHandle str, float* arg, char* msg);//funtion call from
SV
void inicializar();
void encryption();    //encrypts the message
void decryption();    //decrypts the message
void getKeyMessage(const svOpenArrayHandle array_in, const
svOpenArrayHandle str);//gets key and message from user
void inverse();        //finds inverse of key matrix

void hillcipher(const svOpenArrayHandle array_in, const
svOpenArrayHandle str, float* arg, char* msg) {
    inicializar();
    getKeyMessage( array_in,  str);
    encryption();
    decryption();
    //devuelvo a SV
    memcpy(msg,men1,sizeof(men1));
    memcpy(arg, a,sizeof(a));

}

void inicializar(){
int i,j;
for (i = 0; i< 3; i++) {
  men1[i]=0;
  men2[i]=0;
}
for (i = 0; i< 3; i++)
   for (j = 0; j< 3; j++) {
    a[i][j]=0;
    b[i][j]=0;
    c[i][j]=0;
}
for (i = 0; i< 3; i++)
   for (j = 0; j< 2; j++) {
    encrypt[i][j]=0;
    decrypt[i][j]=0;
    mes[i][j]=0;
}
}

void encryption() {
    int i, j, k;

    for(i = 0; i < 3; i++)
        for(j = 0; j < 1; j++)
            for(k = 0; k < 3; k++)
                encrypt[i][j] = encrypt[i][j] + a[i][k] *
mes[k][j];

    printf("\nEncrypted string is: ");
```

```
71      for(i = 0; i < 3; i++){
72          printf("%c", (char)(fmod(encrypt[i][0], 26) + 97));
73          men2[i]= (char)(fmod(encrypt[i][0], 26) + 97);
74  }
75
76  }
77
78  void decryption() {
79      int i, j, k;
80
81      inverse();
82
83      for(i = 0; i < 3; i++)
84        for(j = 0; j < 1; j++)
85           for(k = 0; k < 3; k++)
86             decrypt[i][j] = decrypt[i][j] + b[i][k]* encrypt[k][j];
87
88      printf("\nDecrypted string is: ");
89      for(i = 0; i < 3; i++){
90          printf("%c", (char)(fmod(decrypt[i][0], 26) + 97));
91          men1[i]= (char)(fmod(decrypt[i][0], 26) + 97);
92      }
93      printf("\n");
94  }
95
96  void getKeyMessage(const svOpenArrayHandle array_in, const
97  svOpenArrayHandle str) {
98      int i, j;
99      char msg[3];
100     printf("\n \t \t \t Golden reference prediction \n");
101     printf("\n\nEnter 3x3 matrix for key (It should be
102 inversible):\n");
103
104     for(i = 0; i < 3; i++){
105         for(j = 0; j < 3; j++) {
106             a[i][j] = *(float*)svGetArrElemPtr2(array_in, i, j);
107             c[i][j]= a[i][j];
108              printf("%f\t", c[i][j]);
109         }
110              printf("\n");
111
112         }
113
114     printf("\nEnter a 3 letter string: ");
115
116     for(i = 0; i < 3; i++){
117         msg[i]= *(char*)svGetArrElemPtr1(str, i);
118        mes[i][0] = msg[i] - 97;
119     printf ("%c", msg[i] );
120
121 }
122 }
123
124 void inverse() {
125     int i, j, k;
126     float p, q;
127
128     for(i = 0; i < 3; i++)
129         for(j = 0; j < 3; j++) {
130             if(i == j)
131                 b[i][j]=1;
```

```
132             else
133                 b[i][j]=0;
134         }
135
136     for(k = 0; k < 3; k++) {
137         for(i = 0; i < 3; i++) {
138             p = c[i][k];
139             q = c[k][k];
140
141             for(j = 0; j < 3; j++) {
142                 if(i != k) {
143                     c[i][j] = c[i][j]*q - p*c[k][j];
144                     b[i][j] = b[i][j]*q - p*b[k][j];
145                 }
146             }
147         }
148     }
149
150     for(i = 0; i < 3; i++)
151         for(j = 0; j < 3; j++)
152             b[i][j] = b[i][j] / c[i][i];
153
154     printf("\n\nInverse Matrix is:\n");
155     for(i = 0; i < 3; i++) {
156         for(j = 0; j < 3; j++)
157             printf("%f ", b[i][j]);
158
159         printf("\n");
160     }
161 }
```

Código 4.13: Implementación del modelo de referencia, descrito en C, empleado para realizar las predicciones.

Del Código 4.13 se destcan los siguientes puntos de interés que resaltan las modificaciones empleadas para adaptar el modelo de referencia:

- En la línea 2 se incluye la librería `<math.h>`, que permite acceso a la función que obtiene el módulo 26 del ejemplo de cifrado de *Hill Cipher.*

- La línea 3 incluye el archivo `<svdpi.h>`. Este archivo, como se comentó en el capítulo 3, proporciona compatibilidad entre los tipos que no tienen una representación directa entre los dos dominios de la interfaz DPI y da soporte a la interfaz.

- En la línea 4 se incluye el archivo `<dpihill.h>`. Este archivo es un archivo de conveniencia, es decir, no es obligatorio, pero es recomendado para un correcto uso de flujo DPI en la herramienta

QuestaSim. Este archivo se comenta en el flujo DPI que se emplea en la herramienta *QuestaSim* en el Capítulo 6.

- En las líneas entre la 6 y la 8 se declaran las variables globales empleadas en el modelo de referencia.

- Las líneas entre la 10 y la 18 declaran los prototipos de las funciones empleadas en el modelo de referencia, destacando las líneas 10 y 11 las cuales representan la función importada que se llamará desde el componente `sb_predictor`.

- En las líneas entre la 20 y la 30 se implementa la función importada que se llamará desde el componente `sb_predictor` del `tesbench`. Esta función hace de función global y recibe por parámetros los *arrays* con los valores para realizar la predicción y los *arrays* para devolver el resultado de dicha predicción y llama al resto de funciones que modelan el algoritmo de *Hill Cipher*. En primer lugar, llama a la función `inicializar()`. Esta función inicializa las variables globales, pues estas variables si no se inicializan pueden dar lugar a resultados erróneos al realizar dos o más llamadas concecutivas al modelo de referencia. En segundo lugar, llama a la función `getKeyMessage( array_in, str)`, línea 23, pasándole los parámetros de entrada. Seguidamente se llama a las funciones `encryption()` y `decryption()`, comentadas en el apartado 4.3.1. Finalmente, se devuelven los valores de las predicciones, consideradas como respuestas esperadas, al componente *predictor* para su posterior comparación con las respuestas actuales del DUV. Para devolver los valores, se emplean los argumentos declarados como `output` en la función importada y, además, se emplea la funciones `memcpy()`.

Como se puede apreciar en el Código 4.13 el algoritmo C empleado para implementar el modelo de referencia se trata de un modelo sin tiempos. Los

tesbenches UVM, como se comentó, están orientados a la transacciones y, aunque los modelos de referencia en lenguaje C también pueden estar orientados a transacciones, son modelos sin tiempo, que realizan alguna transformación en un conjunto de datos completo sin tener en cuenta cómo y cuándo esos datos se presentan en la implementación del hardware o en el *testbench*. El modelo de referencia implementado en este TFG recibirá un conjunto de datos para su procesamiento por lotes y devolverá el conjunto de datos considerados como los valores esperados.

4.4.9.2.2. El componente `sb_predictor`

El componente `sb_predictor` se deriva de la clase `uvm_subscriber` y representa el componente en el cual tiene lugar la integración del modelo de referencia descrito en C. En este ejemplo en concreto el modelo de referencia viene representado por el Código 4.13 mostrado anteriormente. El funcionamiento de este componente se implementa en la función `write()`, la cual copia las transacciónes correspondientes a los estímulos del DUV que le llegan del componente *agent* pasivo y las almacena en un fichero. Posteriormente llama al modelo de referencia, empleando la interfaz DPI, para el procesamiento por lotes. Es decir, se le pasa una secuencia de *Hill Cipher* completa (mensaje de tres caracteres y matriz de nueve elementos). Para ello, como se comentó en el Capítulo 3, se debe declarar la función importada descrita en lenguaje C. Cabe destacar que esta llamada se realiza en la función `write()`, se ejecuta en tiempo cero y devuelve los valores de la predicción en la misma llamada. Posteriormente, se vuelve a transformar los valores predichos en transacciones del tipo `data_packet`, que se envían al componente `sb_comparator` para su posterior comparación e informe de resultados. En el Código 4.14 se muestra la implementación del componente `sb_predictor`, en el cula se ha integrado el modelo de referencia descrito en C.

```
1   import "DPI-C" context function void hillcipher(input shortreal
2   dyn_arr[][],input byte s[],output shortreal arg[3][3],output byte
3   men[3]);
4    class sb_predictor extends uvm_subscriber #(data_packet);
5
6     int num_msg, num_elem;
7     integer fileh_predictor, f;
8     shortreal array_from_c[3][3], array_to_c[2:0][2:0];
9     byte msg_from_c[3],msg_to_c[3];
10    data_packet  data_collected_s, data_to_compare;
11    uvm_analysis_port #(data_packet) results_ap;
12
13   `uvm_component_utils(sb_predictor)
14
15    function new(string name, uvm_component parent);
16      super.new(name, parent);
17    endfunction
18
19    function void build_phase(uvm_phase phase);
20      super.build_phase(phase);
21      results_ap = new("results_ap", this);
22      fileh_predictor = $fopen("predictor_file.txt","w");
23      data_collected_s = data_packet::type_id:: create(
24      "data_collected_s");
25
26      data_to_compare = data_packet::type_id::create(
27      "data_to_compare");
28    endfunction
29
30    function void write(data_packet t);
31     if(t.msg_or_elem == 2'b10) begin
32      data_collected_s.msg_in = t.msg_in;
33     `uvm_info(get_type_name( ), $psprintf("REPORT: data_Colected_s
34      = %0d  REPORT:TIME = %0d", data_collected_s.msg_in, $time),
35      UVM_LOW)
36
37      $fdisplay(fileh_predictor, "%s", data_collected_s.msg_in);
38        num_msg++;
39     end
40     else if(t.msg_or_elem == 2'b01 )  begin
41       data_collected_s.elem_in = t.elem_in;
42      `uvm_info(get_type_name( ), $psprintf("REPORT: data_Colected_s
43       = %0d REPORT:TIME = %0d", data_collected_s.elem_in, $time),
44       UVM_LOW)
45       $fdisplay(fileh_predictor, "%f", data_collected_s.elem_in);
46        num_elem++;
47     end
48
49     if(t.msg_or_elem  == 2'b00) begin
50        $fclose(fileh_predictor);
51        fileh_predictor = $fopen("predictor_file.txt","r");
52        for(int i=0; i< num_msg; i++) begin
53        f=$fscanf(fileh_predictor, "%s", msg_to_c[i]);
54     end
55       for(int i=0; i< $sqrt(num_elem); i++)
56        for(int j=0; j< $sqrt(num_elem); j++) begin
57         f=$fscanf(fileh_predictor, "%f", array_to_c[i][j]);
58     end
59      num_msg=0;
60      num_elem=0;
70
```

```
71    hillcipher( array_to_c, msg_to_c, array_from_c, msg_from_c );
72       begin
73          for (int i = 0; i< $size(msg_from_c) ; i++) begin
74            $cast(data_to_compare.msg_in, msg_from_c[i]);
75            data_to_compare.msg_or_elem = 2'b10;
76     //  `uvm_info(get_type_name( ), $psprintf("REPORT:
77  data_to_compare = %0d  REPORT:TIME = %0d", data_to_compare.msg_in,
78  $time), UVM_LOW)
79             results_ap.write(data_to_compare);
80       end
81       end
82       begin
83          for (int i = 0; i< $size(array_from_c); i++)
84            for(int j=0; j< $size(array_from_c); j++) begin
85              $cast(data_to_compare.elem_in, array_from_c[i][j]);
86              data_to_compare.msg_or_elem = 2'b01;
87       //`uvm_info(get_type_name( ), $psprintf("REPORT:
88  data_to_compare = %0d  REPORT:TIME = %0d", data_to_compare.elem_in,
89  $time), UVM_LOW)
90             results_ap.write(data_to_compare);
91       end
92       end
93          $fclose(fileh_predictor);
94          fileh_predictor = $fopen("predictor_file.txt","w");
95       end
96    endfunction
97  endclass: sb_predictor
```

Código 4.14: Implementación del componente `sb_predictor`.

Del Código 4.14 se destacan los siguientes puntos de interés que resaltan lo comentado anteriormente.

- En las líneas 1, 2 y 3 se importa la función implementada en C. Cabe destacar que las declaraciones de importación se deben realizar fuera de la declaración de la clase.

- En las líneas de la 6 a la 9 se declaran las variables empleadas como argumentos para el intercambio de datos a través de la interfaz DPI, así como las variables empleadas en la clase `sb_predictor`.

- La línea 10 declara dos transacciones del tipo `data_packet`. La primera, para manejar las transacciones que obtiene del componente *agent* activo, que sirven para realizar las predicciones. La segunda, para enviar las predicciones al componente `sb_comparator`.

- La línea 11 declara un puerto tipo `uvm_analysis_port` para comunicar con el componente `sb_comparator` el cual deberá implementar la correspondiente función `write()`.

- En las líneas entre la 19 y la 28 se implementa la fase `built_phase`. En esta fase se crean las transacciones a manejar, se crean las conexiones tipo TLM y se abre el fichero que contiene los valores de la predicción.

- En las líneas entre la 30 y la 96 se implementa la función `write()` del puerto `uvm_analisys_export`, de esta clase, que recibe transacciones del puerto `uvm_analisys_port` del componente *monitor* activo. En esta función, según el valor de la variable `msg_or_elem`, se sabe si se recibe un carácter del mensaje o un elemento de la matriz. Además, si esta variable toma el valor `2'b00`, indica de este modo que se ha recibido la secuencia completa.

- La línea 71 muestra la llamada a la función C importada que representa el modelo de referencia. Cabe destacar que esta llamada para el código SystemVerilog se trata como una llamada nativa del propio lenguaje, y se ejecuta en tiempo 0 de simulación retornando los valores de las predicciones del algoritmo de cifrado de *Hill Cipher*.

4.4.9.2.3. El componente `sb_comparator`

El componente `sb_comparator`, el cual forma parte de la arquitectura del componente *scoreboard* se deriva de la clase `uvm_component`. Cabe destacar que se ha optado por obtener y gestionar las secuencias mediante el uso de ficheros en lugar de implementar los puertos `uvm_analisys_export` haciendo uso de FIFOs. El componente comparator, como muestra la Figura 4.5, implementa dos funciones `write()`. En estas funciones se reciben, en la

primera, las transacciones con los valores de las prediciones obtenidas del componente predictor, y en la segunda las transacciones obtenidas de la interfaz de salida del DUV. Una vez se dispone de todos los valores de la secuencia almacenados en ficheros, se procede a su comparación carácter por carácter y elemento por elemento para informar de las coincidencias o desigualdades del proceso de comparación. El Código 4.15 muestra la implementación del componente *sb_comparator.*

