Universidad Nacional del Sur

Tesis de Maestría en Ingeniería

Verificación de Circuitos Digitales Descritos en HDL

Juan I. Francesconi

Bahía Blanca Argentina

2015

Impreso en Argentina.
ISBN XXX-XXX-XXXX-XX-X
Septiembre de 2015

Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado académico de Magister en Ingeniería, de la la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones realizadas en el ámbito del Departamento de Ingeniería Eléctrica y Computadoras en el período comprendido entre Agosto del 2012 y Agosto del 2014, bajo la dirección del Dr. Pedro Julian y del Dr. Agustin Rodriguez.

Bahía Blanca, 7 de Abril de 2015.

Juan I. Francesconi
Departamento de Ingeniería Eléctrica y de Computadoras
Universidad Nacional del Sur

Universidad Nacional del Sur
Secretaría General de Posgrado y Educación
Continua

La presente tesis ha sido aprobada el ..../..../..... , mereciendo la calificación de .....(.....)

Resumen

En el ámbito del diseño de circuitos integrados, el proceso de asegurar que la intención del diseño es mapeada correctamente en su implementación se denomina verificación funcional. En este contexto, los errores lógicos son discrepancias entre el comportamiento previsto del dispositivo y su comportamiento observado.

La verificación funcional es hoy en día el cuello de botella del flujo de diseño digital y la simulación de eventos discretos sigue siendo la técnica de verificación más utilizada, principalmente debido a que es la única aplicable a sistemas grandes y complejos.

En este trabajo se aborda, mediante un enfoque teórico práctico, dos de los conceptos más relevantes de la verificación funcional de hardware basada en simulación, esto es, la arquitectura de los *testbenches*, los cuales le dan soporte práctico a dicha técnica y los modelos de cobertura funcional, los cuales definen las funcionalidades y escenarios que deben ser probados guiando de esta forma la creación de pruebas y el respectivo progreso del proceso de verificación.

En primer lugar, se encara la temática de la arquitectura de los *testbenches* modernos identificando las propiedades deseadas de los mismos, reusabilidad y facilidad para aumentar el nivel de abstracción. En función de estas dos propiedades se selecciona la metodología de *Universal Verification Methodology* (UVM) para el diseño, análisis e implementación de dos *testbenches*.

En segundo lugar, dada la problemática del crecimiento del espacio de prueba de los diseños modernos y la subsecuente dificultad para generar modelos de cobertura adecuados para los mismos a partir de sus especificaciones, se introduce un método empírico de caja negra para derivar un modelo de cobertura para diseños dominados por el control. Este método está basado en la utilización de un modelo abstracto de la funcionalidad del dispositivo bajo prueba (DUV, sigla en inglés de *Device Under Verification*). Este modelo facilita la extracción de conjuntos de secuencias de prueba, los cuales representan el modelo de cobertura funcional. Dada la complejidad de los posibles espacios de prueba generados, las conocidas técnicas de *Testing de Software*, partición en clases de equivalencia y análisis de valores límites, son aplicadas para reducirlos. Adicionalmente, se desarrolla una notación formal para expresar las secuencias de prueba equivalentes extraídas del modelo.

Por ultimo se aplica el método de derivación de modelo de cobertura funcional presentado para obtener casos de prueba relevantes para un modulo de *buffer* FIFO, y se utiliza el *testbench* implementado para darle soporte a la ejecución de dichos casos de prueba, implementando las pruebas derivadas y los correspondientes puntos de cobertura, combinando de esta forma los dos conceptos abordados.

Abstract

In the integrated circuit design field, the process of assuring that the design intent is properly mapped in its implementation is known as functional verification. In this context, logic errors are discrepancies between the device's expected behavior and its observed behavior.

The functional verification of a design is now a days the bottleneck of the digital design flow and discrete event simulation still is the most used verification technique, mostly because it is the only technique which is applicable to big and complex systems.

In this work, through a theoretical and practical approach, two of the most relevant simulation based hardware functional verification concepts are addressed. Those concepts are, the testbench architecture, which gives practical support to the simulation technique, and the functional coverage model, which defines the functionalities and scenarios that should be tested, guiding the creation of tests and the measurement of the verification process's progress.

In first place, modern testbench architectures are studied identifying their desired properties, which are reusability and the facility to raise the level of abstraction. According to these properties the Universal Verification Methodology (UVM) is chosen for the design, analysis and implementation of two testbenches.

In second place, given the test space growth challenge of modern designs and the subsequent difficulty for generating their appropriate coverage models from their specifications, an empirical black box method is introduced for the creation of coverage models for control dominated designs. This method is based in the definition of a functional model of the DUV (Design Under Verification) which facilitates the extraction of sets of test sequences which define a functional coverage model. Given the complexity of the test space, the well known software testing techniques, equivalence class partition and limit value analysis, are applied to reduce it. A formal notation is developed in order to express equivalent test sequences.

Lastly, the presented functional coverage creation method is applied to a FIFO (First Input First Output) buffer module in order to obtain relevant test sequences, and one of the previously implemented testbench is used to give support to the execution of those test cases, implementing the test sequences and its corresponding coverage points, combining in this manner both of this

work addressed concepts.

Agradecimientos

Agradezco a mis directores, el Dr. Pedro Julian y el Dr. Agustín Rodriguez, por brindar su confianza y la asistencia para que este trabajo pudiera ser realizado. A TEAC y ANPCyT por la beca que me fue otorgada. A la Universidad Nacional del Sur y a todos los miembros del Grupo de Investigación en Sistemas Electrónicos y Electromecatrónicos (GISEE) por brindarme el espacio y los medios para el desarrollo de esta tesis. A los compañeros de laboratorio, el Ing. Gabriel Pachiana y el Ing. Manuel Soto, por las valiosas discusiones y su espíritu de superación. También quisiera agradecer a la Dra. Liliana Fraigi, directora del centro INTI CMNB Bahía Blanca y al Dr. Martín Di Federico, jefe de Unidad Técnica del mismo centro, actual lugar de trabajo, por confiar en mi y brindarme la posibilidad y el tiempo necesario para terminar con la Maestría.

También me gustaría agradecer a los Jurados, por aceptar el trabajo de evaluar la Tesis, y tomarse el tiempo necesario para hacerlo, aportando valiosas sugerencias y correcciones.

Y por supuesto agradecer a mis amigos y familia por estar siempre, sobre todo a mi mujer Nazareth por aguantarme y a mi padre Jose, por confiar y brindarme siempre su apoyo y motivación.

Índice general

Capítulo 1

Introducción

1.1. Problemática

Con el rápido desarrollo de las herramientas de Automatización de Diseño Electrónico (EDA, sigla en inglés de *Electronic Design Automation*), las tecnologías de fabricación de Circuitos Integrados (IC, sigla en inglés de *Integrated Circuit*) y los nuevos métodos de síntesis automática, el proceso de fabricación ha entrado en la era sub micrón. La disminución del tamaño del chip junto con el aumento del número de transistores en el mismo hace posible integrar miles de millones de transistores en un único chip. Las técnicas de diseño y fabricación de IC hacen posible que más bloques de Propiedad Intelectual (IP, sigla en inglés de *Intellectual Property*), más memorias y buses sean integrados en un solo chip, generando sistemas de alta complejidad.

Sin embargo, las capacidades de diseño y fabricación de IC han superado por largo tiempo a la capacidad de verificación. La *International Technology Roadmap for Semiconductors* (ITRS) señaló que la verificación ha sido la etapa en el flujo de diseño de circuitos que más tiempo consume. En los proyectos de ingeniería actuales, el número de ingenieros de verificación ha superado al de ingenieros de diseño. Para diseños complejos, la relación llega a 2:1 o 3:1, lo que conduce a la verificación a ser el cuello de botella actual del proceso de diseño. Si no se producen avances en el tema, la verificación se convertirá en la principal barrera para el futuro de la industria de diseño de IC [50].

El desafío de la verificación es un tema de relevancia en la academia y la industria. Estudios recientes estiman que la mitad de todos los chips diseñados hoy en día requieren fabricarse una o más veces debido a errores. Más importante aún, estos estudios indican que el 77 % de estas fallas son errores funcionales [14]. En muchos casos la verificación consume más recursos, en términos de tiempo y trabajo, que todas las demás etapas del flujo de diseño combinadas. Hoy, en la era de los Circuitos Integrados de Aplicación Específica (ASIC, sigla en inglés de *Application-Specific*

Integrated Circuit) de varios millones de compuertas, los IP reutilizables y los *System on Chip* (SoC), la verificación consume alrededor del 70 % del presupuesto de un proyecto [49]. De esta manera, es evidente que la verificación es el cuello de botella en todo proyecto de diseño de IC.

Dada la cantidad de esfuerzo que exige la verificación, la falta de ingenieros de diseño y verificación de hardware, y la cantidad de código que debe producirse, no es de extrañar que, en la mayoría de los proyectos, la verificación recaiga en la ruta crítica. También es la razón por la que la verificación es actualmente blanco de nuevas herramientas y metodologías. Estas herramientas y metodologías intentan reducir el tiempo de verificación permitiendo optimizar el esfuerzo mediante el manejo de mayores niveles de abstracción y automatización.

Debido a la variedad de funciones, interfaces, protocolos y transformaciones que deben ser verificadas, no es posible, dada la tecnología actual, proporcionar una solución automatizada de propósito general para la verificación. Es posible automatizar solo una parte del proceso de verificación, especialmente cuando se aplica a un dominio de aplicación acotado.

En el ámbito del diseño de circuitos integrados, el proceso de asegurar que la intención del diseño es mapeada correctamente en su implementación se denomina verificación funcional [4]. La verificación funcional no se encarga de verificar cuestiones temporales del funcionamiento del diseño, como por ejemplo que se respeten los tiempos de *hold* y *setup* de las señales, tarea que generalmente se realiza, para sistemas sincrónicos, mediante la técnica denominada análisis estático del temporizado. La verificación funcional tampoco se encarga de realizar la validación post silicio del diseño.

En este contexto, los errores lógicos son discrepancias entre el comportamiento previsto del dispositivo y su comportamiento observado. Estos errores son introducidos por el diseñador debido a una especificación ambigua, una mala interpretación de la misma, o un error tipográfico durante la codificación del modelo. La verificación funcional se puede llevar a cabo usando métodos dinámicos/simulaciones, métodos estáticos/formales, o métodos híbridos [51].

Con la simulación, un modelo del circuito es generado y simulado. Los modelos son creados normalmente utilizando Lenguajes de Descripción de Hardware (HDL, sigla en inglés de *Hardware Definition Languaje*) estandarizados como *Verilog* o VHDL. Sobre la base de estímulos, caminos de ejecución del modelo del chip son examinados usando un simulador. Estos estímulos pueden ser proporcionados por un usuario, o por medios automatizados tales como un generador aleatorio. La discrepancia entre la salida del simulador y la salida que describe la especificación determina la presencia de errores. La simulación es la técnica de verificación de hardware más popular y se utiliza en las distintas etapas de diseño. El entorno de verificación presenta al dispositivo de prueba una abstracción de su entorno operativo, aunque por lo general exagera los parámetros de estímulos con el fin de estresar el dispositivo. El medio de verificación también registra los avances de verificación usando una variedad de métricas denominadas cobertura.