```
1   class sb_comparator extends uvm_component ;
2     `uvm_analysis_imp_decl( _active_monitor )
3     `uvm_analysis_imp_decl( _pasive_monitor )
4
5      int num_msg, num_elem, VECT_CNT, PASS_CNT, ERROR_CNT;
6      integer file_active_monitor, file_pasive_monitor, c, d;
7      byte msg_ma[3], msg_mp[3];
8      shortreal array_ma[3][3], array_mp[3][3];
9
10     uvm_analysis_imp_active_monitor #(data_packet, sb_comparator)
11     exp_in;
12     uvm_analysis_imp_pasive_monitor #(data_packet, sb_comparator)
13     exp_out;
14
15     data_packet data_ma;
16     data_packet data_mp;
17     data_packet exp_tr;
18     data_packet out_tr;
19
20    `uvm_component_utils(sb_comparator)
21
22    function new (string name, uvm_component parent);
23      super.new(name, parent);
24    endfunction
25
26    function void build_phase(uvm_phase phase);
27       super.build_phase(phase);
28       file_active_monitor = $fopen("msg1.txt","w");
29       file_pasive_monitor = $fopen("elem1.txt","w");
30
31       exp_in = new("aport_active_monitor", this);
32       exp_out = new("aport_pasive_monitor", this);
33
34       exp_tr = data_packet::type_id::create("exp_tr");
35       out_tr = data_packet::type_id::create("out_tr");
36
37       data_ma = data_packet::type_id::create("data_ma");
38       data_mp = data_packet::type_id::create("data_mp");
39    endfunction
40
41    function void write_active_monitor(data_packet t);
42     if(t.msg_or_elem == 2'b10) begin
43       data_ma.msg_in = t.msg_in;
44   // `uvm_info(get_type_name( ), $psprintf("REPORT: data_mp = %0d
45  REPORT:TIME = %0d", data_ma.msg_in, $time), UVM_LOW)
46       $fdisplay(file_active_monitor, "%s", data_ma.msg_in);
47     end
```

```
48      else if(t.msg_or_elem == 2'b01 )  begin
49        data_ma.elem_in = t.elem_in;
50    // `uvm_info(get_type_name( ), $psprintf("REPORT: data_ma = %0d
51    REPORT:TIME = %0d", data_ma.elem_in, $time), UVM_LOW)
52        $fdisplay(file_active_monitor, "%f", data_ma.elem_in);
53      end
54    endfunction
55
56    function void write_pasive_monitor(data_packet t);
57     if(t.msg_or_elem == 2'b10) begin
58       data_mp.msg_out = t.msg_out;
59    //`uvm_info(get_type_name( ), $psprintf("REPORT: data_mp = %0d
60    REPORT:TIME = %0d", data_mp.msg_in, $time), UVM_LOW)
70        $fdisplay(file_pasive_monitor, "%s", data_mp.msg_out);
71          num_msg++;
72      end
73      else if(t.msg_or_elem == 2'b01) begin
74        data_mp.elem_out = t.elem_out;
75     //`uvm_info(get_type_name( ), $psprintf("REPORT: data_mp = %0d
76    REPORT:TIME = %0d", data_mp.elem_in, $time), UVM_LOW)
77        $fdisplay(file_pasive_monitor, "%f", data_mp.elem_out);
78          num_elem++;
79      end
80    endfunction
81
82
83    task run_phase(uvm_phase phase);
84      compare_data();
85    endtask
86
87    virtual task compare_data();
88    forever begin
89        wait(num_msg == 3 && num_elem == 9)
90        $fclose(file_active_monitor);
91        file_active_monitor = $fopen("msg1.txt","r+");
92       `uvm_info("sb_comparator run task", "WAITING for expected
93        output", UVM_LOW)
94        $fclose(file_pasive_monitor);
95        file_pasive_monitor = $fopen("elem1.txt","r+");
96       `uvm_info("sb_comparator run task", "WAITING for actual
97        output", UVM_LOW)
98
99      compare_msg();
100     compare_elem();
101
102      $fclose(file_active_monitor);
103      file_active_monitor = $fopen("msg1.txt","w");
104      $fclose(file_pasive_monitor);
105      file_pasive_monitor = $fopen("elem1.txt", "w");
106    end
107   endtask
108
109
110   function void compare_msg();
111     for(int i=0; i< num_msg; i++) begin
112      c=$fscanf(file_active_monitor, "%s", msg_ma[i]);
113      d=$fscanf(file_pasive_monitor, "%s", msg_mp[i]);
114     if(msg_ma[i] == msg_mp[i])begin
115         PASS();
116      `uvm_info ("PASS_msg ", $sformatf("Expected=%s   Actual=%s
117       \n", msg_ma[i], msg_mp[i]), UVM_LOW)
```

```
118      end
119      else begin
120          ERROR();
121       `uvm_info ("ERROR_msg ", $sformatf("Expected=%s   Actual=%s
122        \n", msg_ma[i], msg_mp[i]), UVM_LOW)
123      end
124      end
125         num_msg=0;
126    endfunction
127
128    function void compare_elem();
129      for(int i=0; i< $sqrt(num_elem); i++)
130       for(int j=0; j< $sqrt(num_elem); j++) begin
131        c=$fscanf(file_active_monitor, "%f", array_ma[i][j]);
132        d=$fscanf(file_pasive_monitor, "%f", array_mp[i][j]);
133      if(array_ma[i][j] == array_mp[i][j])begin
134          PASS();
135       `uvm_info ("PASS_elem ", $sformatf("Expected=%f   Actual=%f
136        \n", array_ma[i][j], array_mp[i][j]), UVM_LOW)
137      end
138      else begin
139          ERROR();
140       `uvm_info ("ERROR_elem ", $sformatf("Expected=%f   Actual=%f
141        \n", array_ma[i][j], array_mp[i][j]), UVM_LOW)
142      end
143      end
144         num_elem=0;
145    endfunction
146
147    function void report_phase(uvm_phase phase);
148      super.report_phase(phase);
149      if (VECT_CNT && !ERROR_CNT)
150        `uvm_info(get_type_name(),
151         $sformatf("\n\n\n*** TEST PASSED - %0d vectors ran, %0d
152       vectors passed ***\n", VECT_CNT, PASS_CNT), UVM_LOW)
153      else
154        `uvm_info(get_type_name(),
155         $sformatf("\n\n\n*** TEST FAILED - %0d vectors ran, %0d
156         vectors passed, %0d vectors failed ***\n", VECT_CNT,
157         PASS_CNT,ERROR_CNT), UVM_LOW)
158    endfunction
159
160    function void PASS();
161      VECT_CNT++;
162      PASS_CNT++;
163    endfunction
164
165    function void ERROR();
166      VECT_CNT++;
167      ERROR_CNT++;
168    endfunction
169
170  endclass  sb_comparator
```

Código 4.15: Implementación del componente `sb_comparator`.

Del Código 4.15 se destacan los siguientes puntos de interés que detallan la implementación del mismo:

- En las líneas 1 y 2 se declaran las macros, comentadas anteriormente, `uvm_analysis_imp_decl(_name) para poder implementar dos funciones write() en el componente sb_comparator.

- En las líneas entre la 5 y la 8 declaran las variables empleadas para la comparación y información de resultados.

- Las líneas entre la 10 y la 13 declaran los puertos uvm_analysis_imp_ cada uno para la función write() correspondiente.

- En las líneas entre la 15 y la 18 se declaran las transacciones del tipo data_packet, a manejar en las comparaciones.

- Las líneas entre la 26 y la 39 implementan la built_phase. En esta fase se abren los ficheros, en modo lectura, a emplear para almacenar y comparar los valores de las transacciones. Además, se crean los puertos tipo TLM y las transacciones, data_packet.

- En las líneas entre la 41 y la 54 se implementa la función write_active_monitor(data_packet t). Esta función implementa el puerto uvm_analysis_export para recibir por parametro las transacciones, del tipo data_packet, con los valores de las prediciones. Para ello, el protocolo empleado hace uso del valor del campo t.msg_or_elem para distinguir entre un caracter o un elemento y, empleando la función $fdisplay de manejo de ficheros, almacena los valores en un fichero. Cabe destacar que se ha hecho uso del mecanismo de mensajes `uvm_info() que proporciona UVM para poder observar el valor y el instante de tiempo de simulación de

cada elemento o carácter que se ha recibido en la función `write()`. Estos mensajes se mostrarán junto con el cronograma de la simulación para poder observar mejor el mismo.

- Las líneas entre la 56 y la 80 implementan la función `write_pasive_monitor(data_packet t)`. Esta función tiene el mismo comportamiento que la función anterior, con la única diferencia que se encarga de recibir las transacciones de salida o actuales del DUV.

- En las lineas de la 83 a la 85 se implementa la fase `run_phase`. Esta tarea simplemente llama a la tarea `compare_data()`.

- Las líneas entre la 87 y la 126 implementan la tarea `compare_data()`. Esta tarea se ejecuta continuamente un bucle `forever` y se encarga de comparar los valores obtenidos de dos fichero, es decir, se comparan los valores esperados o predichos con los valores de actuales del DUV. Cabe destacar que esta tarea subdivide las comparaciones en dos tareas una para comparar los caracteres del mensaje, `compare_msg()` y la otra para comparar los elemntos de matriz de cifrado `compare_elem()`.

- En las líneas entre la 110 y 126 se implementa la tarea `compare_msg()` y entre la línea 128 y la 145 se implementa la tarea `compare_elem()`. Estas tareas, como se comentó, se encargan de comparar sus respectivos valores. En las dos tareas se ha empleado el mecanismo de mensajes para poder obtener en la simulación, para cada comparación, su valor esperado y su valor actual.

- Las líneas entre la 47 y 158 implementan la fase `report_phase`. Esta fase tiene lugar antes de finalizar la simulación y tiene como objetivo informar del numero de coincidensias o desigualdades del proceso de

comparación. Para ello emplea dos funciones `PASS()` y `ERROR()`, que se llaman en las tareas de comparación para incrementar dos contadores. El primero se incrementa en cada acierto o coincidencia de comparación y el segundo en cada desigualdad de comparación. Estas comparaciones se muestran en el Capítulo 6, correspondiente a los resultados.

- En las líneas entre la 160 y 163 se implementa la función `PASS()` y en las líneas entre la 165 y 168 la función `ERROR()`.

4.4.10. Componente `duv_env`

Como se comentó, el componente `hillcipher_env`, simplemente instanciaba un componente *agent*. En este caso, el componente `duv_env` se corresponde con el entorno global formado por un entorno de entrada `henv_in`, un entorno de salida `henv_out` y el componente *scoreboard*. En esta clase, además de referenciar y conectar los componentes que forman el entorno, se lleva a cabo la configuración de los mismos. Esta configuración, como se comentó en el Capítulo 2, permite que los componentes del entorno se configuren de acuerdo a una topología o caso de prueba necesario. El Código 4.16 muestra la implementación del componente `duv_env`.

```
1   class duv_env extends uvm_env;
2
3       hillcipher_env        henv_in;
4       hillcipher_env        henv_out;
5       hillcipher_scoreboard sb;
6
7
8      `uvm_component_utils(dut_env)
9
10     function new(string name, uvm_component parent);
11        super.new(name, parent);
12     endfunction
13
14     function void build_phase(uvm_phase phase);
15        super.build_phase(phase);
16
17        uvm_config_db#(uvm_active_passive_enum)::set(this,
18        "henv_in.agent", "is_agent_active", UVM_ACTIVE);
```

```
19        uvm_config_db#(uvm_active_passive_enum)::set(this,
20        "henv_out.agent", "is_agent_active", UVM_PASSIVE);
21        uvm_config_db#(string)::set(this, "henv_in.agent.monitor",
22        "monitor_intf", "in_intf");
23        uvm_config_db#(string)::set(this, "henv_out.agent.monitor",
24        "monitor_intf", "out_intf");
25        uvm_config_db#(uvm_active_passive_enum)::set(this,
26        "henv_in.agent.monitor", "is_monitor_active", UVM_ACTIVE);
27        uvm_config_db#(uvm_active_passive_enum)::set(this,
28        "henv_out.agent.monitor", "is_monitor_active", UVM_PASSIVE);
29
30        henv_in = hillcipher_env::type_id::create("henv_in", this);
31        henv_out = hillcipher_env::type_id::create("henv_out", this);
32        sb = hillcipher_scoreboard::type_id::create("sb", this);
33
34      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
35    endfunction: build_phase
36
37    function void connect_phase(uvm_phase phase);
38     henv_in.agent.monitor.item_collected_port.connect(sb.exp_in);
39     henv_out.agent.monitor.item_collected_port.connect(sb.exp_out);
40    `uvm_info(get_full_name( ), "Connect phase complete.", UVM_LOW)
41   endfunction: connect_phase
42
43  endclass: duv_env
```

Código 4.16: Implementación del componente `duv_env`.

Del Código 4.16 se destacan los siguientes puntos de interés:

- Las líneas 3, 4 y 5 declaran los componentes que forman parte del entorno.

- En las líneas entre la 14 y la 35 se implementa la fase `built_phase`. En esta fase, en primer lugar, líneas entre la 17 y 28, se emplea el mecanismo de configuración que mediante el método `::set()`, comentado en el Capítulo 2, se inserta en la base de datos los valores que configuran a los componentes que forman parte del entorno. Además, seguidamente y en las lineas 30, 31 y 32, se crean o referencian dichos componentes, configurados anteriormente.

- Las líneas entre la 37 y la 42 implementan la fase `conect_phase`. En esta fase se conectan los componentes del entorno. En este caso, se conecta empleando la función `connect()` el componente `henv_in`,

que como se comentó está formado por el componente *agent* activo, con el puerto del componente *scoreboard* asociado con el componente *predictor* `sb.exp_in`, línea 38. Del mismo modo se conecta el componente `henv_out`, formado por el componente *agent* pasivo, con el puerto del componente *scoreboard* asociado con el componente *comparator* `sb.exp_out`, línea 39.

4.4.11. Objeto de configuración `hillcipher_sequence_config`

Como se comentó en el apartado de la secuencia `hillcipher_sequence`, la tarea `body()` daba valores a las diferentes transacciones para formar una secuencia empleando un objeto de configuración del cual obtenía, por un lado, el fichero con los valores de las transacciones y, por otro lado, las longitudes de las mismas. El Código 4.17 muestra el objeto de configuración `hillcipher_sequence_confing`.