En la verificación formal, especificación y diseño se traducen en modelos matemáticos. Estas técnicas verifican un diseño, demostrando su correctitud a través de varios tipos de razonamientos matemáticos [17].

Los métodos formales de verificación más populares son: chequeo de modelos/propiedades, demostración de teoremas y chequeo de equivalencia. Estos métodos recurren a herramientas lógicas, axiomáticas, proposicionales, de lógica temporal, etc. para probar que un sistema cumple con determinadas propiedades.

La mayoría de los métodos de verificación formal se ven obstaculizados por el problema de la explosión combinatoria de estados, lo que causa una complejidad exponencial en los algoritmos que los computan. Por lo tanto, dichos métodos, restringen su aplicación a pequeños bloques funcionales de un dispositivo. Al mismo tiempo, los métodos formales producen una verificación completa, integral, de la propiedad demostrada. En conjunto, esto da lugar a la aplicación de métodos estáticos para unidades pequeñas, de lógica compleja, tales como controladores de bus.

Los métodos híbridos, también conocidos como semi-formales, combinan técnicas estáticas y dinámicas a fin de superar las limitaciones de capacidad impuestas por los métodos estáticos y abordan las limitaciones inherentes a la completitud de la verificación de los métodos dinámicos.

1.2. Objetivos

El objetivo general del trabajo consiste en comprender, aplicar y analizar, desde un punto de vista teórico-practico, dos de los conceptos fundamentales de la verificación funcional de hardware:

- La arquitectura de los *testbenches*
- Los modelos de cobertura funcional

El primer objetivo puntual plantea desarrollar los conocimientos y las capacidades, mediante el estudio bibliográfico y la experimentación con casos de estudios, para usar, diseñar e implementar *testbenches* estructurados y reusables, utilizando tecnología del estado del arte, y en función de esto poder analizar su arquitectura en pos de facilitar y mejorar el proceso de verificación.

El segundo objetivo puntual consiste en desarrollar los conocimientos y las capacidades para definir modelos de cobertura funcional de un diseño a partir de modelos funcionales de alto nivel del mismo, facilitando la exploración del espacio de cobertura y mejorando de esta forma el proceso de creación de pruebas.

El objetivo final consiste en lograr integrar los dos objetivos anteriores mediante un *testbench* reusable que brinde soporte para las pruebas extraídas a partir del modelo de cobertura definido, facilitando el mapeo de estos conjuntos de pruebas en pruebas ejecutables, verificando automáticamente el comportamiento del diseño al ser estimulado con dichas pruebas, y midiendo la cobertura de las mismas.

1.3. Marco de Trabajo

Este trabajo se enmarcó dentro de los siguientes proyectos de investigación:

- FS TICS 001 TEAC 2010 : Plataforma Tecnológica para Sistemas de Tecnología Electrónica de Alta Complejidad.
- PICT 2010 2657 : 3D Gigascale Integrated Circuits for Nonlinear Computation, Filter and Fusion with Applications in Industrial Field Robotics.
- PAE 37079 : Proyecto Integrado en el Área de Microelectrónica para el Diseño de Circuitos Integrados.

El mismo se realizó mediante la siguiente beca:

- Proyecto FS TICS N° 001/10 - Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) a través de FONARSEC. Beca de Maestría tipo inicial. Periodo: Agosto 2012 – Agosto 2014.

Las herramientas EDA utilizadas en el trabajo fueron provistas por *Synopsys* gracias al programa universitario (*Synopsys Worldwide University Program*) que la compañía tiene con la Universidad Nacional del Sur.

1.4. Estructura del Trabajo

El trabajo se organiza de manera tal que en el Capítulo 2 se introduce la temática presentando el marco teórico, describiendo los conceptos básicos de la verificación funcional basada en simulación, dándole especial relevancia a la arquitectura de los entornos de verificación modernos y al concepto de cobertura, presentando un estudio bibliográfico de los respectivos temas. En el Capítulo 3 se aborda la temática de la arquitectura de los *testbenches*, concretamente la propuesta por la metodología de verificación denominada *Universal Verification Methodology* (UVM) mediante la experimentación con dos casos de estudio, un módulo de *buffer* FIFO (*First Input*

First Output) y un módulo de memoria I2C EEPROM (*Electrical Eraseable Programmable Read Only Memory*), para los cuales se diseñan, implementan y analizan los respectivos *testbenches*. En el Capítulo 4 se aborda la temática de la cobertura funcional analizando y haciendo un paralelismo con diversas técnicas de *Testing de Software*, como *Model-based Testing*, para luego presentar una técnica empírica para extraer un modelo de cobertura a partir de un modelo abstracto del diseño. Luego se aplica dicha propuesta para desarrollar un modelo de cobertura funcional para la verificación del modulo de *buffer* FIFO utilizando el *testbench* presentado en el capítulo anterior. En el último capítulo se concluye el trabajo resaltando y analizando los puntos más importantes del desarrollo, y presentando algunas propuestas de trabajo futuro y posible linea de investigación.

Capítulo 2

Marco Teórico

En el presente capítulo se introducen los aspectos fundamentales de la verificación funcional basada en simulación, esto es, las métricas de cobertura, la generación de estímulos y la comprobación de la respuesta. Luego se ahonda en las distintas arquitecturas de los entornos de verificación modernos, los cuales son los encargados de darle soporte a los aspectos mencionados. Finalmente se presenta el concepto de modelo de cobertura funcional, el cual representa la base del proceso de verificación. Se introducen diferentes clases de modelos de cobertura funcional haciendo foco en los modelos basados en grafos de transición de estados.

2.1. Aspectos Fundamentales de la Verificación Funcional

Para realizar correctamente un proceso de verificación funcional se debe, primero, definir que aspectos del diseño se deben verificar y luego, como se realizará dicha verificación. Estos aspectos, en conjunto, describen el alcance del problema de verificación para el dispositivo en cuestión, y se convierten en las especificaciones funcionales del entorno de verificación que dará soporte a dicho proceso.

En función de lo descrito, el proceso de verificación comprende tres aspectos fundamentales. El primero es el alcance del problema de verificación, formado por los modelos de cobertura. El segundo es la generación de estímulos, comprendido por la infraestructura necesaria para implementar las pruebas requeridas para cubrir el modelo de cobertura. El tercero es la comprobación de la respuesta, formado también por el mecanismo de la infraestructura de verificación implementado para comparar la respuesta del dispositivo con la respuesta esperada para un estímulo específico.

Se puede observar que los dos últimos aspectos forman parte de la arquitectura y de la implementación del entorno de verificación y ambos se encargan de dar soporte al aspecto prioritario

de la verificación funcional: la cobertura.

En las siguientes sub secciones se introducen cada uno de estos aspectos.

2.1.1. Métricas de Cobertura

La cobertura es una métrica de simulación usada para medir el progreso y completitud del proceso de verificación. La misma es el aspecto principal del proceso de verificación, ya que describe el alcance del problema de verificación y como está particionado.

La cobertura es esencial para evaluar y guiar el proceso de verificación. El ideal es poder lograr una verificación exhaustiva sin esfuerzos redundantes [46]. Las métricas de cobertura ayudan a aproximar este ideal, de la siguiente manera:

- Actuando como medidas heurísticas que cuantifican la completitud de la verificación.
- Identificando los aspectos del diseño ejercitados inadecuadamente y guiando la generación de nuevos estímulos de prueba hacia áreas inexploradas del diseño, asegurando un uso óptimo de los recursos de simulación.

Se podría decir que el mayor conocimiento y creatividad del proceso de verificación se requiere para definir el modelo de cobertura, es decir, la definición de los aspectos del diseño que se probarán y en los que se hará foco.

El análisis de la cobertura provee retroalimentación de la precisión y efectividad de las pruebas aleatorias. También sirve como un indicador de calidad de la convergencia de la verificación y por lo tanto como un criterio importante para decidir si el diseño está listo para enviarse a fabricar.

Casi todos los ingenieros sostienen que la instrumentación de la cobertura fuerza al estudio de la arquitectura y los detalles del RTL (*Register Transfer Level*), y consecuentemente lleva al desarrollo de mejores pruebas [52]. El análisis de los huecos de la cobertura revela aquellos casos cuya ocurrencia es imposible y resalta factores que fueron originalmente ignorados. Esto sirve para modificar el plan de cobertura inicial.

Otro beneficio derivado de la medición y análisis de la cobertura es el aumento del nivel de confianza del proceso de verificación. Mientras que la verificación consista en pruebas aleatorias, la incertidumbre sobre los casos de prueba generados tiende a la generación de un número masivo de pruebas. La retro alimentación de la cobertura permite, al saber que funcionalidades se ejercitaron en cada prueba generada, recortar ciclos de simulación reduciendo el número de pruebas sin impactar en la calidad de la verificación.

Existen dos principales clases de cobertura, la cobertura funcional y la cobertura estructural. La cobertura funcional mide qué funcionalidades del diseño han sido ejercitadas. La misma

es subjetiva y difícil de medir ya que la definición de su espacio de cobertura y su posterior implementación deben ser realizados manualmente. La cobertura estructural o de código, indica qué tan bien ha sido ejercitada la implementación del diseño. Debido a que el diseño estará implementado en HDL, este tipo de cobertura también se puede denominar cobertura de código. Por ejemplo, se puede medir si todas las lineas de código HDL han sido ejecutadas. La medición de este último tipo de cobertura es más sencilla, ya que la mayoría de las herramientas EDA cuentan con esta funcionalidad ya implementada.

Lo siguiente que se necesita considerar es cómo generar los estímulos para lograr una cobertura satisfactoria.

2.1.2. Generación de Estímulos

Los estímulos requeridos para ejercitar completamente un diseño, es decir, hacer que exhiba todos los posibles comportamientos, es responsabilidad del aspecto de generación de estímulos del entorno de verificación.

Históricamente, un archivo formado por vectores binarios, escritos manualmente uno por uno, servía como estímulo de simulación. Con el tiempo, se introdujeron representaciones simbólicas de los vectores, como instrucciones en lenguaje ensamblador, junto a llamadas procedurales a rutinas de generación de vectores. Más tarde se desarrollaron generadores de vectores, comenzando con programas generadores de pruebas aleatorias (RTPG, sigla en inglés de *Random Test Pattern Generator*) [3], que evolucionaron a través de generadores de prueba basados en modelos (MBTG, sigla en inglés de *Model Based Test Generators*) [27] hasta llegar a los actuales generadores de estímulos aleatorios con restricciones mediante el uso de solucionadores de restricciones [34].

Las principales técnicas de generación de estímulos se pueden clasificar en las siguientes categorías:

- **Estímulos dirigidos:** se desarrollan secuencias de pruebas apuntadas a ejercitar cierta funcionalidad o cierto aspecto puntual del diseño. No es la técnica más eficiente ya que insume un tiempo considerable en su desarrollo debido a que hay que escribir puntualmente prueba por prueba. Adicionalmente, estas pruebas sólo podrán detectar errores esperados por el verificador. Muchas veces la ocurrencia de un error se debe a la sincronización de una serie de eventos paralelos, por lo que es muchas veces difícil de predecir y posteriormente generar una prueba para verificar estos casos. Los estímulos dirigidos son adecuados para verificar la funcionalidad básica del diseño, sobre todo en la etapa de *bring-up*.
- **Estímulos dirigidos con aleatoriedad:** este tipo de pruebas es similar al anterior, salvo que en este caso se hace aleatorio cierto aspecto de la misma. Por ejemplo, se puede

dirigir la prueba para estimular una funcionalidad aritmética particular de un diseño pero haciendo los valores de los operandos de entrada aleatorios.

- **Estímulos basados en modelos:** este tipo de pruebas se generan a partir de un modelo de alto nivel del diseño, facilitando la creación de los estímulos. Por ejemplo, a partir de una máquina de estados del diseño, la misma se puede ir recorriendo y extrayendo casos de pruebas relevantes. De esta forma se facilita y agiliza la extracción de pruebas. Las mismas pueden ser totalmente dirigidas o pueden incluir cierta aleatoriedad.

- **Estímulos aleatorios con restricciones:** en este tipo de pruebas se hacen aleatorios todos los aspectos relevantes de la prueba. Por ejemplo, el tipo de operación, los datos de entrada, los temporizados, etc. Lo fundamental de este tipo de pruebas es que la aleatorizacion no es totalmente libre, sino que los valores están acotados a cierto rango definido por restricciones. Por ejemplo, se podría restringir la aleatorización de las direcciones de memoria a cierto rango para que se produzca sólo en esa región, descartando valores inválidos o ya probados.

- **Estímulos guiados por la cobertura:** en este caso los estímulos son creados, ya sean de forma dirigida o aleatoria, con el objetivo de alcanzar cierto grado de cobertura. El enfoque de la verificación dirigida por la cobertura hace de la cobertura el núcleo que dirige el flujo de verificación. El espacio de cobertura se define inicialmente, y la cobertura es usada para medir la calidad de las pruebas aleatorias y para dirigir los recursos de verificación hacia los huecos del espacio de cobertura hasta que se obtenga un nivel de cobertura satisfactorio. Esto, en teoría, permite alcanzar verificación de alta calidad dentro un tiempo realista.

  Sin embargo, la verificación dirigida por la cobertura no es totalmente práctica si se la utiliza aisladamente. Entre las razones están las dificultades de la aplicación de la cobertura en las etapas tempranas del diseño debido, principalmente, a la inestabilidad del código RTL, la presión de asegurar el funcionamiento básico de la lógica, el conocimiento aún no adquirido sobre el diseño y la falta de implementación temprana de monitores de cobertura.

En un enfoque más integral de generación de estímulos, como el sugerido en [52], la verificación es dirigida primero por la detección de todos los errores posibles de RTL usando pruebas aleatorias y dirigidas que siguen lo detallado en el plan de verificación. Cuando estos métodos producen una caída en la detección de errores, la cobertura se comienza a medir gradualmente y los resultados dirigen a la verificación hacia la completitud de los huecos del espacio de cobertura.

2.1.3. Comprobación de la Respuesta

Una vez diseñada la infraestructura para generar los estímulos necesarios para alcanzar el grado de cobertura deseado se debe verificar que el DUV se comporte apropiadamente para cada una de las pruebas. Existen dos estrategias generales para comprobar si los resultados de una simulación conforman con las especificaciones funcionales. La primera está basada en modelos de referencia, la segunda está basada en *Checkers* distribuidos.

En el primer enfoque, un modelo de referencia corre en paralelo con el DUV y la salida de ambos es comparada. Este enfoque usualmente se aplica cuando hay disponible un buen modelo de referencia, por ejemplo en la verificación de un micro procesador o en las pruebas de regresión de algún cambio incremental. Este es un enfoque de caja negra, ya que se observa el comportamiento del dispositivo solo a través de sus entradas y salidas.

Los *Checkers* distribuidos, que son generalmente implementados utilizando aserciones [15], se consideran un enfoque de caja blanca, donde la confirmación de las especificaciones es observada por un conjunto de monitores de propiedades que deben mantenerse verdaderas durante la simulación. Estas propiedades comprueban ciertos comportamientos en señales críticas del diseño como registros de estado, contadores, líneas de buses, etc, donde el enfoque basado en modelo de referencia está limitado a observar el comportamiento sólo en las fronteras del diseño.

Generalmente, ambas estrategias son utilizadas al mismo tiempo.

2.2. Arquitecturas de los Testbenches

El *testbench*, o entorno de verificación, modela el universo para el DUV por lo que debe dar soporte a los aspectos fundamentales mencionados en la sección anterior. Teniendo presente esto, el entorno básico consta de los componentes para: generar estímulos, observar el comportamiento del diseño, comprobar la exactitud de la salida del DUV para los estímulos inyectados y para medir la cobertura funcional. En la Figura 2.1 se puede observar la arquitectura básica del entorno de verificación. En la misma, el área gris representa el *testbench* incluyendo todos sus componentes. Las flechas que entran al DUV representan los estímulos que el *testbench* le inyecta. Por otra parte, las flechas que salen del DUV y entran al *testbench* representan las respuestas del DUV para los estímulos que le son inyectados. También se puede observar que el componente de monitoreo mide tanto las señales de entrada del DUV como así su respuesta. De esta manera, el componente encargado de comprobar la respuesta puede verificar si el estímulo inyectado genera una respuesta correcta. Algunos autores incluyen también al DUV y al generador de señal de reloj como parte del *testbench*.

En general, un *testbench* es todo el código utilizado para crear, observar y comprobar el

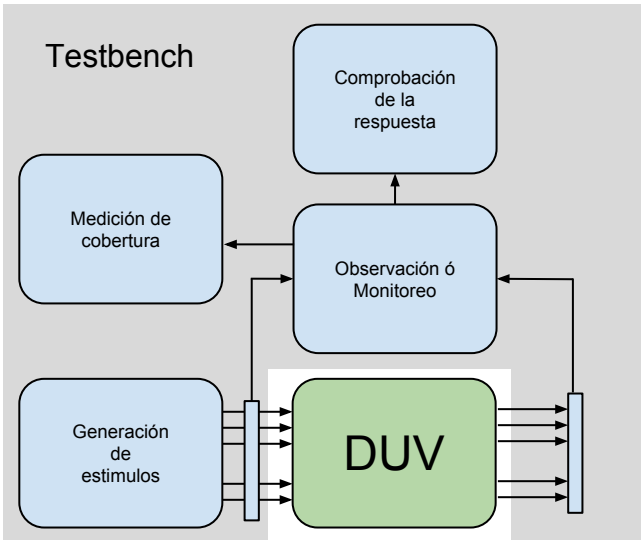

Figura 2.1: Arquitectura básica del entorno de verificación.

correcto funcionamiento de una secuencia pre determinada de estímulos inyectada a un diseño [51]. La secuencia pre determinada de estímulos puede ser generada de forma dirigida o por un método aleatorio.

El entorno de verificación es un sistema cerrado, es decir, el nivel más alto del *testbench* no tiene entradas ni salidas. Es en efecto un modelo del universo desde el punto de vista del DUV.

El ingeniero de verificación debe crear el código para los componentes o puede incorporar componentes de verificación de terceros (VIPs, sigla en inglés de *Verification Intellectual Property*). El *testbench* puede estar implementado en HDL, en algún lenguaje de verificación de alto nivel (HVL, sigla en inglés de *Hardware Verification Language*) como *SystemVerilog* (SV) [44] o en un lenguaje de propósito general como C o C++. Ocasionalmente, es necesario mezclar y conectar varios lenguajes, por ejemplo, un modelo de referencia provisto por un tercero escrito en C++ con un *testbench* escrito en SV.

2.2.1. Limitaciones del Testbench Monolítico

Históricamente, los *testbenches* eran programas monolíticos escritos en HDL donde las distintas responsabilidades del entorno, como la generación de estímulos y el monitoreo de las señales, no estaban separadas claramente y todas estaban implementadas en el mismo nivel de jerarquía.

Generalmente los *testbenches* generaban un archivo de salida con los resultados de la simulación, lo cual debía ser luego contrastado con los resultados esperados (modelo de referencia),

haciendo que el proceso sea lento y no se pueda automatizar (ver Figura 2.2). Una evolución natural de los *testbenches* fue la comprobación automática del comportamiento del DUV dentro del mismo entorno, aumentando la productividad, permitiendo correr grandes cantidades de pruebas sin intervención humana, sobre todo en las etapas de pruebas de regresión.

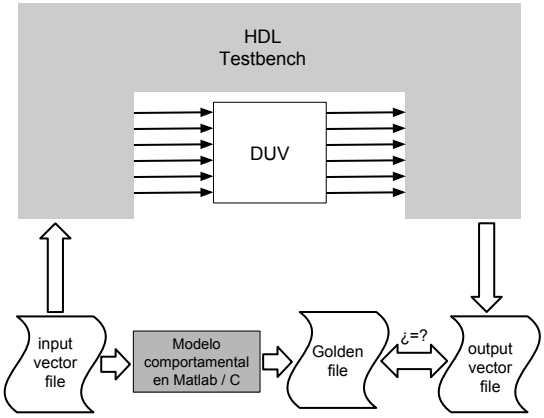

Figura 2.2: Históricamente los *testbenches* monolíticos no realizaban la comprobación automática del comportamiento del DUV por lo que se debía realizar esta tarea externamente.

A pesar de esto, los *testbenches* monolíticos seguían sufriendo serias limitaciones, sobre todo en cuestiones de re usabilidad y aumento del nivel de abstracción de las pruebas. Los *testbenches* estaban implementados en HDL, lenguajes pensados para la tarea de diseño y no de verificación, y la mayoría de la lógica del *testbench* se manejaba a bajo nivel, a nivel de señal, haciendo que se dedique mucho tiempo en implementar las pruebas en lugar de poder diseñarlas a un nivel mayor de abstracción delegando la responsabildiad de manejo de bajo nivel al *testbench*, facilitando de esta manera el proceso y enfocando el trabajo en definir las pruebas y no en cómo implementarlas.

En esta clase de *testbenches* no era posible intercambiar fácilmente los componentes de un *testbench* a otro, ya que la arquitectura plana hacía que existiera un fuerte acople y dependencia entre los componentes de un mismo *testbench*. La otra limitación importante era que esta jerarquía plana implicaba que si había que modificar algún detalle de bajo nivel, por ejemplo, el método de generación de señales, seguramente también había que modificar el código que hacía uso de estos métodos.

2.2.2. Testbench Estructurado

Los *testbenches* estructurados basados en clases, como los propuestos por la metodología OVM [38] y UVM (ver Capítulo 3), se basan en componentes independientes, con responsabilidades claramente separadas, que pueden ser fácilmente intercambiados y conectados entre si, logrando una alta reusabilidad. Este modelo de arquitectura permite construir *testbenches* para distintos proyectos reutilizando componentes.