```
1  class hillcipher_sequence_config extends uvm_object;
2       string file_name = "file.txt";
3      `uvm_object_utils(hillcipher_sequence_config)
4
5     function new( string name = "hillcipher_sequence_config" );
6        super.new( name );
7     endfunction: new
8
9  endclass: hillcipher_sequence_config
```

Código 4.17: Implementación del objeto de configuración `hillcipher_sequence_config`.

Del Código 4.17 solo cabe destacar la línea 2, en la que se declara una variable de tipo `string` a la que se le asocia el nombre del fichero de configuración empleado para asignar los valores a las transacciones que forman una secuencia.

4.4.12. La librería `test_library`

El Código 4.18 muestra la librería de *test* empleada en el entorno generado. Esta librería está compuesta de un clase `base_test` empleada para modelar aspectos comunes a cualquier test. Esta clase `base_test` es exactamente la misma que se describió en el Capítulo 2, pues como se comentó, sirve de base para cualquier *test*. Como ejemplo se añadió un caso de *test* en la clase `hillcipher_test`, esta clase configura los valores del objeto de configuración anterior para generar un determinado caso de *test* correspondiente a los valores asignados en el fichero comentado anteriormente.

```
1   class base_test extends uvm_test;
2      `uvm_component_utils(base_test)
3
4      dut_env env;
5      uvm_table_printer printer;
6
7      function new(string name, uvm_component parent);
8         super.new(name, parent);
9      endfunction: new
10
11     function void build_phase(uvm_phase phase);
12        super.build_phase(phase);
13        env = dut_env::type_id::create("env", this);
14        printer = new( );
15        printer.knobs.depth = 5;
16     endfunction:build_phase
17
18     virtual function void end_of_elaboration_phase(uvm_phase
19     phase);
20        `uvm_info(get_type_name( ), $sformatf("Printing the test
21        topology :\n%s", this.sprint(printer)), UVM_LOW)
22     endfunction: end_of_elaboration_phase
23
24     virtual task run_phase(uvm_phase phase);
25        phase.phase_done.set_drain_time(this, 1500);
26     endtask: run_phase
27
28  endclass: base_test
29
30  class hillcipher_test extends base_test;
31
32    hillcipher_sequence_config seq_config;
33
34     `uvm_component_utils(hillcipher_test)
35
36     function new(string name, uvm_component parent);
37        super.new(name, parent);
38     endfunction: new
39
```

```
40      function void build_phase(uvm_phase phase);
41         super.build_phase(phase);
42         seq_config  = hillcipher_sequence_config ::type_id::create(
43         "hillcipher_sequence_config");
44         uvm_config_db#(hillcipher_sequence_config)::set(.cntxt
45         (null),.inst_name( "*" ), .field_name( "seq_config "),
46         .value( seq_config ) );
47      endfunction: build_phase
48
49      virtual task run_phase(uvm_phase phase);
50         hillcipher_sequence seq;
51         super.run_phase(phase);
52         phase.raise_objection(this);
53         seq = hillcipher_sequence::type_id::create("seq");
54         seq.start(env.henv_in.agent.sequencer);
55         phase.drop_objection(this);
56      endtask: run_phase
57   endclass: hillcipher_test
```

Código 4.18: Implementación de la librería `test_library`.

Del Código 4.18 perteneciente a la clase `hillcipher_test` se destacan los siguientes puntos:

- En la línea 32 se declara el objeto de configuración que configura mediante un fichero los valores de las trasacciones que forman la secuencia de *Hill Cipher.*

- Las líneas entre la 40 y 47 implementan la fase `built_phase`. En esta fase se crea una referencia al objeto de configuración y empleando el metodo `::set()` del mecanismo de acceso a la base de datos, se inserta este objeto en la base de datos, con el propósito de configurar la secuencia, como se comentó, para el caso de *test* correspondiente.

- En las líneas entre la 49 y 59 se implementa la fase `run_phase`. En esta fase se implementa el comportamiento del *test*. En esta fase se puede observar el comportamiento dinámico de los objetos de secuencia pues, a diferencia de los componentes, estos se pueden crear en la fase `run_phase`.

4.4.13. El módulo *top*

En el Código 4.19 se muestra el módulo de mayor jerarquía, el módulo `top`, este módulo simplemente incluye los `package` y macros necesarias para la simulación, referencia las interfaces, el DUV y gestiona la lógica de control para la generación del reset y de la señal de reloj. Además, como se comentó, mediante el mecanismo de configuración permite configurar la interfaz virtual.

```
1   `include "uvm_macros.svh"
2   `include "hillcipher_pkg.sv"
3   `include "hillcipher.v"
4   `include "hillcipher_if.sv"
5
6   module top;
7      import uvm_pkg::*;
8      import hillcipher_pkg::*;
9      bit clk;
10     bit rst;
11
12     hillcipher_if ivif(.clk(clk), .rst(rst));
13     hillcipher_if ovif(.clk(clk), .rst(rst));
14
15     hillcipher hillcipher_top(.clk(clk),
16                    .rst(rst),
17                    .adata(ivif.elem_in),
18                    .adata_en(ivif.elem_in_en),
19                    .msg(ivif.msg_in),
20                    .msg_en(ivif.msg_in_en),
21                    .msg_or_adata(ivif.msg_or_elem),
22                    .odata(ovif.elem_out),
23                    .odata_en(ovif.elem_out_en),
24                    .omsg(ovif.msg_out),
25                    .omsg_en(ovif.msg_out_en)
26                   );
27
28     always #5 clk = ~clk;
29
30     initial begin
31        #5 rst = 1'b1;
32        #25 rst = 1'b0;
33     end
34
35     initial begin
36       uvm_config_db#(virtual hillcipher_if)::set(uvm_root::get( ) ,
37       "*.agent.*" , "in_intf", ivif);
38       uvm_config_db#(virtual hillcipher_if)::set(uvm_root::get( ) ,
39       "*.monitor" , "out_intf", ovif);
40
41       run_test();
42
43     end
44  endmodule
```

Código 4.19: Implementación del modulo *top.*

En el Código 4.19 correspondiente al módulo `top` se destacan los siguientes puntos:

- En las líneas 12 y 13 se declaran las dos interfaces empleadas para interactuar con el DUV, la interfaz de entrada y la interfaz de salida.

- En las líneas entre la 15 y 16 se declara el DUV y se asocia sus señales, de entrada o salida, con la interfaz de entrada o salida según corresponda.

- La línea 28 genera una señal de reloj continua con un período de 5 unidades de tiempo. Esta señal sirve de sincronismo en la etapa de simulación.

- Las líneas entre la 23 y la 33 se emplan para generar la señal de reset. Esta señal, activa a nivel alto, inicializa las señales del DUV.

- Finalmente, entre las líneas 35 y 43, en un proceso secuencial, se emplea el mecanismo de configuracion de UVM para insertar las interfaces en la base de datos y permitir, de esta manera, que el componente *monitor* y el componente *driver* puedan tener acceso a la misma a través de la interfaz virtual, como se comentó en el Capítulo 2.

Capítulo 5 Resultados

5.1. Introducción

En este capítulo se mostrarán los resultados obtenidos mediante el uso de la la herramienta de simulación *QuestaSim de Mentor Graphics*, ejecutando un test de prueba asociado al entorno desarrollado. Los resultados obtenidos se presentarán de dos formas:

- En modo texto, mediante ventana de comandos, en el que se podrán observar los diferentes resultados obtenidos, por una parte, desde el *testbench* y haciendo uso del sistema de mensajería de UVM y, por otra parte, desde las salidas del modelo de referencia descrito en C y mostradas por pantalla empleando la función `printf()`.

- En modo cronograma, en el que se podrá ver el aspecto que tiene el fichero de registro de señales `wave.vcd` perteneciente al entorno diseñado.

5.2. La interfaz DPI con la herramienta *QuestaSim*

En la herramienta *QuestaSim* de *Mentor Graphics* se implementa la interfaz *SystemVerilog* DPI conforme al estándar IEEE Std 1800-2005. Esta interfaz no requiere del registro de las aplicaciones DPI, sin embargo, cada tarea/función importada o exportada por DPI debe identificarse utilizando la sintaxis propia para la importación y/o exportación comentadas en el Capítulo 3.

El flujo predeterminado está asociado a proporcionar archivos descritos en *SystemVerilog* y C en la línea de comandos, empleando el comando `vlog`. El compilador `vlog` de la herramienta compilará automáticamente los archivos

SystemVerilog y los archivos en lenguaje C especificados, preparándolos para cargarlos en la simulación. A continuacion se muestra un ejemplos de estos comandos:

```
vlog top.sv hillcipher.c

vsim top -do <do_file>
```

Opcionalmente, los archivos DPI C se pueden compilar externamente e integrarlos en una biblioteca compartida. Por ejemplo, los modelos IP de terceros pueden distribuirse de esta manera. La biblioteca compartida puede cargarse en el simulador con la opción de línea de comandos:

```
-sv_lib <lib> o -sv_liblist <bootstrap_file>
```

La opción `-sv_lib` especifica el nombre de la biblioteca compartida, sin extensión. La herramienta agrega una extensión de archivo, según corresponda.

5.2.1. Flujo de uso DPI en la herramienta *QuestaSim*

El uso correcto de la interfaz DPI en la herramienta *QuestaSim* de *Mentor Graphics* se corresponde con el flujo representado en la Figura 5.1 [23].

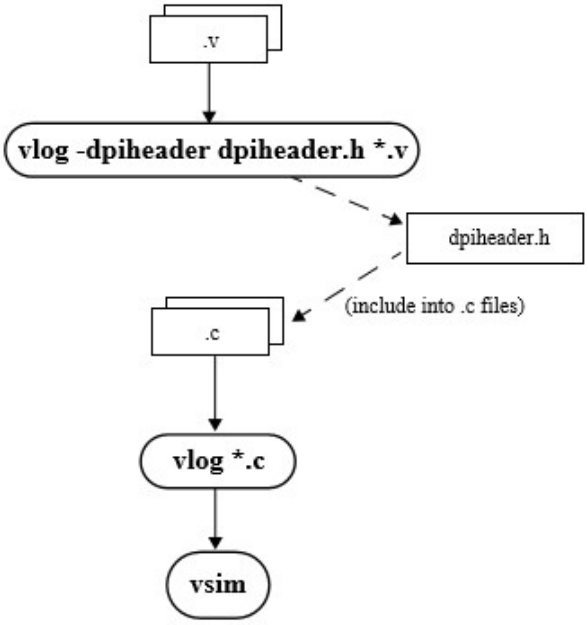

Figura 5.1: Flujo de uso DPI en la herramienta *QuestaSim* de *Mentor Graphics*.

El uso de esta interfaz parte de la creación de la librería `vlib work` y considerando que ya se dispone en ella de todos los componentes descritos en *SystemVerilog* que conforman el entorno desarrollado, así como el código del modelo de referencia descrito en C. Con estas consideraciones, los pasos a seguir son:

1. Ejecutar el comando `vlog` para generar el archivo `dpiheader.h`.

- Este archivo define la interfaz entre el entorno en lenguaje C y *Systemverilog* en el entorno de simulación *QuestaSim* para las tareas/funciones exportadas e importadas. Aunque el fichero `dpiheader.h` es un archivo de conveniencia del usuario en lugar de un requisito, incluirlo en el código C puede resolver inmediatamente los problemas causados por una interfaz definida incorrectamente. El comando empleado para crear el archivo de encabezado es el siguiente:

```
vlog -dpiheader dpiheader.h files.v
```

2. Incluir el archivo `dpiheader.h` en el código C.

- La herramienta *QuestaSim* recomienda que cualquier código DPI C de usuario que acceda a tareas/funciones exportadas o defina tareas/ funciones importadas, debe incluir el archivo `dpiheader.h`. Esto permite al compilador C verificar la interfaz entre C y *SystemVerilog* en el entorno *QuestaSim*.

3. Compilar el entorno completo empleando el comando `vlog *`:

- La alternativa consiste en especificar los archivos descritos en C en la línea de comandos del módulo `vlog`. En este caso, el comando invocará el compilador C pasándole el correspondiente archivo. Por ejemplo, puede emplearse el siguiente comando:

  ```
  vlog SystemVerilog1.sv SystemVerilog2.sv dpicode.c
  ```

  Este comando compila todos los archivos *SystemVerilog* y archivos C en la biblioteca de trabajo.

4. Simular el diseño:

- El comando `vsim` del entorno *QuestaSim* carga automáticamente el código descrito en C el cual ha sido compilado en el tiempo de elaboración. A continuación se muestra un ejemplo de utilización de este comando, donde se realizaría la llamada al simulador *QuestaSim*.