Los entornos de verificación basados en simulación son casi siempre construidos basados en un conjunto de elementos estructurados, debido al poder y la flexibilidad que estos proveen. Como en cualquier proyecto complejo de software, hay muchas ventajas de usar un enfoque estructurado. Entre estas ventajas se encuentran [29]:

- Capacidad para manejar interacciones complejas,
- Reusabilidad de bloques de verificación y pruebas,
- Habilidad para escalar el proyecto,
- Facilidad para escribir pruebas,
- Claridad de la funcionalidad

El elemento genérico reusable de un entorno de verificación se define como *Transactor*, Agente o VIP. El *Transactor* facilita el proceso de verificación basado en simulación, implementando internamente todos los componentes que dan soporte a la generación de estímulos, monitoreo, medición de cobertura y comprobación de la respuesta del DUV permitiendo crear de forma eficiente pruebas expresadas en alto nivel.

Desde un punto de vista conceptual, el *Transactor* hace una transformación entre distintos niveles de abstracción, facilitando el manejo de la complejidad del proceso de verificación.

Hay varios elementos conceptuales que componen un *Transactor* (Figura 2.3). Hay una interfaz de bajo nivel mediante la cual este componente se comunica con un *Transactor* de menor jerarquía o con el mismo DUV. Hay otra interfaz de alto nivel o Interfaz de Programación de Aplicación (API, sigla en inglés de *Application Programming Interface*) mediante la cual se opera el *Transactor* comunicándole comandos e información de alto nivel. Esta interfaz también se utiliza para extraer datos de alto nivel. Finalmente, existe una transformación interna entre estas interfaces. Básicamente un *Transactor* se podría ver como un componente que recibe comandos de alto nivel y los convierte a un protocolo de menor nivel de abstracción. La interfaz de bajo nivel es generalmente un protocolo estándar. Si la interfaz de bajo nivel se conecta al DUV y la

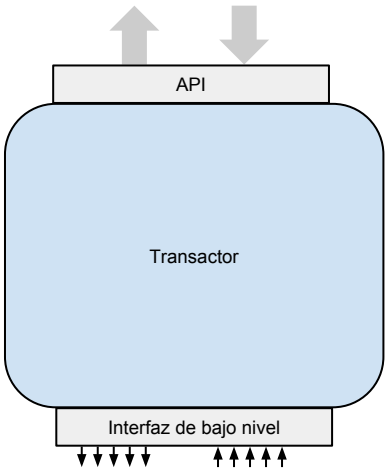

Figura 2.3: Esquemático del *Transactor*. Se puede observar la API que maneja un protocolo de comunicación de alto nivel, muchas veces implementado mediante llamado a métodos o en otros casos mediante puertos de comunicación estándar. La interfaz de bajo nivel maneja un protocolo de menor nivel de abstracción, en este caso particular las lineas finas modelan comunicación a nivel de señal, por lo que este *Transactor* se podría utilizar para estimular un DUV conectándole la interfaz de bajo nivel a los puertos de entrada salida.

misma trabaja a nivel de señal, dicho componente se denomina *Driver* o Modelo Funcional del Bus (BFM, sigla en inglés de *Bus Functional Model*).

Estructuralmente un *Transactor* básico (ver Figura 2.4) está compuesto por un componente encargado de convertir el protocolo de alto nivel en uno de bajo nivel, denominado *Driver*, y de un componente encargado de hacer la conversión inversa, es decir, transformar el protocolo de bajo nivel a protocolo de alto nivel, denominado Monitor. Este esquema básico se puede utilizar para facilitar la estimulación y observación del DUV, aumentando el nivel de abstracción, ya que todas las operaciones se realizan a través de los métodos de la API de alto nivel.

En un esquema básico las pruebas serán implementadas en alto nivel en función del API del *Transactor*, permitiendo al verificador enfocarse en las pruebas y no en cómo implementar los detalles de bajo nivel.

Más detalladamente, un Agente básico cuenta con una estructura para generar los estímulos, marcado con lineas punteadas en la Figura 2.5, que se puede separar en dos elementos, uno encargado de generar la secuencia de transacciones u operaciones, y otra encargada de convertir cada uno de los elementos de la secuencia en el respectivo protocolo de bajo nivel. En el caso básico el Generador creará secuencias de transacciones aleatorias con restricciones o correrá secuencias de transacciones predefinidas, enviando elemento a elemento al *Driver*, este convertirá

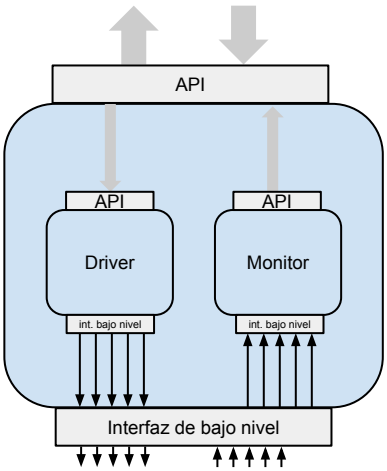

Figura 2.4: Estructura de un *Transactor* básico con un *Driver* para inyectar estímulos y un Monitor para observar el comportamiento del DUV

el elemento de alto nivel en su correspondiente estímulo a nivel de señales. El Monitor realizará la tarea inversa, construyendo elementos de alto nivel a partir de lo medido en las señales.

Como se explicó, los *Transactors* o Agentes están compuestos por subcomponentes como el Generador de Secuencias, el *Driver*, y el Monitor (ver Figura 2.5). A su vez, varios Agentes se pueden instanciar en un nivel de jerarquía superior denominado Entorno (ver Figura 2.8). Luego en una instancia aún más alta se encuentra la Prueba, la cual utiliza al Entorno, parametrizandolo y configurándolo para generar los estímulos deseados mediante el uso de los Agentes instanciados en el mismo. A su vez la Prueba será instanciada en un módulo Tope, en el cual se encuentra intanciado el DUV, el generador de señales de reloj y los respectivos conexionados. Esta jerarquía se puede ver gráficamente en la Figura 2.9.

Para poder medir la cobertura y comprobar la correcta respuesta del DUV para los estímulos inyectados, a la arquitectura básica del Agente se le pueden agregar los correspondientes componentes, el Medidor de Cobertura y el *Scorecobard*, como se muestra en la Figura 2.6. En este esquema, el Monitor es el encargado de medir los datos de bajo nivel y enviar la pertinente información en el correspondiente nivel de abstracción a los elementos encargados de medir la cobertura y comprobar la respuesta del DUV. De esta forma, las responsabilidades de los componentes quedan claramente separadas facilitando la reusabildiad de los mismos.

Para poder comprobar la correcta respuesta del sistema o para poder medir la cobertura global de sistemas más complejos, como así también durante la etapa de verificación de integración en las cuales el DUV esta conformado por varios bloques, sera necesario utilizar la información

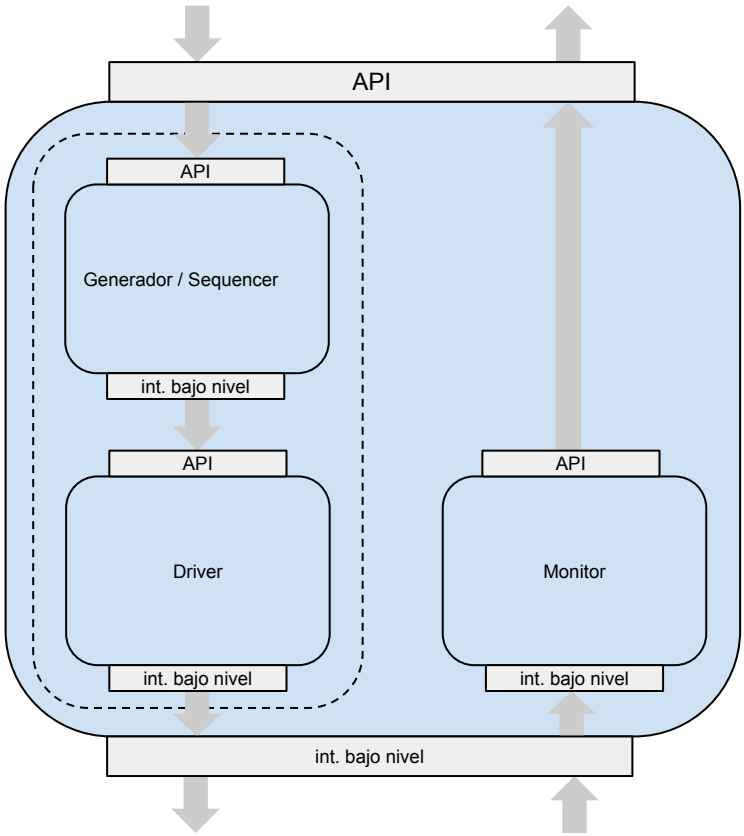

Figura 2.5: Agente Básico. La generación de estímulos, marcada con lineas punteadas, se puede implementar mediante el uso de dos componentes, el Generador de Secuencias y el *Driver*.

extraída por más de un Agente. En esta situación el Medidor de Cobertura y el *Scoreboard* se instanciarán en el Entorno recibiendo datos de alto nivel procedentes de los Agentes como se observa en la Figura 2.8.

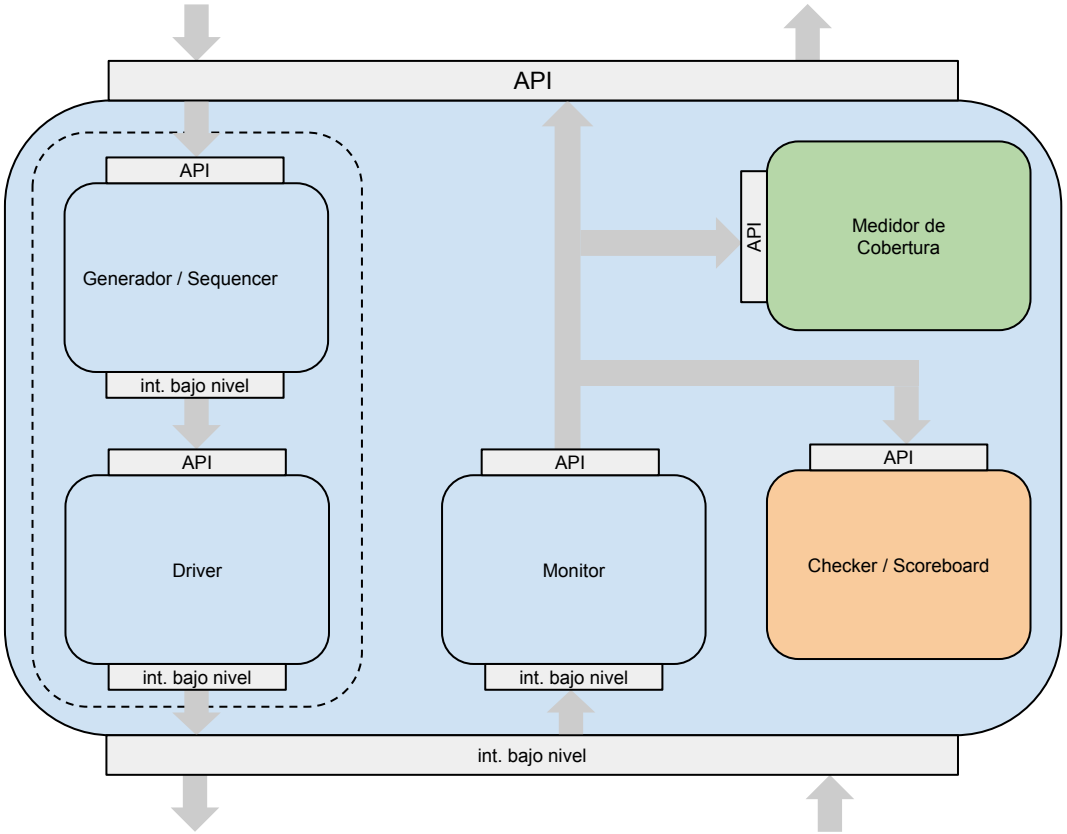

Figura 2.6: A la arquitectura del Agente básico se le agregan los componentes para medir la cobertura y comprobar la correcta respuesta del DUV para los estímulos inyectados. El Monitor es el encargado de sensar la información de bajo nivel y enviar los correspondientes datos en el correspondiente nivel de abstracción a los elementos de medición de cobertura y comprobación de la respuesta. De esta forma las responsabilidades de los componentes queda claramente separada facilitando la resusabildiad de los mismos.