  ```
  vsim top
  ```

Cabe resaltar que el comando `qverilog` también acepta archivos C en la línea de comandos. Este commando funciona de forma similar a `vlog`, pero invoca automáticamente al módulo `vsim` al final de la fase de compilación.

5.3. Resultados de la simulación

A continuación se mostrará la simulación asociada al entorno desarrollado en este TFG. En la Figura 5.2 se pueden observar la totalidad de los archivos asociados al entorno desarrollado, los cuales se han empleado para la simulación del entorno.

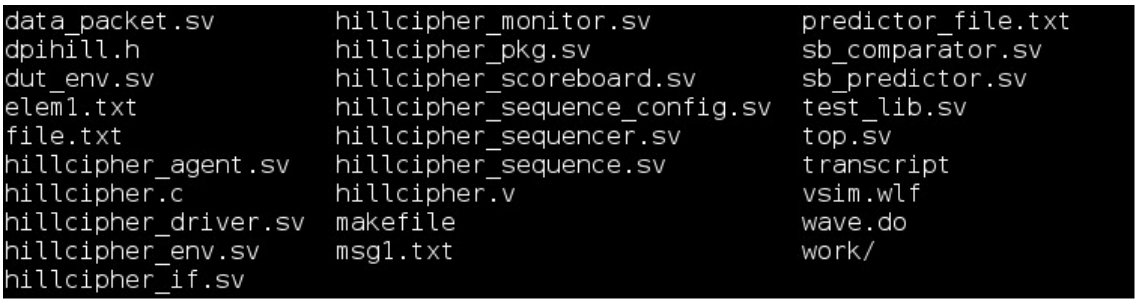

Figura 5.2: Archivos del entorno desarrollado empleados en la simulación.

5.3.1. Resultados de la simulación por ventana de comandos

En este apartado se presentarán los resultados obtenidos tras la simulación de un test. En este test se han realizado dos pruebas empleando para cada una, una secuencia distinta. Los resultados obtenidos mediante la ventana de comandos se adjunta por completo en el Anexo II, este archivo es bastante extenso y se ha obtado por mostrarlo en diferentes códigos para su explicación.

El Código 5.1 muestra la salida por la ventana de comandos, mostrando los mensajes UVM_INFO que notifican el avance de la simulación, líneas 3 a 45. Seguidamente se muestra la topología del entorno con las dependencias entre clases, es decir, el árbol jerárquico del entorno implementado, líneas entre la 46 y 88.

```
1  # vsim -c top -do "run -all; quit -f"
2  ....
3  # UVM_INFO
4  verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0:
5  reporter [Questa UVM] QUESTA_UVM-1.2.2
6  # UVM_INFO
7  verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0:
```

```
reporter [Questa UVM]  questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test hillcipher_test...
# UVM_INFO dut_env.sv(32) @ 0: uvm_test_top.env [uvm_test_top.env]
Build stage complete.
# UVM_INFO hillcipher_env.sv(14) @ 0: uvm_test_top.env.henv_in
[uvm_test_top.env.henv_in] Build stage complete.
# UVM_INFO hillcipher_agent.sv(28) @ 0:
uvm_test_top.env.henv_in.agent [uvm_test_top.env.henv_in.agent]
Build stage complete.
# UVM_INFO hillcipher_driver.sv(14) @ 0:
uvm_test_top.env.henv_in.agent.driver
[uvm_test_top.env.henv_in.agent.driver] Build stage complete.
# UVM_INFO hillcipher_monitor.sv(27) @ 0:
uvm_test_top.env.henv_in.agent.monitor [hillcipher_monitor]
INTERFACE USED = in_intf
# UVM_INFO hillcipher_monitor.sv(35) @ 0:
uvm_test_top.env.henv_in.agent.monitor
[uvm_test_top.env.henv_in.agent.monitor] Build stage complete.
# UVM_INFO hillcipher_env.sv(14) @ 0: uvm_test_top.env.henv_out
[uvm_test_top.env.henv_out] Build stage complete.
# UVM_INFO hillcipher_agent.sv(28) @ 0:
uvm_test_top.env.henv_out.agent [uvm_test_top.env.henv_out.agent]
Build stage complete.
# UVM_INFO hillcipher_monitor.sv(27) @ 0:
uvm_test_top.env.henv_out.agent.monitor [hillcipher_monitor]
INTERFACE USED = out_intf
# UVM_INFO hillcipher_monitor.sv(35) @ 0:
uvm_test_top.env.henv_out.agent.monitor
[uvm_test_top.env.henv_out.agent.monitor] Build stage complete.
# UVM_INFO hillcipher_agent.sv(34) @ 0:
uvm_test_top.env.henv_in.agent [uvm_test_top.env.henv_in.agent]
Connect stage complete.
# UVM_INFO hillcipher_agent.sv(34) @ 0:
uvm_test_top.env.henv_out.agent [uvm_test_top.env.henv_out.agent]
Connect stage complete.
# UVM_INFO dut_env.sv(38) @ 0: uvm_test_top.env [uvm_test_top.env]
Connect phase complete.
# UVM_INFO test_lib.sv(19) @ 0: uvm_test_top [hillcipher_test]
Printing the test topology :
#
----------------------------------------------------------------
----------------
# Name                          Type
Size  Value
#
----------------------------------------------------------------
----------------
# uvm_test_top                  hillcipher_test    -     @466
#   env                         dut_env            -     @474
#     henv_in                   hillcipher_env     -     @500
#       agent                   hillcipher_agent   -     @524
#         driver                hillcipher_driver  -     @660
#           rsp_port            uvm_analysis_port  -     @677
#           seq_item_port       uvm_seq_item_pull_port-  @668
#         monitor               hillcipher_monitor -     @686
#           item_collected_port uvm_analysis_port  -     @701
#           is_monitor_active   uvm_active_passive_enum  1
UVM_ACTIVE
#         sequencer             hillcipher_sequencer-    @537
#           rsp_export          uvm_analysis_export -    @545
#           seq_item_export     uvm_seq_item_pull_imp -  @651
```

```
69  #           arbitration_queue    array           0     -
70  #           lock_queue           array           0     -
71  #           num_last_reqs        integral          32    'd1
72  #           num_last_rsps        integral          32    'd1
73  #         is_agent_active        uvm_active_passive_enum   1
74  UVM_ACTIVE
75  #     henv_out                   hillcipher_env        -  @508
76  #       agent                    hillcipher_agent      -  @723
77  #         monitor                hillcipher_monitor    -  @736
78  #           item_collected_port  uvm_analysis_port     -  @750
79  #           is_monitor_active    uvm_active_passive_enum  1
80  UVM_PASSIVE
81  #         is_agent_active        uvm_active_passive_enum  1
82  UVM_PASSIVE
83  #     sb                         hillcipher_scoreboard -  @516
84  #       cmp                      sb_comparator         -  @802
85  #         aport_active_monitor   uvm_analysis_imp_active_monitor-
86  @810
87  #         aport_pasive_monitor   uvm_analysis_imp_pasive_monitor
88  -     @819
89  #       exp_in                   uvm_analysis_export   -     @767
90  #       exp_out                  uvm_analysis_export   -     @776
91  #       prd                      sb_predictor          -     @785
92  #         analysis_imp           uvm_analysis_imp      -     @793
93  #         results_ap             uvm_analysis_port     -     @844
94  #
```

Código 5.1: Resultado de la simulación en dominio del codigo *SystemVerilog* del entorno UVM.

Las líneas 50 a 94 muestran la composición completa del entorno UVM generado, consistente en un componente *top*, línea 55, que, a su vez contiene un componente *env* (*enviroment*), línea 56. Este último a su vez contiene dos componentes *enviroment,* uno de entrada, *henv_in*, línea 57, asociado al interfaz de entrada, y un segundo componente, denominado *henv_out*, línea 75, asociado a la interfaz de salida. El listado muestra que el componente *henv_in* contiene un componente *agent* activo, línea 58, con sus componentes, líneas 59, 62 y 66 respectivamente, *driver*, *monitor* y *sequencer*. Mientras que el *henv_out*, línea 75, simplemente contiene un componente *monitor,* línea 77. Además se puede ver la arquitectura del componente *scoreboard*, `sb`, línea 83, con sus componentes *comparator,* línea 84, y *predictor,* línea 91.

Una vez se han recibido todos los datos en el dominio del código *SystemVerilog*, se llama al modelo de referencia descrito en C para realizar la predicción, empleando la función importada definida en el Capítulo 4. El Código 5.2 muestra la salida por la ventana de comandos del modelo descrito en C que muestra los resultados de la predicción realizada por dicho modelo.

```
1     #           Golden reference prediction
2     #
3     #
4     # Enter 3x3 matrix for key (It should be inversible):
5     # 6.000000     24.000000 1.000000
6     # 13.000000   16.000000  10.000000
7     # 20.000000   17.000000  15.000000
8     #
9     # Enter a 3 letter string: act
10    # Encrypted string is: poh
11    #
12    # Inverse Matrix is:
13    # 0.158730 -0.777778 0.507937
14    # 0.011338 0.158730 -0.106576
15    # -0.224490 0.857143 -0.489796
16    #
17    # Decrypted string is: act
```

Código 5.2: Resultado de la simulación para la predicción de la primera secuencia en el modelo de referencia descrito en C.

Las líneas 5 a 7 muestran los datos de la matriz utilizada para el ejemplo, mientras que la línea 9 muestra la clave alfanumérica utilizada, en este caso *act*. Además, la línea 10 muestra el resultado de encriptar esta clave y las líneas 13 a 15 muestran los valores de la matriz inversa correspondiente a la matriz de entrada. Por último, y a modo de comprobación, la línea 17 muestra la clave tras el proceso de desencriptación.

Como se comentó, la función importada retorna los valores al código descrito en *SystemVerilog* para realizar las comparaciones. El Código 5.3 muestra las comparaciones llevadas acabo en el componente *comparator* de la arquitectura *scoreboard*.

```
1     # UVM_INFO sb_comparator.sv(80) @ 175: uvm_test_top.env.sb.cmp
2     [sb_comparator run task] WAITING for expected output
3     # UVM_INFO sb_comparator.sv(84) @ 175: uvm_test_top.env.sb.cmp
4     [sb_comparator run task] WAITING for actual output
5     # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
6     [PASS_msg ] Expected=a   Actual=a
7     #
8     # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
9     [PASS_msg ] Expected=c   Actual=c
10    #
11    # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
12    [PASS_msg ] Expected=t   Actual=t
13    #
14    # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
15    [PASS_elem ] Expected=6.000000   Actual=6.000000
```

```
16  #
17  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
18  [PASS_elem ] Expected=24.000000   Actual=24.000000
19  #
20  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
21  [PASS_elem ] Expected=1.000000   Actual=1.000000
22  #
23  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
24  [PASS_elem ] Expected=13.000000   Actual=13.000000
25  #
26  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
27  [PASS_elem ] Expected=16.000000   Actual=16.000000
28  #
29  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
30  [PASS_elem ] Expected=10.000000   Actual=10.000000
31  #
32  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
33  [PASS_elem ] Expected=20.000000   Actual=20.000000
34  #
35  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
36  [PASS_elem ] Expected=17.000000   Actual=17.000000
37  #
38  # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
39  [PASS_elem ] Expected=15.000000   Actual=15.000000
```

Código 5.3: Comparaciones llevadas acabo en el componente *comparator* de la arquitectura *scoreboard* para la primera secuencia.

El Código 5.3 muestra la lectura correcta de los datos obtenidos desde la entrada y su posterior envío al DUV. Las líneas 6, 9 y 12 reflejan los caracteres correspondientes a la clave, que se corresponden con la clave de entrada, mientras que a partir de la línea 14 se muestran los diferentes valores de la matriz de entrada. El segundo campo de los mensajes `UVM_INFO` indica que estos valores se han obtenido en el componente `sb_comparator`.

Seguidamente, se comienza a recibir la segunda secuencia y el procedimiento es el mismo que el mostrado anteriormente, pero con diferentes valores. El Código 5.4 muestra las predicciones del modelo de referencia para la segunda secuencia.

```
1   #           Golden reference prediction
2   #
3   #
4   # Enter 3x3 matrix for key (It should be inversible):
5   # 1.000000 2.000000   3.000000
6   # 13.000000   16.000000  10.000000
7   # 20.000000   17.000000  15.000000
8   #
9   # Enter a 3 letter string: pvp
```

```
10  # Encrypted string is: yfy
11  #
12  # Inverse Matrix is:
13  # -0.322581 -0.096774 0.129032
14  # -0.023041 0.207373 -0.133641
15  # 0.456221 -0.105991 0.046083
16  #
17  # Decrypted string is: pvp
```

Código 5.4: Resultado de la simulación para la predicción de la segunda secuencia en el modelo de referencia descrito en C.

Al igual que se comentó para el primer ejemplo, las líneas 5 a 7 muestran los datos de la matriz utilizada para el ejemplo, mientras que la línea 9 muestra la clave alfanumérica utilizada, en este caso *pvp*. Además, la línea 10 muestra el resultado de encriptar esta clave y las líneas 13 a 15 muestran los valores de la matriz inversa correspondiente a la matriz de entrada. Por último, y a modo de comprobación, la línea 17 muestra la clave tras el proceso de desencriptación.

El Código 5.5 muestra las comparaciones para la segunda secuencia.