Una interfaz genérica de alto nivel facilita la interconexión y reemplazo de los componentes facilitando la creación de *testbenches* conformados por componentes conectados y facilitando su posterior reemplazo y reutilización.

La arquitectura de *testbench* estructurado hasta aquí presentada también es modelada en forma de capas según [30] y [7]. En estas propuestas cada capa representa un nivel de abstracción superior al de la capa inferior, siendo la capa de menor nivel de abstracción la más cercana al

DUV. Cada capa agrupa los componentes del *testbench* hasta aquí presentados que lidian con cuestiones y problemáticas propias de ese nivel de abstracción (ver Figura 2.7). Por ejemplo, se define una capa Funcional de alto nivel a la cual pertenece el *Scoreboard* y el Generador de Secuencias. Los elementos de esta capa tratan con cuestiones a nivel funcional, como analizar si la funcionalidad del DUV es correcta y en generar secuencias de prueba a nivel funcional, despreciando detalles de implementación de bajo nivel. En una capa inferior denominada capa de Comandos se encuentra el *Driver* y el Monitor, los cuales se encargan de implementar los comandos utilizados por la capa Funcional. Los comandos implementados en la capa de Comandos consideran detalles de temporizado y cuestiones a nivel de señal. Una relación análoga se da entre el resto de las capas, donde la capa inferior le da soporte a la capa superior, permitiendo separar responsabilidades y tareas.

Este enfoque de capas promueve la adaptabilidad mediante el agrupamiento de componentes en capas de abstracción, permitiendo modificar una capa sin necesidad de modificar las otras. Adicionalmente, permite definir pruebas concentrándose en el diseño funcional de las mismas mediante el uso de los componentes de las capas superiores, delegando las cuestiones y detalles de implementación a los componentes de las capas inferiores.

El modelo de capas provee una abstracción que facilita el agrupamiento de las clases y la separación de sus responsabilidades, pero la arquitectura del *testbench* sigue estando conformada por los mismos componentes que hasta lo ahora presentado.

A continuación, siguiendo la arquitectura basada en *Transactors*, se describen con más detalle cada uno de los componentes que componen un *testbench* estructurado.

2.2.2.1. Generador de Estímulos

La generación de estímulos es, generalmente, responsabilidad de dos componentes activos pertenecientes al *Transactor*: el Generador, el cual genera secuencias de operaciones de alto nivel, y el *Driver*, el cual recibe estas operaciones de alto nivel y las convierte a estímulos a nivel de señal.

Como se dijo anteriormente, si la interfaz de bajo nivel del *Driver* maneja cuestiones a nivel de señal, entonces dicho componente también se lo denomina BFM. Como un *Driver*, un BFM provee una API que mapea un bus o puertos de entrada/salida a nivel de señales a una tarea de mayor nivel de abstracción.

El Generador se parametriza utilizando restricciones para generar secuencias de prueba aleatorias con restricciones, o puede tener secuencias dirigidas predefinidas que el mismo ejecuta.

Otra alternativa utilizada en lugar de los generadores de estímulos son los *Responders*, los cuales son componentes que responden a los eventos censados en la interfaz de bajo nivel en

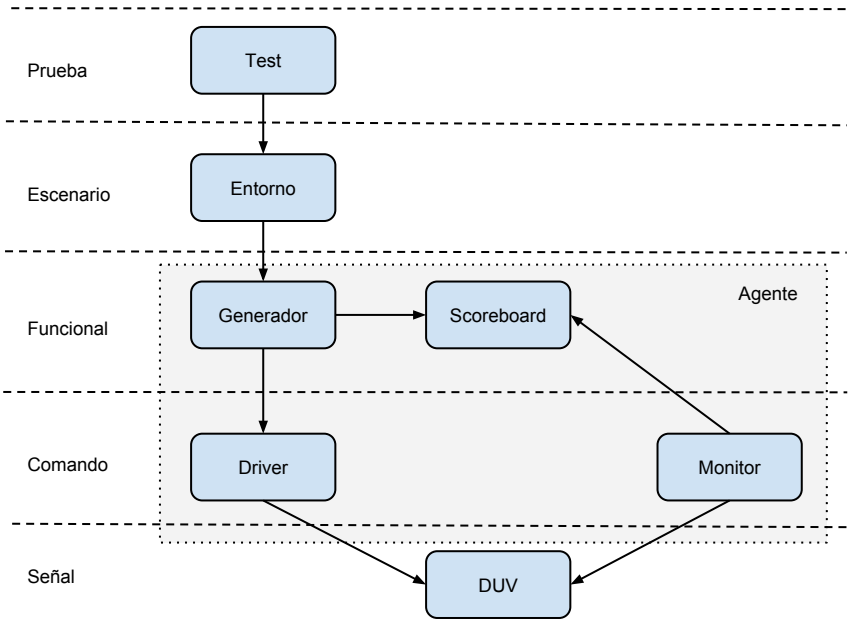

Figura 2.7: Esquema de modelo de capas. Se puede observar que el Agente engloba la capa Funcional y la capa de Comandos. También existe una capa de menor nivel de abstracción denominada capa de Señal, la cual muchas veces se unifica a la capa de Comandos en el *Driver* y en el Monitor. En otras ocasiones, por ejemplo, en un *testbench* de SV se podría colocar la Interfaz de SV en la capa de Señal, y el *Driver* y el Monitor que hacen uso de la misma para comunicarse con el DUV estarían en la capa de Comando.

lugar de crearlos, simulando de esta manera la respuesta de un dispositivo de hardware que esta conectado al DUV y el cual es necesario para realizar la verificación del DUV. Por ejemplo, el *Driver* inyecta comandos al DUV que este luego escribe en una memoria que puede estar representada mediante un *Responder*.

2.2.2.2. Monitor

Este componente pasivo que se encuentra en el *Transactor* se encarga de censar las señales de bajo nivel que entran y salen del DUV, extraer la información relevante, encapsularla y enviarla a los denominados componentes de análisis, esto son, los componentes encargados de medir la cobertura y los componentes encargados de comprobar la correctitud de las respuestas del DUV para los estímulos inyectados.

2.2.2.3. Scoreboard

Este componente, a veces llamado *Checker*, recibe los elementos de alto nivel procedentes de algún Monitor, los cuales son los mismos datos que entran y salen del DUV, y se encarga de comparar estos valores con los de un modelo de referencia comprobando de esta manera la funcionalidad del DUV desde una óptica de caja negra.

El modelo de referencia es, generalmente, un modelo funcional de alto nivel que representa el funcionamiento del DUV sin contemplar cuestiones de bajo nivel como el temporizado. El modelo de referencia también puede ser alguna versión anterior en HDL ya verificada del diseño, esto se realiza cuando se diseña una nueva versión del dispositivo, en el cual la funcionalidad es la misma, pero se realizaron mejoras en la implementación como modificaciones a la arquitectura interna para lograr, por ejemplo, menor tiempo de respuesta o menor consumo.

En algunas arquitecturas los *Scoreboards* se encuentran dentro del Monitor, pero esto reduce su reusabilidad.

Estos componentes también podrían ir dentro del Agente como se muestra en la Figura 2.6, pero muchas veces el *Scoreboard* puede necesitar información de distintos Agentes para poder comprobar el correcto funcionamiento del DUV, por lo que en estas situaciones el *Scoreboard* deberá colocarse en el Entorno como se muestra en la Figura 2.8. Por ejemplo, el DUV podría ser un *AHB to APB bridge*, el cual se comunica por un lado bajo el protocolo AHB y por el otro bajo el protocolo APB, haciendo en el medio la conversión pertinente entre los dos protocolos. Para verificar parte de su funcionamiento, un Agente AHB inyectará un mensaje AHB y enviará dicho mensaje al *Scoreboard*. Del otro lado, un Agente APB detectará el correspondiente mensaje en el puerto APB del DUV y enviará dicho mensaje también al *Scoreboard* para compararlo con su análogo AHB y así verificar si la transformación fue correcta.

2.2.2.4. Medidor de Cobertura Funcional

Este componente se encarga de medir la cobertura funcional. Recibe elementos provenientes de los Monitores de los respectivos Agentes. Generalmente se instancia fuera del Agente ya que la cobertura del sistema estará conformada por la información recibida proveniente de distintos Agentes.

Si la cobertura de un componente queda definida por la información provista por un mismo Agente, entonces el componente encargado de medir la cobertura funcional puede estar instanciado en el mismo Agente, como se muestra en la Figura 2.6.

2.2.2.5. Entorno

En muchos casos es necesario instanciar más de un Agente, por lo que se define una clase jerárquica donde se instancian estos componentes. La misma se define como Entorno o Escenario y se muestra en la Figura 2.8. En el Entorno además se instancian los componentes que necesitan datos de varios Agentes, como puede ser un *Scoreboard* o un medidor de cobertura global.

2.2.2.6. Prueba

La Prueba es la clase de mayor jerarquía del *testbench*, la misma es la encargada de instanciar al Entorno y configurar y parametrizar todos sus componentes. La Prueba podría, por ejemplo, crear un Entorno que contenga dos Agentes, y configurarlos apropiadamente, definiendo restricciones de aleatorización o eligiendo secuencias de prueba pre definidas para ejecutar en los respectivos Generadores, de forma que ambos Agentes trabajen de forma conjunta para estimular el DUV.

2.2.2.7. Tope

Todos los componentes presentados siguen la jerarquía presentada en la Figura 2.9, en la cual se destaca como el componente de más alta jerarquía el Tope. El Tope es el nivel de jerarquía común donde se instancia el DUV y el *testbench*, y ambos se conexionan, para que finalmente las pruebas se puedan llevar a cabo.