```
1   # UVM_INFO sb_comparator.sv(80) @ 305: uvm_test_top.env.sb.cmp
2   [sb_comparator run task] WAITING for expected output
3   # UVM_INFO sb_comparator.sv(84) @ 305: uvm_test_top.env.sb.cmp
4   [sb_comparator run task] WAITING for actual output
5   # UVM_INFO sb_comparator.sv(103) @ 305: uvm_test_top.env.sb.cmp
6   [PASS_msg ] Expected=p   Actual=p
7   #
8   # UVM_INFO sb_comparator.sv(103) @ 305: uvm_test_top.env.sb.cmp
9   [PASS_msg ] Expected=v   Actual=v
10  #
11  # UVM_INFO sb_comparator.sv(103) @ 305: uvm_test_top.env.sb.cmp
12  [PASS_msg ] Expected=p   Actual=p
13  #
14  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
15  [PASS_elem ] Expected=1.000000   Actual=1.000000
16  #
17  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
18  [PASS_elem ] Expected=2.000000   Actual=2.000000
19  #
20  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
21  [PASS_elem ] Expected=3.000000   Actual=3.000000
22  #
23  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
24  [PASS_elem ] Expected=13.000000   Actual=13.000000
25  #
26  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
27  [PASS_elem ] Expected=16.000000   Actual=16.000000
28  #
29  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
```

```
30  [PASS_elem ] Expected=10.000000   Actual=10.000000
31  #
32  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
33  [PASS_elem ] Expected=20.000000   Actual=20.000000
34  #
35  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
36  [PASS_elem ] Expected=17.000000   Actual=17.000000
37  #
38  # UVM_INFO sb_comparator.sv(120) @ 305: uvm_test_top.env.sb.cmp
39  [PASS_elem ] Expected=15.000000   Actual=15.000000
```

Código 5.5: Comparaciones llevadas acabo en el componente *comparator* de la arquitectura *scoreboard* para la segunda secuencia.

El Código 5.5 muestra la lectura correcta de los datos obtenidos desde la entrada y su posterior envío al DUV. Las líneas 6, 9 y 12 reflejan los caracteres correspondientes a la clave, que se corresponden con la clave de entrada, mientras que a partir de la línea 14 se muestran los diferentes valores de la matriz de entrada. El segundo campo de los mensajes `UVM_INFO` indica que estos valores se han obtenido en el componente `sb_comparator`.

Finalmente, el Código 5.6 muestra los resultados finales de la simulación, obteniendo el numero de transacciones que sen han recolectado e informando del numero de coincidencias o desigualdades del proceso de comparacion. Ademas, se informa del número de mensajes mostrados por cada componente del entorno y de los mensajes *errors* y *warnings*.

```
1   # UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
2   1795: reporter [TEST_DONE] 'run' phase is ready to proceed to the
3   'extract' phase
4   #      UVM_INFO      hillcipher_monitor.sv(130)      @      1795:
5   uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
6   REPORT: COLLECTED PACKETS = 26
7   #      UVM_INFO      hillcipher_monitor.sv(130)      @      1795:
8   uvm_test_top.env.henv_out.agent.monitor      [hillcipher_monitor]
9   REPORT: COLLECTED PACKETS = 24
10  # UVM_INFO sb_comparator.sv(134) @ 1795: uvm_test_top.env.sb.cmp
11  [sb_comparator]
12  #
13  #
14  # *** TEST PASSED - 24 vectors ran, 24 vectors passed ***
15  #
16  #
17  # --- UVM Report Summary ---
18  #
19  # ** Report counts by severity
20  # UVM_INFO :   77
21  # UVM_WARNING :    0
```

```
22  # UVM_ERROR :    0
23  # UVM_FATAL :    0
24  # ** Report counts by id
25  # [PASS_elem ]    18
26  # [PASS_msg ]     6
27  # [Questa UVM]     2
28  # [RNTST]     1
29  # [TEST_DONE]     1
30  # [hillcipher_driver]     2
31  # [hillcipher_monitor]     6
32  # [hillcipher_test]     1
33  # [sb_comparator]     1
34  # [sb_comparator run task]     4
35  # [sb_predictor]    24
36  # [uvm_test_top.env]     2
37  # [uvm_test_top.env.henv_in]     1
38  # [uvm_test_top.env.henv_in.agent]     2
39  # [uvm_test_top.env.henv_in.agent.driver]     1
40  # [uvm_test_top.env.henv_in.agent.monitor]     1
41  # [uvm_test_top.env.henv_out]     1
42  # [uvm_test_top.env.henv_out.agent]     2
43  # [uvm_test_top.env.henv_out.agent.monitor]     1
44  #       **       Note:       $finish                              :
45  /home/soft/eucad/mentor/2015/questa_sv_af_10.4b_Linux/questasim/l
46  inux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
47  #    Time: 1795 ns  Iteration: 53  Instance: /top
48  # Errors: 0, Warnings: 0
```

Código 5.6: Resultado final de la simulación.

El código 5.6 muestra que se está ejecutando la fase de `extract_phase`, línea 2, correspondiente a la fase en la que el sistema analiza los resultados e imprime las estadísticas. Dentro de la información que se muestra, la línea 6 indica que el número de paquetes procesados es un total de 26. De estos 26, 12 corresponden al primer test (9 paquetes para enviar la matriz y 3 paquetes para enviar la clave) y los otros 12 al segundo test. Los dos paquetes restantes corresponden a las secuencias de reset que, al igual que los datos, en UVM se transmite como un paquete más. Las líneas 21, 22 y 23 indican que no existe ningún mensaje de *warning*, *error* o *fatal*, lo que hubiese supuesto realizar un análisis con profundidad de su causa.

5.4.2. Resultados de la simulación mediante cronogramas

En este apartado se muestra la representación del fichero `wave.vcd`. Se representa, de forma grafica y sobre una escala temporal, los diferentes

valores que toman las señales descritas en el fichero. Dicha representación se muestra en las Figuras 5.3, 5.4, 5.5 y 5.6. Cabe destacar que, como se comentó, el DUV implementado como una caja negra recibe señales de la interfaz de entrada y en el ciclo posterior de la señal de reloj los pone en la interfaz de salida.

En primer lugar, la Figura 5.3 muestra el intervalo de tiempo desde que comienza la simulación, instante 0 ns, hasta el instante 85 ns. En esta representación se comienza inicializando las señales activando la señal de reset y seguidamente comienza la primera secuencia de transacciones, enviándose en primer lugar los caracteres correspondientes al mensaje a cifrar, habilitados mediante sus respectivas señales `enable(msg_in_en` y `msg_or_elem)`. Esta transferencia se muestra en el recuadro amarillo. Durante este intervalo se produce el envío de una secuencia compuesta de tres transacciones. Cada transacción contiene el valor correspondiente a un carácter de la clave, en este caso los valores son `8'aa`, `8'ac` y `8'at`. Además, el protocolo de comunicación se activa correctamente ya que el sistema valida cada uno de los datos activando la señal `msg_in_en`.

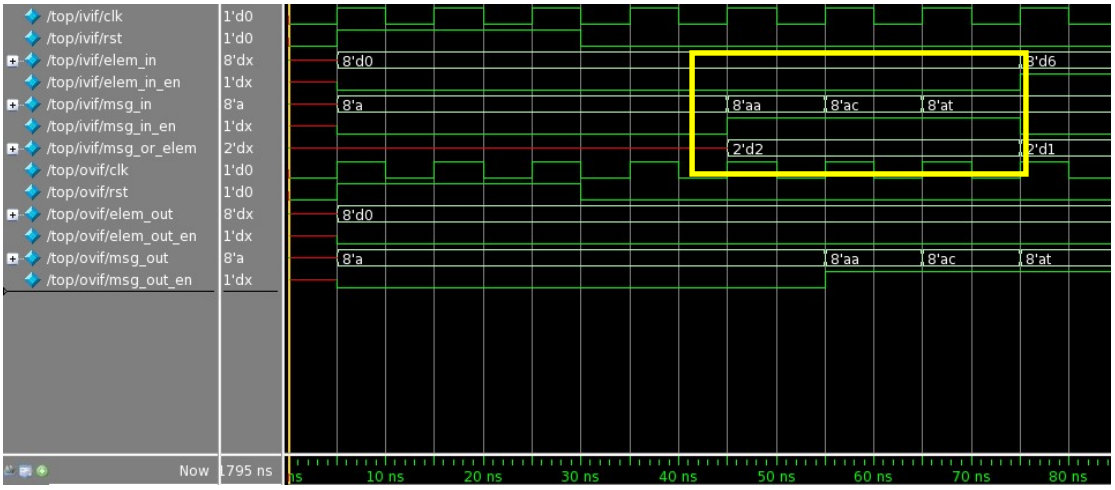

Figura 5.3: Cronograma del resultado obtenido de la simulación, correspondiente a la primera secuencia de transacciones.

En la Figura 5.4 se muestra la continuación de la simulación, hasta el instante de tiempo 180 ns, correspondiente al envío de los elementos de la matriz empleada para realizar el cifrado (recuadro amarillo). En este intervalo se transmiten 9 transacciones, correspondientes a los elementos de la matriz. Estos elementos se transmiten como una única secuencia, activando la señal `elem_in_en`. El análisis de los datos muestra que se corresponden a los valores de los elementos de la matriz.

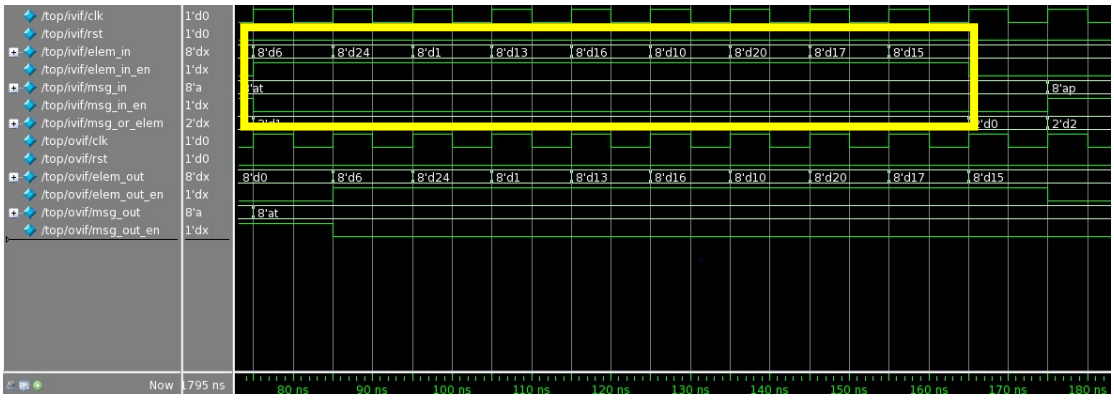

Figura 5.4: Cronograma del resultado obtenido de la simulación, correspondiente a la primera secuencia de transacciones.

Para mostrar mejor los resultados, así como los instantes del tiempo de simulación para la recepción de cada uno de los valores, representados anteriormente, se muestra el Código 5.7. Este listado pertenece a una parte de los resultados obtenidos por la ventana de comandos, en concreto a los mensajes mostrados por el componente `sb_predictor` en el cual se ha empleado el mecanismo de mensaje UVM para poder observar los valores y los tiempos de las distintas transacciones que forman la primera secuencia de *Hill Cipher* .

```
1  # UVM_INFO sb_predictor.sv(28) @ 55: uvm_test_top.env.sb.prd
2  [sb_predictor] REPORT: data_Colected_s = 117 REPORT:TIME = 55
3  # UVM_INFO sb_predictor.sv(28) @ 65: uvm_test_top.env.sb.prd
4  [sb_predictor] REPORT: data_Colected_s = 118 REPORT:TIME = 65
5  # UVM_INFO sb_predictor.sv(28) @ 75: uvm_test_top.env.sb.prd
6  [sb_predictor] REPORT: data_Colected_s = 109 REPORT:TIME = 75
7  # UVM_INFO sb_predictor.sv(34) @ 85: uvm_test_top.env.sb.prd
8  [sb_predictor] REPORT: data_Colected_s = 6 REPORT:TIME = 85
9  # UVM_INFO sb_predictor.sv(34) @ 95: uvm_test_top.env.sb.prd
10 [sb_predictor] REPORT: data_Colected_s = 24 REPORT:TIME = 95
```

```
11  # UVM_INFO sb_predictor.sv(34) @ 105: uvm_test_top.env.sb.prd
12  [sb_predictor] REPORT: data_Colected_s = 1 REPORT:TIME = 105
13  # UVM_INFO sb_predictor.sv(34) @ 115: uvm_test_top.env.sb.prd
14  [sb_predictor] REPORT: data_Colected_s = 13 REPORT:TIME = 115
15  # UVM_INFO sb_predictor.sv(34) @ 125: uvm_test_top.env.sb.prd
16  [sb_predictor] REPORT: data_Colected_s = 16 REPORT:TIME = 125
17  # UVM_INFO sb_predictor.sv(34) @ 135: uvm_test_top.env.sb.prd
18  [sb_predictor] REPORT: data_Colected_s = 10 REPORT:TIME = 135
19  # UVM_INFO sb_predictor.sv(34) @ 145: uvm_test_top.env.sb.prd
20  [sb_predictor] REPORT: data_Colected_s = 20 REPORT:TIME = 145
21  # UVM_INFO sb_predictor.sv(34) @ 155: uvm_test_top.env.sb.prd
22  [sb_predictor] REPORT: data_Colected_s = 17 REPORT:TIME = 155
23  # UVM_INFO sb_predictor.sv(34) @ 165: uvm_test_top.env.sb.prd
24  [sb_predictor] REPORT: data_Colected_s = 15 REPORT:TIME = 165
25  #      UVM_INFO      hillcipher_monitor.sv(91)      @      165:
26  uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
27  REPORT: DATA IDLE = 0 REPORT:TIME = 165
```

Código 5.7: Salida por la ventana de comandos perteneciente a los mensajes del componente `sb_predictor`.

Como se comentó, en este *test* se generó una segunda secuencia con valores diferentes. En las Figuras 5.5 y 5.6 se muestran, respectivamente, los caracteres y elementos de la matriz de la nueva secuencia, así como sus valores sobre el cronograma, resaltándose estas transferencias en un recuadro amarillo.

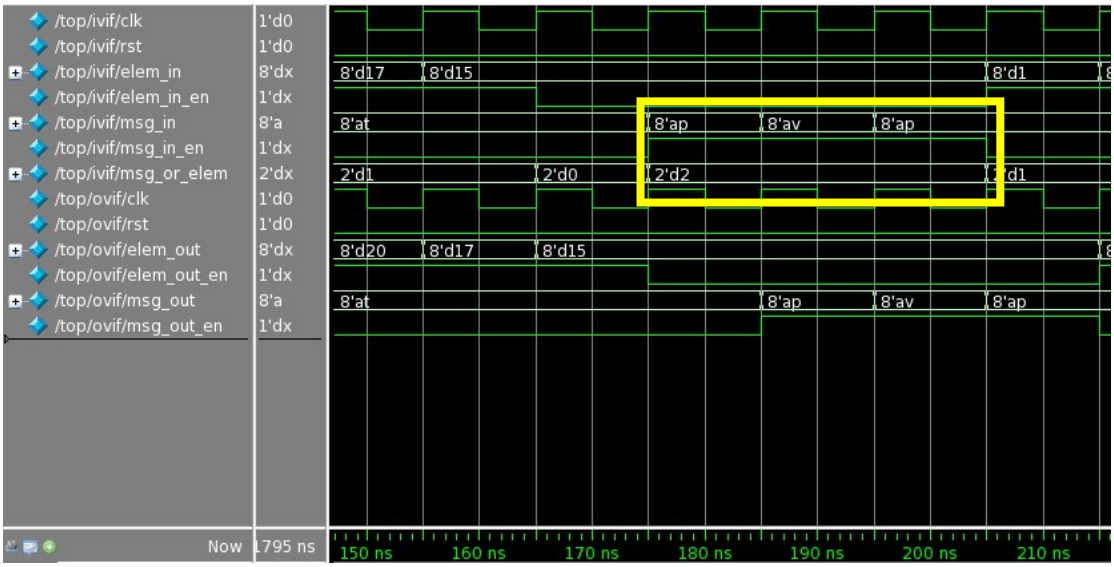

Figura 5.5: Cronograma del resultado obtenido de la simulación, correspondiente a la segunda secuencia de valores.

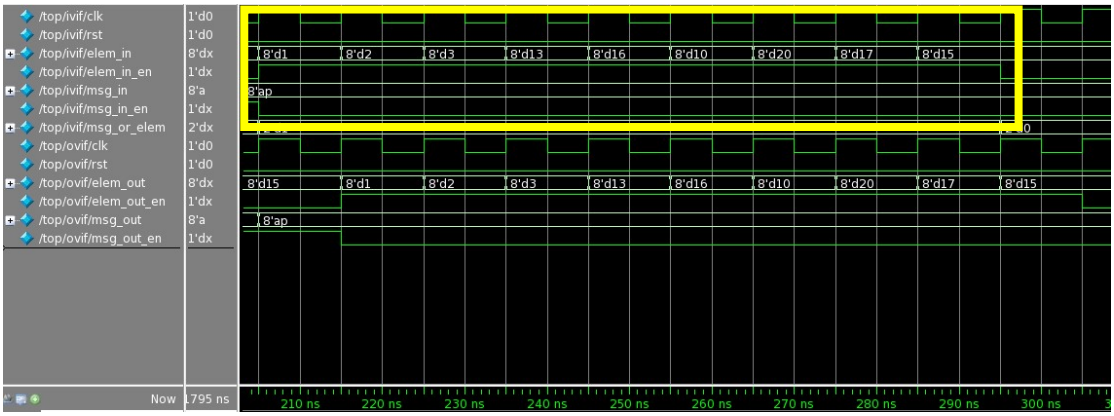

Figura 5.6: Cronograma del resultado obtenido de la simulación, correspondiente a la segunda secuencia de valores.

Seguidamente, el Código 5.8 muestra los mensajes que se obtienen a partir del mecanismo de mensaje UVM con los diferentes valores de las transacciones que conforman la segunda secuencia de *Hill Cipher*.

```
1  # UVM_INFO sb_predictor.sv(28) @ 185: uvm_test_top.env.sb.prd
2  [sb_predictor] REPORT: data_Colected_s = 112 REPORT:TIME = 185
3  # UVM_INFO sb_predictor.sv(28) @ 195: uvm_test_top.env.sb.prd
4  [sb_predictor] REPORT: data_Colected_s = 118 REPORT:TIME = 195
5  # UVM_INFO sb_predictor.sv(28) @ 205: uvm_test_top.env.sb.prd
6  [sb_predictor] REPORT: data_Colected_s = 112 REPORT:TIME = 205
7  # UVM_INFO sb_predictor.sv(34) @ 215: uvm_test_top.env.sb.prd
8  [sb_predictor] REPORT: data_Colected_s = 1 REPORT:TIME = 215
9  # UVM_INFO sb_predictor.sv(34) @ 225: uvm_test_top.env.sb.prd
10 [sb_predictor] REPORT: data_Colected_s = 2 REPORT:TIME = 225
11 # UVM_INFO sb_predictor.sv(34) @ 235: uvm_test_top.env.sb.prd
12 [sb_predictor] REPORT: data_Colected_s = 3 REPORT:TIME = 235
13 # UVM_INFO sb_predictor.sv(34) @ 245: uvm_test_top.env.sb.prd
14 [sb_predictor] REPORT: data_Colected_s = 13 REPORT:TIME = 245
15 # UVM_INFO sb_predictor.sv(34) @ 255: uvm_test_top.env.sb.prd
16 [sb_predictor] REPORT: data_Colected_s = 16 REPORT:TIME = 255
17 # UVM_INFO sb_predictor.sv(34) @ 265: uvm_test_top.env.sb.prd
18 [sb_predictor] REPORT: data_Colected_s = 10 REPORT:TIME = 265
19 # UVM_INFO sb_predictor.sv(34) @ 275: uvm_test_top.env.sb.prd
20 [sb_predictor] REPORT: data_Colected_s = 20 REPORT:TIME = 275
21 # UVM_INFO sb_predictor.sv(34) @ 285: uvm_test_top.env.sb.prd
22 [sb_predictor] REPORT: data_Colected_s = 17 REPORT:TIME = 285
23 # UVM_INFO sb_predictor.sv(34) @ 295: uvm_test_top.env.sb.prd
24 [sb_predictor] REPORT: data_Colected_s = 15 REPORT:TIME = 295
25 # UVM_INFO hillcipher_monitor.sv(91) @ 295:
26 uvm_test_top.env.henv_in.agent.monitor [hillcipher_monitor]
27 REPORT: DATA IDLE = 0 REPORT:TIME = 295
```

Código 5.8: Salida por la ventana de comando de los mensajes con los valores de las transacciones de la segunda secuencia.

En el código anterior se puede comprobar que los datos que llegan al componente `sb_predictor` son los datos correspondientes al segundo test. Así, las líneas 2, 4 y 6, muestran la recepción de la clave usada en este ejemplo (*pvp*, cuyos valores ASCII en decimal son 112, 118 y 112). El resto de los valores correspondientes a la matriz de datos de este segundo test corresponden con los listados entre las líneas 7 y 24.

Capítulo 6 Conclusiones y líneas futuras

En este capítulo se presentan las conclusiones generales obtenidas durante el desarrollo del presente TFG. Además, se propondrán una seríe de líneas futuras asociadas al mismo.

6.1. Conclusiones generales

Dado que en la realización del presente TFG se obordaron una gran cantidad de conceptos, las conclusiones se han agrupado en varias partes, relacionadas con la metodología UVM, los entornos multilenguajes y, finalmente, con la integración de los modelos de referencia.

En primer lugar, se puede decir que la metodología UVM ha supuesto un gran paso para agilizar y simplificar el proceso de la verificacación funcional de un IP o sistema completo. Hay que destacar que adoptar esta metodología supone un gran esfuerzo inicial, pues se deben adquirir diferentes conceptos que implican una curva de apredinzaje con una pendiente inicial muy elevada. También, hay que destacar que el diseñar un primer *testbench* partiendo desde cero implica un alto nivel de comprensión de la metodología, un tiempo de desarrollo elevado y un trabajo adicional, destinado a la puesta a punto del *tesbench*, considerable. Sin embargo, conforme se avance en la adquisición de conocimientos y mejores practicas de esta metodología, como se comentó anterioridad, ésta ofrece un alto grado de reutilización en los componentes que conforman un *tesbench*. Un ejemplo de ello se puede observar en tecnologías que comparten características en común relacionadas con las interfaces, donde el hecho de crear un *tesbench* para llevar a cabo el proceso de verificación funcional, se simplifica enormente, gracias la reutilización de componentes.

El uso de estas metodologías basándose en las características básicas de la misma no es recomendable. Esto es debido a que la gran potencialidad de UVM radica en los conceptos más avanzados de la metodología, tales como el uso de la *Factory* para manejar la sustitución de componentes de forma dinámica, el uso de la capa de modelado de registros, muy útil para añadir métricas de cobertura basadas en los diferentes campos, y condiciones de los mismos, de todos los registros de un IP/Sistema. En caso de no poder abordar todos estos conceptos, se recomienda encarecidamente el uso de una metodología UVM reducida, como es el caso de la metodología UVM *Express*.

En segundo lugar, en cuanto a los entornos multilenguajes, se puede decir que en este campo sí es cierto que hoy en día hay diversas soluciones. Además, son soluciones muy dependientes de los desarrolladores de los entornos y, por lo tanto, no aportan una grado de reutilización destacable. En este punto, se puede decir que todavía hay mucho por hacer. Una de las aportaciones más relevantes en este punto ha sido la publicación de una primera revisión del *Portable Stimulus languaje Specification* (PSS). Esta propuesta de estándar va en la línea de unificar la forma en la que se especifican los estímulos de *test* para un diseño, sin importar el lenguaje que se vaya a usar para su diseño. Esta propuesta ha sido publicada el 14 de junio de 2017 [24].

Finalmente, en cuanto a la integración de modelos de referencia descritos en lenguaje C en entornos basados en la metodología UVM, empleando la interfaz *SystemVerilog* DPI, se puede decir que esta interfaz es una herramienta potente y simple que permite que un *tesbench* o cualquier diseño implementados en *SystemVerilog* puedan llamar a funciones implementadas en lenguaje C y viceversa. Además, en cuanto a la reutilización, como se comentó con anterioridad, la interfaz DPI es muy dependiente de la implementación y no aporta gran reutilización. Para el intercambio de datos simples, es decir los tipos de datos que tienen una equivalencia directa entre los lenguajes *SystemVerilog* y C, la interfaz resulta simple y eficaz. Sin embargo, en algunas estructuras de tipos de datos que no tiene una

equivalencia directa su interactuación suele requerir una elevada codificación adicional, muy dependiente de la solución. No obstante, en el caso de querer obtener sincronismo entre el *tesbench* y el modelo de referencia, puede no resultar eficiente emplear un modelo descrito en lenguaje C, ya que la interfaz DPI no permite exportar funciones/tareas miembros de las clases y, por lo tanto, esta solución no es aplicable. Sin embargo, esta solución es interesante cuando se requiere de modelos sin tiempo que, en la mayor parte de los casos, son empleados en el proceso de verificación funcional. La ventaje está en que en el *testbench* la llamada al modelo de referencia se ejecuta en tiempo cero de simulación, es decir, que no consume tiempo de simulación. Cabe destacar que la llamada a la función importada desde C realiza las operaciones sobre los datos y retorna, a su vez, las predicciones.

6.2. Líneas futuras

A continuación se detallan algunas de las líneas futuras de trabajo más importantes que se proponen tras la realización del presente TFG. Hay que indicar que estas líneas vienen propuestas debido a la limitación en tiempo para el desarrollo de este TFG. Si a esta limitación se añade el hecho de que la curva de aprendizaje de la metodología es muy elevada en su tramo inicial, resulta casi lógico lás siguientes propuestas:

- Profundizar en la utilización de la interfaz DPI, mediante la realización de pruebas más detalladas y con un mayor grado de complejidad, con el propósito de comunicar e integrar componentes descritos en diferentes lenguajes en un mismo *tesbench* (*SystemC*, C++, *etc.*).

- Implementar un entorno con un grado de complejidad mayor, como podría ser un entorno orientado al cifrado de imágenes, dado que las imágenes están formadas por matrices donde cada píxel de la imágen se representa por un elemento de la matriz. Este hecho permitiria el uso de secuencías más complejas y, por lo tanto, el uso de estructuras

de entorno más complejas a intercambiar entre los dominios en *SystemVerilog* y el dominio del lenguaje C.

- Uso de un DUV de mayor complejidad, no sólo a nivel de interfaz, sino de comportamiento, incluyendo latencias variables y protocolos más complejos con dependencia entre sus interfaces.

- Comprobación de la viabilidad de la integración de código descrito en lenguaje C en otros componentes del *tesbench*.

Bibliografía

[1] D. Subhranil, "Delivering functional verification engagements", [En línea] 2015. Synopsys white paper. Disponible en https://pdfs.semanticscholar.org/66f1/dd0814367c393464f5efa0e1a7b854a025d5.pdf. [Consulta: febrero- 2017].

[2] H. Foster, " Understanding and Minimizing Study Bias", Verification Horizons BLOG, 2016. [En línea]. Disponible en: https://blogs.mentor.com/verificationhorizons/blog/2016/08/08/understanding-and-minimizing-study-bias-2016-study/ [Consulta: febrero- 2017].

[3] V. Armas, R. Ojeda, Z. Perdomo, F. Tobajas, "Development of an UVM environment for functional verification of digital systems. XXVII Conference on Design of Circuits and Integrated Systems (DCIS2012). Avignon (France), 2012.

[4] S. Rosenberg, K. A. Meade, (2013) "A Practical Guide to Adopting the Universal Verification Methodology (UVM)". Cadence Design Systems, Inc. ISBN 9780578059556, 9780578059956.

[5] Cadence, "Universal Verification Methodology". [En línea]. Disponible en: https://www.cadence.com/content/cadence-www/global/en_US/home/alliances/standards-and-languages/universal-verification-methodology.html. [Consulta: febrero- 2017].

[6] A. Jain, H. Gupta, S. Jana, K. Kumar. "Early development of UVM based verification environment of image signal processing designs using TLM reference model of RTL". International Journal of Advanced Computer Science and Applications. August 2014. DOI: 10.14569/IJACSA.2014.050212.

[7] S. Konale, N. B. Rao. C-based predictor for scoreboard in universal verification methodology. International Conference on Advances in Engineering & Technology Research (ICAETR - 2014). DOI: 10.1109/ICAETR.2014.7012913.

[8] S. Vasudevan, "Effective Functional Verification". Boston, MA: Springer, 2006. ISBN 9780387326207.

[9] C. Spear, "Systemverilog for verification". Marlborough, MA: Springer, 2006, p. 126. ISBN 9780387270364, 9780387270388.

[10] Verificationguide, "SystemVerilog Tutorial", 2017. [En línea]. Disponible en: http://www.verificationguide.com/p/systemverilog-tutorial.html. [Consulta: febrero - 2017].

[11] Mentor Graphics Corp, "UVM Cookbook", 2014. [En línea]. Disponible en: https://verificationacademy.com/cookbook/uvm. [Consulta: febrero - 2017].

[12] C. Cummings, "The OVM/UVM Factory & Factory Overrides How They Work - Why They Are Important", 1st ed. Sunburst Design, 2012. [En línea]. Disponible en: http://sunburst-design.com/papers/CummingsSNUG2012SV_UVM_Factories.pdf. [Consulta: febrero - 2017].

[13] Acellera. "Universal Verification Methodology (UVM) 1.1 User's Guide", 2011. Disponible en: http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf. [Consulta: febrero - 2017].

[14] V. Cooper, P. Marriott, "Demystifying the UVM Configuration Database", 1st ed. Verilab, 2013. [En línea]. Disponible en: http://www.verilab.com/files/configdb_dvcon2014.pdf. [Consulta: febrero - 2017].

[15] P. Ashwin, M.Vyom, S.Jignesh. "An Overview of Transaction-Level Modeling (TLM) in Universal Verification Methodology (UVM) ". [En línea]. Disponible en: http://www.ejournal.aessangli.in/ASEEJournals/EC105.pdf [Consulta: febrero - 2017].

[16] B. Sniderman, V. Yankelevich. "Multi-Language Verification: Solutions for Real World Problems ". [En línea]. Disponible en: http://events.dvcon.org/2014/proceedings/papers/12_1.pdf [Consulta: febrero - 2017].

[17] Acellera. "SystemVerilog 3.1a Language Reference Manual – Accellera's extension to Verilog". Disponible en: http://www.ece.uah.edu/~gaede/cpe526/SystemVerilog_3.1a.pdf [Consulta: febrero - 2017].

[18] IEEE Standards Association. "IEEE Standard for SystemVerilog-Unified Hardware Design, Specication, and Verication Language," IEEE Std 1800-2009, pp. 901-915 and 1184-1216. [En línea]. Disponible en: http://ieeexplore.ieee.org/document/6469140/. [Consulta: febrero - 2017]. ISBN: 9780738181103

[19] S. Sutherland. " The Verilog PLI Is Dead (maybe) Long Live The SystemVerilog DPI! " [En línea]. Disponible en: http://www.sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf. [Consulta: febrero - 2017].

[20] C. Cummings. "Sunburst Design, Inc. Sunburst Design. OVM/UVM Scoreboard Fundamental Architectures". [En línea]. Disponible en: http://www.sunburst-design.com/papers/CummingsSNUG2013SV_UVM_Scoreboards.pdf. [Consulta: febrero - 2017].

[21] A. Mahapartra, R. Dash. "Data Encryption and Descryption by using Hill Cipher technique and self repetitive matrix". [En línea]. Disponible en:https://pdfs.semanticscholar.org/0580/7cd29b50cfbe160a0f299c79eb7e5e60026c.pdf .[Consulta: marzo - 2017].

[22] R. Doyle. "Hill's Cipher: Linear Algebra in Cryptography". [En línea]. Disponible en: https://rctechnical.files.wordpress.com/2014/09/r_doyle_hill-cipher.pdf. [Consulta: marzo - 2017].

[23] Mentor Graphics Corporation. "Including Supoport for Questa SV/AFV. Software Versión 10.0b" [En línea]. Disponible en: http://rise.cse.iitm.ac.in/people/faculty/kama/prof/questa_sim_user_manual.pdf[Consulta: marzo - 2017].

[24] M. Ballance. "Making Legacy Portable with the Portable Stimulus Specification". [En línea]. Disponible en: https://s3.amazonaws.com/verificationacademy-news/DVCon2017/Papers/dvcon-2017_making-legacy-portable-with-the-portable-stimulus-specification_paper.pdf. [Consulta: julio - 2017].

Parte II

Pliego de condiciones

PC1. Recursos generales

El procedimiento desarrollado y los resultados obtenidos a lo largo del presente TFG están basados en la versión 1.1d de UVM, los cuales son válidos para las versiones de hardware y software que se muestran a continuación.

PC2. Recursos hardware

Los recursos *hardware* utilizados en el desarrollo del TFG se muestran en la Tabla PC1.

Equipo	Modelo
Estación de trabajo	Acer Predator AG3-605

Tabla PC1: Condiciones *hardware*.

PC3. Recursos *software*

Los recursos *software* utilizados en el desarrollo del TFG se presentan en la Tabla PC2.

Concepto	Versión	Fabricante
Entorno de simulación	QuestaSim-64	Mentor Graphics
Escritorio remoto X2Go	10.6	X2Go
Sistema Operativo	Red Hat Linux	Red Hat
Editor de texto gedit	2.16	gnome

Tabla PC2: Condiciones *software*.

La correcta ejecución viene determinada por el valor de las variables de entorno recogidas en la Tabla PC3.

Variable	Significado
UVM_HOME	Ruta de acceso de UVM
UVM_LIB	Ruta de acceso a las librerías de UVM

Tabla PC3: Variables de entorno.

Parte III

Presupuesto

Presupuesto

Para la realización del presente TFG ha sido necesario emplear tanto recursos humanos, como recursos materiales, con el objetivo de lograr el cumplimiento de los objetivos propuesto para este TFG.

A continuación se mostrarán los costes asociados a los recursos humanos y materiales (recursos *hardware* y *software*) que han sido necesarios para la realización de este TFG, destacando que la cuantía del trabajo elaborado se ha fijado según las indicaciones de contrataciones de la Universidad de Las Palmas de Gran Canaria.

PR1. Recursos humanos

Para el cálculo del coste asociado a los recursos humanos, se hará uso de las últimas tablas de honorarios publicadas el 4 de noviembre de 2010 por la Universidad de Las Palmas de Gran Canaria. En base a estas tablas, el coste total de un Técnico superior, Diplomado o Ingeniero Técnico, con una dedicación de 20 horas semanales, asciende a 901,06€. Por lo tanto, se tiene que el coste aproximado asociado a una hora de trabajo será de 11,26€.

Para la realización del presente TFG, el número total de horas invertidas se corresponde con el tiempo asociado a 12 créditos ECTS, es decir, 300 horas. Por lo tanto, el coste asociado a los recursos humanos se corresponde con:

$$Honorarios\ (coste\ RRHH) = 11{,}26€/h \cdot 300h = \textbf{\textit{3.378,00 €}}$$

PR2. Recursos hardware

Para la realización de este TFG se han empleado las herramientas *hardware* que se muestran en la Tabla PR1.

Recurso	Valor de adquisición (€)	Vida útil (meses)	Coste mensual (€)	Tiempo de uso (meses)	Importe (€)
Estación de trabajo Acer Predator AG3-605	800	60	13,33	4	53,2
Impresora modelo – HP Laserjet 4250DTN PCL	300	60	5	1	5
				Total	59,2

Tabla PR1: Costes asociados a los recursos hardware.

PR3. Recursos software

Para la realización de este TFG se han empleado las herramientas *software* que se detallan en la Tabla PR2.

Recurso	Valor de adquisición (€)	Vida útil (meses)	Coste mensual (€)	Tiempo de uso (meses)	Importe (€)
QuestaSim	20.000	60	333,33	4	1.333,33
X2Go	0	0	0	0	0
Red Hat Linux	0	0	0	0	0
Gedit	0	0	0	0	0
				Total	1.333,33

Tabla PR2: Costes asociados a los recursos software.

Los costes totales asociados a los recursos materiales, correspondientes a los recursos hardware y software ascienden a mil trescientos noventa y dos euros con cincuenta y tres céntimos (1.392,53 €).

PR4. Material Fungible

Los costes asociados al material fungible utilizado en la realización de este TFG se muestran en la Tabla PR3.

Material	Coste (€)
Material de papelería	10,00
CD-ROM	0,50
Toner de impresora	50,00
Encuadernación	5,00
Total	65,50

Tabla PR3: Costes asociados a los materiales fungibles.

PR5. Coste total del proyecto

Para la determinación del coste total asociado a la realización de este TFG el valor del Impuesto General Indirecto Canario (IGIC) impone el presupuesto calculado con un incremento de un 7%. Teniendo en cuenta este factor, el coste total del presente TFG se desglosa en la Tabla PR4.

Material	Coste (€)
Recursos humanos	3.378,00
Recursos materiales	1.392,53
Material fungible	65,50
Subtotal	4.836,03
I.G.I.C. (7 %)	338,52
Total	5.174,55

Tabla PR4: Coste total del TFG.

El importe final al que asciende el presupuesto del presente TFG es de un total de cinco mil ciento setenta y cuatro euros con cincuenta y cinco céntimos (5.174,55 €).

Fdo: Alejandro Piñan Roescher

Las Palmas de Gran Canaria, a 18 de Julio de 2017

Parte IV

Anexos

Anexo I – hillcipher_sequencer.sv