El Tope, que no forma parte del *tesbench*, tiene tres responsabilidades. La primera es instanciar el DUV, generar la señales de reloj e intanciar el *testbench* basado en clases (Prueba). La segunda responsabilidad es conexionar el DUV, generalmente escrito en HDL, con el *testbench* basado en clases. Esta tarea generalmente se realiza, como por ejemplo cuando se utiliza SV, mediante el uso de un componente denominado interfaz, la cual por un lado se conecta al DUV y al generador de señales de reloj, y luego se le pasa al *testbench* (Prueba) para que los distintos

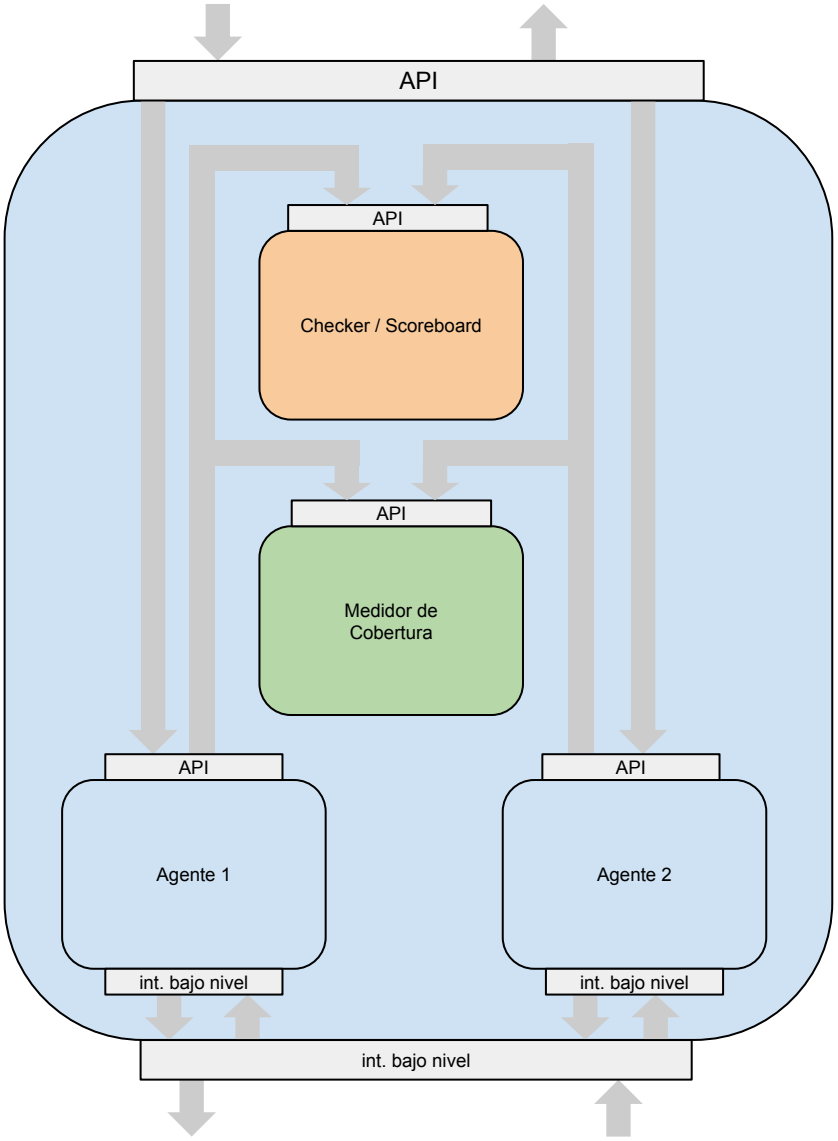

Figura 2.8: En el Entorno se pueden instanciar varios Agentes para que trabajen de forma coordinada. En algunos casos será necesario tener un Medidor de Cobertura global, o un *Scoreboard* global que necesitaran la información entregada por más de un Agente para poder realizar sus respectivas tareas.

componentes basados en clases puedan comunicarse con el DUV. La ultima responsabilidad del Tope es comenzar la ejecución del *testbench* (Prueba).

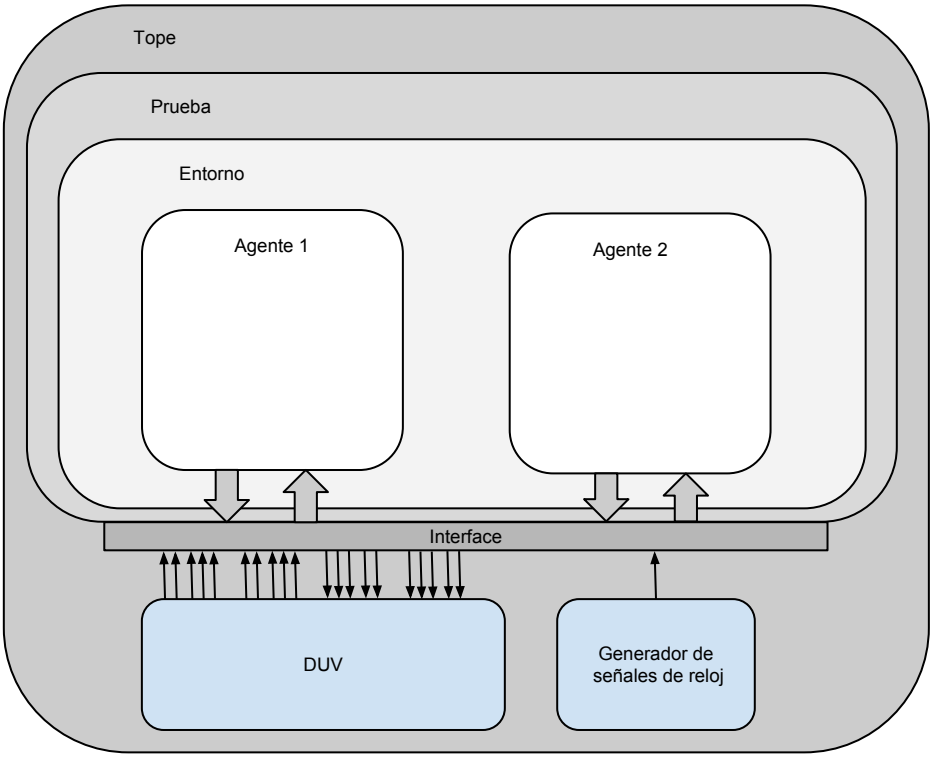

Figura 2.9: La jerarquía del *testebench* embebido en el tope junto al DUV y al generador de señales, todos conectados mediante una interfaz común.

2.2.3. Otras Consideraciones

En las siguientes subsecciones se describen dos puntos que no son directamente cuestiones de diseño de la arquitectura del *testbench*, pero son conceptos fundamentales de implementación que permiten darle soporte a dicho esquema estructurado, de forma de lograr reusabilidad real en la práctica.

2.2.3.1. Comunicación mediante Transacciones

Para facilitar la reusabilidad muchas arquitecturas recurren a una estandarización de la comunicación entre sus componentes mediante el uso de canales o puertos estándar de comunicación. Esto se realiza mediante el intercambio de datos modelados a alto nivel y una estandarización de

la API de comunicación. Este tipo de comunicación se implementa mediante pasaje de mensajes entre los procesos o hilos que representan la ejecución de cada componente del *testbench*.

Una estandarización del API de comunicación utilizada ampliamente es la provista por OSCI *Transaction Level Modeling* (TLM) 1.0 [37] la cual provee un estándar para la comunicación mediante el modelo de pasaje de mensajes. En este modelo los objetos, transacciones en este caso, son pasados entre los componentes del *tesbench*. Este estándar es implementado por distintas metodologías de verificación, como por ejemplo UVM [2], para la comunicación de sus componentes.

La primer ventaja de esta metodología de comunicación es poder reemplazar fácilmente un componente del *testbench* por otro, ya que todos los componentes se comunican de la misma manera. Esta estandarización de las APIs favorece la reusabilidad de los componentes. Esto se logra mediante el uso de puertos de entrada y puertos de salida implementados en cada componente que luego son conectados desde un elemento de mayor jerarquía el cual instancia estos componentes y luego los conecta.

La otra ventaja de este enfoque es permitir el aumento del nivel de abstracción, ya que los datos que se intercambian entre los componentes son datos de alto nivel que encapsulan la información relevante para ese nivel de abstracción. Por ejemplo, cada transacción que envíe un Generador de Estímulos a un *Driver* podrá ser un elemento que representa una operación de escritura sobre el DUV, sin información de temporizado o detalles de bajo nivel. Luego el *Driver* se encargara de convertir ese ítem en los correspondientes estímulos de señales considerando los detalles de temporizado y demás cuestiones de bajo nivel.

2.2.3.2. Flexibilidad provista por la Programación Orientada a Objetos

Los HVL modernos, como SV, pertenecen al paradigma de la Programación Orientada a Objetos (POO) permitiendo aplicar todas sus ventajas, principalmente las relacionadas a la reusabilidad, al desarrollo de *testbenches*.

La técnica de herencia aplicada a un *testbench* permite extender y especializar rápida y eficientemente un componente de verificación modelado como una clase. Por ejemplo, como se muestra en la Figura 2.10, cada componente de un *testbench* puede heredar de una clase componente la cual provee ciertas funcionalidades básicas ya implementadas, como por ejemplo los puertos de comunicación. Luego una clase *Driver* Genérico puede heredar de la clase componente agregando cierta funcionalidad como la extracción de elementos del puerto, ciertos mensajes de información, y algunos métodos que deberán ser implementados por las clases hijas (los denominados métodos virtuales de la POO). Luego se puede implementar un *Driver* Genérico de AHB y otro de AXI, los cuales definan ciertas constantes comunes y métodos genéricos. Finalmente

se podrían tener dos clases hijas del *Driver* Genérico AHB, una que implemente el protocolo AHB-Lite y otra el protocolo AHB. Luego un *Driver* podrá ser reemplazado fácilmente por otro, con mínimo impacto en el resto del *testbench*. Por ejemplo, si un DUV que inicialmente se comunica mediante el protocolo AHB se reutiliza en otro proyecto modificando su interfaz para comunicarse mediante el protocolo AXI, en el *testbench* solo sera necesario remplazar el *Driver* AHB por el *Driver* AXI. En la Figura los bloques punteados representan clases abstractas que no pueden ser instanciadas directamente, sino que deben ser heredadas e implementadas.

La herencia también facilita la creación de nuevas pruebas o secuencia de pruebas, mediante la simple extension de pruebas anteriores.

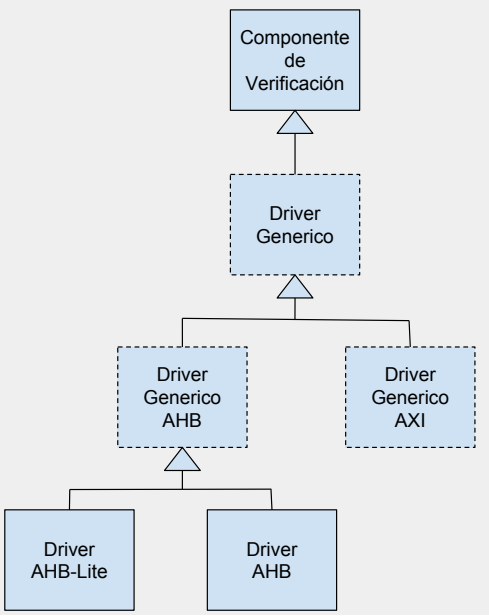

Figura 2.10: Diagrama de clases en el cual se observan las distintas variaciones de *Drivers* mediante la utilización de la técnica de herencia.