```
1  class hillcipher_sequencer extends uvm_sequencer #(data_packet);
2
3     `uvm_sequencer_utils(hillcipher_sequencer)
4
5     function new(string name, uvm_component parent);
6        super.new(name, parent);
7     endfunction: new
8  endclass: hillcipher_sequencer
```

Anexo II – Resultados de la simulación por ventana de comandos

```
vsim -c top -do "run -all; quit -f"
Reading pref.tcl

# 10.4b

# vsim -c top -do "run -all; quit -f"
# Start time: 19:26:26 on Jul 17,2017
# ** Note: (vsim-8009) Loading existing optimized design _opt
# //  Questa Sim-64
# //  Version 10.4b linux_x86_64 May 26 2015
# //
# //  Copyright 1991-2015 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# //  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
# //  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
# //   THIS DOCUMENT CONTAINS TRADE SECRETS AND COMMERCIAL OR
FINANCIAL
# //  INFORMATION THAT ARE PRIVILEGED, CONFIDENTIAL, AND EXEMPT FROM
# //   DISCLOSURE UNDER THE FREEDOM OF INFORMATION ACT, 5 U.S.C.
SECTION 552.
# //  FURTHERMORE, THIS INFORMATION IS PROHIBITED FROM DISCLOSURE
UNDER
# //  THE TRADE SECRETS ACT, 18 U.S.C. SECTION 1905.
# //
# Loading sv_std.std
# Loading mtiUvm.uvm_pkg
# Loading work.hillcipher_pkg(fast)
# Loading work.top(fast)
# Loading mtiUvm.questa_uvm_pkg(fast)
# Loading work.hillcipher_if(fast)
#                                                          Loading
/tmp/apinan@femes.iuma.ulpgc.es_dpi_1358/linux_x86_64_gcc-4.7.4/v
sim_auto_compile.so
#                                                          Loading
/home/soft/eucad/mentor/2015/questa_sv_af_10.4b_Linux/questasim/u
vm-1.1d/linux_x86_64/uvm_dpi.so
# run -all
#
----------------------------------------------------------------
# UVM-1.1d
# (C) 2007-2013 Mentor Graphics Corporation
# (C) 2007-2013 Cadence Design Systems, Inc.
# (C) 2006-2013 Synopsys, Inc.
# (C) 2011-2013 Cypress Semiconductor Corp.
#
----------------------------------------------------------------
#
#   ***********       IMPORTANT RELEASE NOTES         ************
#
#   You are using a version of the UVM library that has been compiled
#   with `UVM_NO_DEPRECATED undefined.
#   See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
```

```
56  #   You are using a version of the UVM library that has been compiled
57  #   with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
58  #   See http://www.eda.org/svdb/view.php?id=3770 for more details.
59  #
60  #       (Specify +UVM_NO_RELNOTES to turn off this notice)
70  #
71  #                                                          UVM_INFO
72  verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215)   @   0:
73  reporter [Questa UVM] QUESTA_UVM-1.2.2
74  #                                                          UVM_INFO
75  verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217)   @   0:
76  reporter [Questa UVM]  questa_uvm::init(+struct)
77  # UVM_INFO @ 0: reporter [RNTST] Running test hillcipher_test...
78  # UVM_INFO dut_env.sv(32) @ 0: uvm_test_top.env [uvm_test_top.env]
79  Build stage complete.
80  #  UVM_INFO  hillcipher_env.sv(14)  @  0:  uvm_test_top.env.henv_in
81  [uvm_test_top.env.henv_in] Build stage complete.
82  #       UVM_INFO       hillcipher_agent.sv(28)       @       0:
83  uvm_test_top.env.henv_in.agent   [uvm_test_top.env.henv_in.agent]
84  Build stage complete.
85  #       UVM_INFO       hillcipher_driver.sv(14)       @       0:
86  uvm_test_top.env.henv_in.agent.driver
87  [uvm_test_top.env.henv_in.agent.driver] Build stage complete.
88  #       UVM_INFO       hillcipher_monitor.sv(27)       @       0:
89  uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
90  INTERFACE USED = in_intf
91  #       UVM_INFO       hillcipher_monitor.sv(35)       @       0:
92  uvm_test_top.env.henv_in.agent.monitor
93  [uvm_test_top.env.henv_in.agent.monitor] Build stage complete.
94  #  UVM_INFO  hillcipher_env.sv(14)  @  0:  uvm_test_top.env.henv_out
95  [uvm_test_top.env.henv_out] Build stage complete.
96  #       UVM_INFO       hillcipher_agent.sv(28)       @       0:
97  uvm_test_top.env.henv_out.agent [uvm_test_top.env.henv_out.agent]
98  Build stage complete.
99  #       UVM_INFO       hillcipher_monitor.sv(27)       @       0:
100 uvm_test_top.env.henv_out.agent.monitor      [hillcipher_monitor]
101 INTERFACE USED = out_intf
102 #       UVM_INFO       hillcipher_monitor.sv(35)       @       0:
103 uvm_test_top.env.henv_out.agent.monitor
104 [uvm_test_top.env.henv_out.agent.monitor] Build stage complete.
105 #       UVM_INFO       hillcipher_agent.sv(34)       @       0:
106 uvm_test_top.env.henv_in.agent   [uvm_test_top.env.henv_in.agent]
107 Connect stage complete.
108 #       UVM_INFO       hillcipher_agent.sv(34)       @       0:
109 uvm_test_top.env.henv_out.agent [uvm_test_top.env.henv_out.agent]
110 Connect stage complete.
111 # UVM_INFO dut_env.sv(38) @ 0: uvm_test_top.env [uvm_test_top.env]
112 Connect phase complete.
113 #  UVM_INFO  test_lib.sv(19)  @  0:  uvm_test_top  [hillcipher_test]
114 Printing the test topology :
115 #
116 ------------------------------------------------------------------
117 ----------------
118 #  Name                                                      Type
119 Size  Value
120 #
121 ------------------------------------------------------------------
122 ----------------
123 #  uvm_test_top                                    hillcipher_test
124 -     @466
125 #      env                                                 dut_env
```