El Método *Factory* es un Patrón de Diseño [19] de la POO el cual define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué tipo de clase instanciar, permitiendo que una clase delegue en sus subclases la creación de objetos. El Método *Factory* permite escribir aplicaciones que son más flexibles respecto de los tipos a utilizar, difiriendo la creación de las instancias en el sistema a subclases que pueden ser extendidas a medida que evoluciona el sistema. En el proceso de verificación funcional es frecuentemente necesario introducir variaciones a las clases que implementan el *testbench*. Por ejemplo, en muchas pruebas puede ser necesario derivar los elementos de alto nivel que representan las transacciones a partir

de una definición genérica y agregar más restricciones o campos, o puede ser necesario usar una nueva clase derivada en todo el *testbench* o solo en un componente especifico, o tal vez puede ser necesario modificar la forma en que los datos son enviados al DUV derivando un nuevo *Driver*. El Método *Factory* permite sustituir los componentes de verificación sin tener que también proveer una versión derivada del componente padre.

El uso del mecanismo de Método *Factory* permite elegir desde la clase Prueba el tipo de objeto que sera instanciado o sobrescrito en algún lugar del *testbench*, aunque un dado componente, transacción o secuencia puede solo ser sobrescrito con uno que extienda la clase de objeto original. Este es uno de los principales mecanismos a través del cual un componente reusable de verificación puede ser customizado al entorno actual, permitiendo que el desarrollo de la infraestructura del *testbench* sea en gran medida independiente del desarrollo de los *tests* que usan el *testbench*.

2.3. Modelo de Cobertura Funcional

Los modelos de cobertura funcional consisten en métricas que se refieren directamente a la computación realizada por un sistema en lugar de su estructura o implementación y son, típicamente, específicas para una familia de diseños. Las mismas tienen, comúnmente, la forma de una lista cuidadosamente construida de escenarios propensos a errores y fragmentos de funcionalidad extraídas, inicialmente, analizando las especificaciones del diseño. Luego, durante la simulación, cada uno de estos escenarios y funcionalidades deben ser ejercitados. Algunas funcionalidades complejas, como ciertas secuencias de instrucciones de un procesador o ciertas secuencias de transacciones de un bus pueden necesitar ser ejercitadas varias veces en diferentes combinaciones. Luego, las herramientas de medición de cobertura reportan la cantidad de veces que se ejercitó cada secuencia o combinación de secuencias.

Idealmente, una especificación debe encapsular sólo las funcionalidades del diseño y ningún detalle de implementación. De esta manera, la especificación provee una lista de comportamientos ejercitables obtenidos independientemente de la estructura del diseño, haciendo de ésta la fuente ideal para derivar inicialmente el modelo de cobertura funcional.

Los escenarios propensos a errores son, comúnmente, especificados manualmente. Para cada caso identificado se construye un medidor de cobertura utilizando los constructores provistos por los lenguajes de verificación de hardware. Durante la simulación, el estado de los medidores de cobertura indican si los casos y escenarios especificados fueron ejercitados.

Las métricas que se refieren a funcionalidades se pueden denominar como puntos de cobertura de tipo instantáneos o temporales. Las puntos de cobertura instantáneos son condiciones sobre los valores de las variables en un ciclo de reloj determinado. Los puntos de cobertura temporales

involucran condiciones que abarcan varios ciclos de reloj (no necesariamente consecutivos), por ejemplo, una secuencia de 6 ciclos de reloj que contenga un patrón de tipo instrucción, interrupción, instrucción para un dado procesador. Los monitores de cobertura son definidos utilizando lenguaje de verificación de hardware que tienen operadores temporales (mecanismos para especificar comportamientos en ciclos de reloj pasados y futuros) y soportan operaciones aritméticas y lógicas. El código que implementa el medidor de cobertura puede contar la cantidad de veces que se pasó por un estado, se ejercitó un evento o las combinaciones de estado-comando, o secuencia de eventos, y esta información puede luego utilizarse para dirigir futuras simulaciones.

Las métricas basadas en grafos de transición de estados también pueden considerarse métricas de cobertura funcional, donde cada estado de una Máquina de Estados Finitos (FSM, sigla en inglés de *Finite State Machine*) especifica un punto de cobertura de tipo instantáneo, y cada camino que requiere de cobertura constituye una métrica de cobertura de tipo temporal.

Para los diseños que conforman las interfaces estándar de protocolos de comunicación existen *suites* de pruebas. Sin embargo, para la mayoría de los diseños, el verificador debe entender el diseño lo suficiente como para definir por su cuenta métricas funcionales efectivas. Las métricas de cobertura útiles son el producto de experiencia acumulada con familias de diseños. Como tal, las métricas funcionales son altamente dependientes del diseño, e implementarlas requiere de un significativo esfuerzo manual y experiencia, sobre todo en los diseños actuales, los cuales son parametrizables en diferentes configuraciones, aumentando considerablemente los casos de prueba necesarios.

La generación de pruebas dirigidas a cubrir métricas funcionales es más compleja que la generación de pruebas para métricas de cobertura estructural. Debido a que las métricas funcionales pueden abarcar varios ciclos de reloj y no están necesariamente localizadas en el código HDL, la generación automática de pruebas es difícil. La dificultad aumenta si la métrica de cobertura involucra cierta intercalación, o restricción de temporizado entre eventos de diferentes módulos. Aunque este enfoque detecta mayor cantidad de errores difíciles de encontrar que otros enfoques, el esfuerzo involucrado en generar pruebas es alto.

Las métricas de cobertura funcional se enfocan en probables fuentes de error determinado por la experiencia o la intuición. A diferencia de otras métricas, la completitud de la cobertura funcional provee una lista de casos asegurando el comportamiento del diseño. Para diseños complejos, un *suite* de métricas de cobertura funcional es indispensable para llevar a cabo el proceso de verificación, y el mismo puede ser transferido a nuevas generaciones de diseños.

2.3.1. Modelos de Cobertura Funcional basados en Grafos de Transición de Estados

Los modelos de cobertura funcional tradicionales basados en listas de funcionalidades sufren de limitaciones para cuantificar y expresar requerimientos para comportamientos secuenciales.

Las métricas de cobertura basadas en código están definidas en función de representaciones estructurales estáticas del diseño, por lo tanto su habilidad para cuantificar y plantear requerimientos para comportamientos secuenciales es limitada. Las métricas basadas en grafos de transición de estados son más poderosas para esta cuestión. Estas métricas requieren de cobertura de estado, transición o camino sobre las representaciones del diseño en maquinas de estados finito. Algunas cuestiones relacionadas al control del diseño se representan mejor utilizando una FSM o una colección de FSM que interaccionan entre sí. En este último caso tiene sentido usar métricas definidas para múltiples FSM, por ejemplo, la métrica de pares de arcos requiere que se ejerciten todos los pares de transiciones para cada par de FSM.

Debido a que las descripciones de FSM para sistemas completos son demasiado grandes, estas métricas deben ser definidas sobre FSM más pequeñas, más abstractas. En este contexto se pueden definir dos categorías amplias de FSM:

- FSM escritas a mano que capturan el comportamiento del diseño a un nivel alto de abstracción.
- FSM extraídas automáticamente de la descripción del diseño. Típicamente, después de que un conjunto de variables de estado es seleccionada, el diseño es proyectado sobre este conjunto para obtener una FSM abstracta.

Las métricas en la primer categoría son menos dependientes de los detalles de implementación y encapsulan la intención del diseño más concisamente. Sin embargo construir la FSM abstracta y mantenerla a medida que el diseño evoluciona requiere de un esfuerzo considerable. Más aun, no hay garantías de que la implementación se ajuste correctamente al modelo de alto nivel. A pesar de estas desventajas, especificar el sistema desde un punto de vista alternativo es un método efectivo para exponer los defectos de diseño. La experiencia ha demostrado que utilizar casos de pruebas apuntados a incrementar este tipo de cobertura ha detectado muchos errores difíciles de encontrar [6, 24].

Las variables de estado de las FSM abstractas utilizadas para métricas en la segunda categoría pueden ser seleccionadas manualmente o mediante técnicas heurísticas [42].

Seleccionar FSM abstractas requiere de un compromiso entre la cantidad de información que se vuelca en la FSM y la facilidad de utilizar la información de cobertura obtenida. Los beneficios relativos de la elección de la FSM y de las métricas de cobertura definidas sobre la

misma son dependientes del diseño. Aumentar la cantidad de detalles en la FSM aumenta la precision de la métrica de cobertura, pero hace que sea más difícil interpretar la información de cobertura adquirida. Si la FSM es grande, lograr cobertura alta con respecto a las métricas más sofisticadas como cobertura de caminos es más difícil. Sin embargo, los diseñadores pueden necesitar considerar caminos en lugar de sólo estados o transiciones para asegurar que secuencias de comportamiento importantes sean ejercitadas.

El mayor desafío de las métricas basadas en grafos de transición de estados consiste en escribir pruebas dirigidas por la cobertura. Determinar si ciertos estados, transiciones, o caminos pueden ser cubiertos puede ser difícil. Las variables de estado de una FSM pueden ser difíciles de extraer del diseño, y lograr la cobertura deseada puede requerir satisfacer varias restricciones secuenciales. Más aun, inspeccionar y evaluar la información de cobertura puede ser difícil, sobre todo si la FSM fue extraída automáticamente. Algunos enfoques automáticos utilizan técnicas de prueba secuenciales [31]. Otros utilizan una correspondencia entre la información de cobertura y los estímulos de entrada utilizando técnicas de búsqueda de patrones (*Pattern Matching* en inglés) en corridas de simulación previas [42]. La capacidad de los métodos automáticos es a menudo insuficiente para manejar la generación de pruebas dirigidas por la cobertura en casos prácticos, en donde el verificador puede necesitar entender el diseño completo para poder generar los estímulos necesarios. Sin embargo, las métricas basadas en grafos de transición de estados son muy valiosas para identificar casos raros, fragmentos de ejecución propensos a errores e interacciones de FSM que fueron pasados por alto durante la simulación. Últimamente, la elección cuidadosa de FSM abstractas puede aliviar muchos de los problemas mencionados [46].

Capítulo 3

Experimentación con la Arquitectura del Testbench

En el presente capítulo se describen inicialmente las propiedades deseadas de un *testbench* basado en al arquitectura presentada en el capítulo anterior. Luego, teniendo en cuenta esto, se analiza primero de forma teórica la arquitectura de *testbench* propuesta por UVM y las facilidades que ésta metodología provee en términos de implementación. Luego, se justifica su elección, se experimenta y analiza de forma práctica con la misma mediante el diseño e implementación de *testbenches* para dos casos de estudio, modificando y ajustando las arquitecturas a las diferentes necesidades y requerimientos de cada tipo de diseño que se desea verificar.