```
126  -      @474
127  #          henv_in                                    hillcipher_env
128  -      @500
129  #             agent                                    hillcipher_agent
130  -      @524
131  #                driver                                hillcipher_driver
132  -      @660
133  #                    rsp_port                          uvm_analysis_port
134  -      @677
135  #                   seq_item_port                uvm_seq_item_pull_port
136  -      @668
137  #                monitor                               hillcipher_monitor
138  -      @686
139  #                           item_collected_port        uvm_analysis_port
140  -      @701
141  #                    is_monitor_active          uvm_active_passive_enum
142  1      UVM_ACTIVE
143  #                sequencer                            hillcipher_sequencer
144  -      @537
145  #                    rsp_export                        uvm_analysis_export
146  -      @545
147  #                     seq_item_export              uvm_seq_item_pull_imp
148  -      @651
149  #                                  arbitration_queue                 array
150  0      -
151  #                           lock_queue                               array
152  0      -
153  #                            num_last_reqs                        integral
154  32     'd1
155  #                            num_last_rsps                        integral
156  32     'd1
157  #                is_agent_active                uvm_active_passive_enum
158  1      UVM_ACTIVE
159  #            henv_out                                     hillcipher_env
160  -      @508
161  #             agent                                      hillcipher_agent
162  -      @723
163  #                monitor                               hillcipher_monitor
164  -      @736
164  #                           item_collected_port        uvm_analysis_port
165  -      @750
166  #                    is_monitor_active          uvm_active_passive_enum
167  1      UVM_PASSIVE
168  #                is_agent_active                uvm_active_passive_enum
169  1      UVM_PASSIVE
170  #              sb                                   hillcipher_scoreboard
171  -      @516
172  #               cmp                                          sb_comparator
173  -      @802
174  #            aport_active_monitor    uvm_analysis_imp_active_monitor
175  -      @810
176  #            aport_pasive_monitor    uvm_analysis_imp_pasive_monitor
177  -      @819
178  #             exp_in                                   uvm_analysis_export
179  -      @767
180  #              exp_out                                 uvm_analysis_export
181  -      @776
182  #               prd                                           sb_predictor
183  -      @785
184  #                    analysis_imp                       uvm_analysis_imp
185  -      @793
```

```
#                results_ap                          uvm_analysis_port
-     @844
#
------------------------------------------------------------------
----------------
#
#       UVM_INFO       hillcipher_driver.sv(25)       @       0:
uvm_test_top.env.henv_in.agent.driver         [hillcipher_driver]
Resetting signals ...
#       UVM_INFO       hillcipher_driver.sv(40)       @       0:
uvm_test_top.env.henv_in.agent.driver [hillcipher_driver] Reciving
signals ...
#  UVM_INFO  sb_predictor.sv(28)  @  55:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 97  REPORT:TIME = 55
#  UVM_INFO  sb_predictor.sv(28)  @  65:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 99  REPORT:TIME = 65
#  UVM_INFO  sb_predictor.sv(28)  @  75:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 116  REPORT:TIME = 75
#  UVM_INFO  sb_predictor.sv(34)  @  85:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 6 REPORT:TIME = 85
#  UVM_INFO  sb_predictor.sv(34)  @  95:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 24 REPORT:TIME = 95
#  UVM_INFO  sb_predictor.sv(34)  @ 105:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 1 REPORT:TIME = 105
#  UVM_INFO  sb_predictor.sv(34)  @ 115:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 13 REPORT:TIME = 115
#  UVM_INFO  sb_predictor.sv(34)  @ 125:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 16 REPORT:TIME = 125
#  UVM_INFO  sb_predictor.sv(34)  @ 135:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 10 REPORT:TIME = 135
#  UVM_INFO  sb_predictor.sv(34)  @ 145:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 20 REPORT:TIME = 145
#  UVM_INFO  sb_predictor.sv(34)  @ 155:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 17 REPORT:TIME = 155
#  UVM_INFO  sb_predictor.sv(34)  @ 165:  uvm_test_top.env.sb.prd
[sb_predictor] REPORT: data_Colected_s = 15 REPORT:TIME = 165
#       UVM_INFO      hillcipher_monitor.sv(91)       @      165:
uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
REPORT: DATA IDLE = 0 REPORT:TIME = 165
#
#           Golden reference prediction
#
#
# Enter 3x3 matrix for key (It should be inversible):
# 6.000000 24.000000  1.000000
# 13.000000   16.000000  10.000000
# 20.000000   17.000000  15.000000
#
# Enter a 3 letter string: act
# Encrypted string is: poh
#
# Inverse Matrix is:
# 0.158730 -0.777778 0.507937
# 0.011338 0.158730 -0.106576
# -0.224490 0.857143 -0.489796
#
# Decrypted string is: act
#  UVM_INFO  sb_comparator.sv(80)  @ 175:  uvm_test_top.env.sb.cmp
[sb_comparator run task] WAITING for expected output
#  UVM_INFO  sb_comparator.sv(84)  @ 175:  uvm_test_top.env.sb.cmp
[sb_comparator run task] WAITING for actual output
```

```
247 # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
248 [PASS_msg ] Expected=a   Actual=a
249 #
250 # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
251 [PASS_msg ] Expected=c   Actual=c
252 #
253 # UVM_INFO sb_comparator.sv(103) @ 175: uvm_test_top.env.sb.cmp
254 [PASS_msg ] Expected=t   Actual=t
255 #
256 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
257 [PASS_elem ] Expected=6.000000   Actual=6.000000
258 #
259 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
260 [PASS_elem ] Expected=24.000000   Actual=24.000000
261 #
262 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
263 [PASS_elem ] Expected=1.000000   Actual=1.000000
264 #
265 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
266 [PASS_elem ] Expected=13.000000   Actual=13.000000
267 #
268 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
269 [PASS_elem ] Expected=16.000000   Actual=16.000000
270 #
271 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
272 [PASS_elem ] Expected=10.000000   Actual=10.000000
273 #
274 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
275 [PASS_elem ] Expected=20.000000   Actual=20.000000
276 #
277 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
278 [PASS_elem ] Expected=17.000000   Actual=17.000000
279 #
280 # UVM_INFO sb_comparator.sv(120) @ 175: uvm_test_top.env.sb.cmp
281 [PASS_elem ] Expected=15.000000   Actual=15.000000
282 #
283 # UVM_INFO sb_predictor.sv(28) @ 185: uvm_test_top.env.sb.prd
284 [sb_predictor] REPORT: data_Colected_s = 112  REPORT:TIME = 185
285 # UVM_INFO sb_predictor.sv(28) @ 195: uvm_test_top.env.sb.prd
286 [sb_predictor] REPORT: data_Colected_s = 118  REPORT:TIME = 195
287 # UVM_INFO sb_predictor.sv(28) @ 205: uvm_test_top.env.sb.prd
288 [sb_predictor] REPORT: data_Colected_s = 112  REPORT:TIME = 205
289 # UVM_INFO sb_predictor.sv(34) @ 215: uvm_test_top.env.sb.prd
290 [sb_predictor] REPORT: data_Colected_s = 1 REPORT:TIME = 215
291 # UVM_INFO sb_predictor.sv(34) @ 225: uvm_test_top.env.sb.prd
292 [sb_predictor] REPORT: data_Colected_s = 2 REPORT:TIME = 225
293 # UVM_INFO sb_predictor.sv(34) @ 235: uvm_test_top.env.sb.prd
294 [sb_predictor] REPORT: data_Colected_s = 3 REPORT:TIME = 235
295 # UVM_INFO sb_predictor.sv(34) @ 245: uvm_test_top.env.sb.prd
296 [sb_predictor] REPORT: data_Colected_s = 13 REPORT:TIME = 245
297 # UVM_INFO sb_predictor.sv(34) @ 255: uvm_test_top.env.sb.prd
298 [sb_predictor] REPORT: data_Colected_s = 16 REPORT:TIME = 255
299 # UVM_INFO sb_predictor.sv(34) @ 265: uvm_test_top.env.sb.prd
300 [sb_predictor] REPORT: data_Colected_s = 10 REPORT:TIME = 265
301 # UVM_INFO sb_predictor.sv(34) @ 275: uvm_test_top.env.sb.prd
302 [sb_predictor] REPORT: data_Colected_s = 20 REPORT:TIME = 275
303 # UVM_INFO sb_predictor.sv(34) @ 285: uvm_test_top.env.sb.prd
304 [sb_predictor] REPORT: data_Colected_s = 17 REPORT:TIME = 285
205 # UVM_INFO sb_predictor.sv(34) @ 295: uvm_test_top.env.sb.prd
306 [sb_predictor] REPORT: data_Colected_s = 15 REPORT:TIME = 295
307 #      UVM_INFO      hillcipher_monitor.sv(91)      @      295:
```

```
uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
REPORT: DATA IDLE = 0 REPORT:TIME = 295
#
#           Golden reference prediction
#
#
# Enter 3x3 matrix for key (It should be inversible):
# 1.000000 2.000000   3.000000
# 13.000000   16.000000  10.000000
# 20.000000   17.000000  15.000000
#
# Enter a 3 letter string: pvp
# Encrypted string is: yfy
#
# Inverse Matrix is:
# -0.322581 -0.096774 0.129032
# -0.023041 0.207373 -0.133641
# 0.456221 -0.105991 0.046083
#
# Decrypted string is: pvp
#  UVM_INFO  sb_comparator.sv(80)  @  305:  uvm_test_top.env.sb.cmp
[sb_comparator run task] WAITING for expected output
#  UVM_INFO  sb_comparator.sv(84)  @  305:  uvm_test_top.env.sb.cmp
[sb_comparator run task] WAITING for actual output
#  UVM_INFO  sb_comparator.sv(103)  @  305:  uvm_test_top.env.sb.cmp
[PASS_msg ] Expected=p   Actual=p
#
#  UVM_INFO  sb_comparator.sv(103)  @  305:  uvm_test_top.env.sb.cmp
[PASS_msg ] Expected=v   Actual=v
#
#  UVM_INFO  sb_comparator.sv(103)  @  305:  uvm_test_top.env.sb.cmp
[PASS_msg ] Expected=p   Actual=p
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=1.000000   Actual=1.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=2.000000   Actual=2.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=3.000000   Actual=3.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=13.000000   Actual=13.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=16.000000   Actual=16.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=10.000000   Actual=10.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=20.000000   Actual=20.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=17.000000   Actual=17.000000
#
#  UVM_INFO  sb_comparator.sv(120)  @  305:  uvm_test_top.env.sb.cmp
[PASS_elem ] Expected=15.000000   Actual=15.000000
#
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
```

```
1795: reporter [TEST_DONE] 'run' phase is ready to proceed to the
'extract' phase
#      UVM_INFO      hillcipher_monitor.sv(130)      @      1795:
uvm_test_top.env.henv_in.agent.monitor       [hillcipher_monitor]
REPORT: COLLECTED PACKETS = 26
#      UVM_INFO      hillcipher_monitor.sv(130)      @      1795:
uvm_test_top.env.henv_out.agent.monitor      [hillcipher_monitor]
REPORT: COLLECTED PACKETS = 24
# UVM_INFO sb_comparator.sv(134) @ 1795: uvm_test_top.env.sb.cmp
[sb_comparator]
#
#
# *** TEST PASSED - 24 vectors ran, 24 vectors passed ***
#
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   77
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [PASS_elem ]    18
# [PASS_msg ]     6
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [hillcipher_driver]     2
# [hillcipher_monitor]     6
# [hillcipher_test]     1
# [sb_comparator]     1
# [sb_comparator run task]     4
# [sb_predictor]    24
# [uvm_test_top.env]     2
# [uvm_test_top.env.henv_in]     1
# [uvm_test_top.env.henv_in.agent]     2
# [uvm_test_top.env.henv_in.agent.driver]     1
# [uvm_test_top.env.henv_in.agent.monitor]     1
# [uvm_test_top.env.henv_out]     1
# [uvm_test_top.env.henv_out.agent]     2
# [uvm_test_top.env.henv_out.agent.monitor]     1
#       **       Note:       $finish                                :
/home/soft/eucad/mentor/2015/questa_sv_af_10.4b_Linux/questasim/l
inux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 1795 ns  Iteration: 53  Instance: /top
# End time: 19:26:30 on June 17,2017, Elapsed time: 0:00:04
# Errors: 0, Warnings: 0
```

Anexo III – Contenido del CD-ROM

El contenido del CD-ROM adjunto a este documento se corresponde con la siguiente estructura:

- Memoria del TFG “Integración de un modelo de referencia descrito en C en un entorno de verificación UVM” en formato PDF.

- Directorio “Códigos” en el que se encuentran los diferentes códigos que forman el entorno realizado en el desarrollo de este TFG.