Una versión reducida del trabajo desarrollado en el presente capítulo fue publicado en la 8^a *Conferencia de Micro-Nanoelectrónica, Tecnología y Aplicaciones* (CAMTA) del año 2014 bajo el título *UVM based testbench architecture for unit verification* [16].

3.1. Motivación

Una de las principales propiedades con la que debe contar un *testbench* es la de reusabilidad. El mismo se debe poder reutilizar de forma horizontal, es decir en diferentes proyectos, y de forma vertical, es decir a través de diferentes niveles del mismo proyecto, desde la verificación a nivel de bloque, pasando por la verificación a nivel de integración, hasta la verificación a nivel de sistema. Para lograr esta reusabilidad, los *testbenches* deben estar implementados utilizando componentes intercambiables, por ejemplo, el *Driver* de un protocolo especifico, como AHB-Lite, podrá ser utilizado para estimular un diseño A con funcionalidad f_a y luego reutilizado para estimular un diseño B con funcionalidad f_b. Cada *testbench* tendrá sus propias secuencias de pruebas y sus respectivos modelos de referencia, pero el *Driver* será reutilizado de un entorno

a otro de forma transparente. Para poder lograr reusabilidad dentro de la industria y favorecer la creación de mercados de VIP surgen las metodologías estándar de verificación, las cuales brindan un *framework* y establecen un conjunto de prácticas de cómo utilizar dicho *framework* para generar componentes reusables.

La otra propiedad fundamental que debe tener un *testbench* es la capacidad para aumentar el nivel de abstracción, separando principalmente los detalles de bajo nivel relacionados a los temporizados y sincronizados de señales de la funcionalidad que se desea probar.

Una propiedad que se desprende de las dos anteriores es la modularización o separación de las responsabilidades dentro de la arquitectura. Es claro que cada componente reusable de un *testbench* encapsula cierta responsabilidad, pero la separación de responsabilidades que aquí se señala hace referencia a la separación del *testbench* en una parte que se puede denominar infraestructura de servicio y otra que es la Prueba, que se implementa utilizando las facilidades provistas por la infraestructura de servicio, facilitando su implementación, ejecución y control. Por lo tanto, un *testbench* tiene una parte que consiste de la infraestructura que facilita la implementación de las Pruebas (Agentes y todos los componentes que este contiene) y otra parte que define cuales serán estas Pruebas. La infraestructura puede ser implementada por una parte y las Pruebas por otra; ambas se pueden implementar de forma paralela por grupos separados de ingenieros. La infraestructura puede ser reutilizada para correr distintas Pruebas, como habitualmente se hace, y un mismo grupo de Pruebas se puede reutilizar para probar un diseño similar mediante otro *testbench*, que por ejemplo utilice un *Driver* distinto.

Para poder diseñar e implementar una infraestructura de verificación con estas propiedades se recurre a la metodología UVM con el fin de implementar un *testbench* con tecnología y metodología del estado del arte, el cual resulte reusable entre distintos proyectos y distintos niveles de integración cumpliendo con las propiedades presentadas.

3.2. Framework UVM

UVM [2] es el último miembro de una familia de metodologías y sus bibliotecas de clases base asociadas para utilizar SV para la verificación funcional de hardware digital.

Las metodologías basadas en SV desempeñan un papel importante en el desarrollo de entornos de verificación. Definen convenciones para la estructura y la dinámica de un *testbench*, capturan las mejores prácticas y evitan la necesidad de reinventar los mecanismos necesarios para utilizar las clases de SV para construir entornos de verificación.

Esta estandarización permite a los usuarios implementar entornos de verificación que son portables y compatibles, y ayuda a explotar las habilidades especificas de los ingenieros sobre el dominio de aplicación de un proyecto a otro. Ampliamente promocionadas en la industria, las

metodologías de SV catalizan el rápido intercambio de conocimientos y técnicas probadas entre la base de usuarios y fomenta el crecimiento de un mercado de VIPs.

En la jerga de UVM, los Componentes de Verificación (VC, sigla en inglés de *Verification Component*) son entornos de verificación encapsulados, configurables, listos para usar en un protocolo de interfaz, un diseño de submódulo o un sistema completo. Cada VC sigue una arquitectura consistente y esta compuesto de un conjunto completo de elementos para estimular, verificar, y recolectar información de cobertura funcional para una protocolo o diseño específico. Los mismos VCs pueden ser reutilizados horizontalmente a través de proyectos o verticalmente desde el nivel verificación de bloque hasta la verificación a nivel de integración o a nivel de sistema. La definición de VCs es vaga en la bibliografía, pero serían los denominados Entornos presentados en el capítulo anterior, aunque los Agentes también podrían denominarse VC.

UVM ofrece una amplia biblioteca de clases base que brinda soporte para el desarrollo de *testbenches* compuestos de VC reutilizables. UVM esta explícitamente orientado a simulación, con la intención de realizar verificación aleatoria con restricciones (CRV, sigla en inglés de *Constrained Random Verification*) dirigida por la cobertura (CDV, sigla en inglés de *Coverage Driven Verification*), pero también se puede utilizar junto a la verificación basada en aserciones [11], con aceleración por hardware [40] o emulación. Además de una rica biblioteca de clases base, proporciona una metodología de referencia con mejores prácticas. Es totalmente compatible con los principales proveedores de herramientas y es mantenida por un reconocido organismo de la industria llamado *Accellera*.

Le herencia de UVM incluye AVM de *Mentor Graphics*, OVM de *Mentor Graphics* y *Cadence Design Systems*, eRM de *Verisity* y VMM-RAL de *Synopsys*. Estas librerías de metodologías previas proveyeron un rico legado sobre el cual UVM se construyó. Más notablemente, OVM-2.1.1 fue el punto de partida para UVM, el código fuente que le dio inicio al desarrollo.

3.2.1. La Biblioteca de Clases

La biblioteca de clases de UVM [1], (BCL, sigla en inglés de *Base Class Library*) proporciona los elementos básicos necesarios para desarrollar VCs y entornos de verificación bien construidos y reutilizables. La biblioteca consta de clases base y de utilidades de infraestructura. Las clases en la jerarquía UVM recaen en gran medida en dos categorías distintas: componentes y datos. La jerarquía de clases de componente se deriva de *uvm_component* y está destinada a modelar partes estructurales permanentes del *testbench*, tales como monitores y *drivers*. Las clases de datos se derivan de *uvm_sequence_item* y están destinadas a modelar estímulos y transacciones. Las utilidades de infraestructura proporcionan varias utilidades para simplificar el desarrollo, la gestión y el uso del entorno verificación, tales como control de ejecución mediante fases, métodos

de configuración, métodos de *Factory*, facilidades para la interconexión de los componentes y control de mensajes de informe.

3.2.1.1. Componentes

Todos los componentes de la infraestructura en un entorno de UVM se derivan de la clase *uvm_component* y conforman una jerarquía que incluye: *Sequencer, Driver*, Monitor, *Coverage Collector, Scoreboard, Environment* y *Test*. Dichos componentes son análogos a los presentados en el capítulo anterior y conforman una arquitectura similar.

Cada componente en el *testbench* de un usuario se deriva de su correspondiente componente en la biblioteca de clases de UVM. Usando estos elementos basados en clases se aumenta la legibilidad del código, ya que el rol de cada componente está predeterminado por su clase padre.

En los siguientes apartados se presentan brevemente los componentes que conforman un *testbench* UVM, ya que los mismos son análogos a los presentados en el capítulo anterior.

Sequencer

El *Sequencer* (Generador) es la entidad que ejecuta código de generación de estímulos y envía la secuencia de elementos hacia el Driver.

Driver

El *Driver* es responsable de estimular el DUV a través su interfaz. Siempre se encuentra conectado a un *Sequencer* al cual le solicita elementos de secuencia a través de un puerto de comunicación estándar. Mapea los elementos de secuencia al formato de nivel de señal requerido por la interfaz del DUV.

Monitor

El Monitor sensa el tráfico a nivel de señales que va hacia y desde el DUV, a partir de lo cual ensambla elementos que distribuye al resto del entorno de verificación a través de uno o más puertos de comunicación estándar.

Coverage Collector

Esta entidad (Medidor de Cobertura Funcional) mide el avance del proceso de verificación al registrar el tipo de pruebas y los resultados que se producen sobre un modelo de cobertura.

Scoreboard

Este componente analiza la correctitud funcional de las salidas de las pruebas comparándolas con los resultados de un modelo de referencia.

Tanto el *Coverage Collector* como el *Scoreboard* dependen de los datos proporcionados por algún Monitor, por lo que es mejor mantenerlo separado como una entidad aparte para favorecer la reutilización. Los componentes Monitor suelen ser específicos para determinados protocolos y sensibles a parámetros a nivel de señal, pero son independientes de la aplicación. En contraste, el código de los *Coverage Collectors* y los *Scoreboards* suelen ser muy específicos a la aplicación y menos afectados por los protocolos y temporizados de las interfaces.

Agent

El Agente es un contenedor abstracto que encapsula un *Driver*, un *Sequencer* y un Monitor. Por lo general son configurables en modos activo o pasivo. En el modo activo se utilizan para emular dispositivos e inyectar transacciones de acuerdo con las directivas del test, mientras que en el modo pasivo sólo se utilizan para sensar la actividad DUV.

Environment

El *Environment* (Entorno) es la entidad que ensambla la estructura del *testbench*. Crea una instancia de uno o más Agentes, etc, así como otros componentes tales como un monitor a nivel de bus para realizar la medición de cobertura y verificación a nivel de sistema. Tiene los parámetros de configuración que permiten reestructurar y volver a utilizarlo para diferentes escenarios.

Test

El *Test* (Prueba) es la entidad que hace uso del *Environment*. Se lo utiliza para:

- Controlar la generación del *Environment* configurando el comportamiento de sus VCs.
- Determinar el comportamiento dinámico del proceso de verificación iniciando secuencias específicas en *Sequencers* determinados.
- Extender y adaptar el *tesbench* reemplazando los componentes de la estructura o reemplazando los elementos de datos y secuencias.

Cada test sólo contendrá algunas modificaciones y/o parametrizaciones sobre el *Environment* de verificación que distinguen al *Test* del usuario de la situación por defecto.

3.2.1.2. Datos

En la metodología UVM los componentes presentados se comunican entre si mediante transacciones de alto nivel, creando datos y enviándolos a los otros componentes. Dicho dominio de datos dinámicos está representado por:

Sequence Item

Son los objetos de datos básicos que son pasados entre los componentes. En contraste con las señales de VHDL y los cables de *Verilog*, los *Sequence Items* representan comunicación a un nivel de abstracción mayor.

Sequence

Se ensamblan a partir de *Sequence Items* y son utilizados para construir conjuntos realistas de estímulos. Una secuencia podría generar un conjunto predeterminado de transacciones, un conjunto totalmente aleatorio de transacciones o un conjunto de transacciones intermedias, es decir, con con algunas transacciones predeterminadas y otras aleatorias. Las *Sequences* pueden ejecutar otras *Sequences* y pueden ser configuradas en capas de tal manera que las *Sequences* de más alto nivel envíen transacciones a secuencias de más bajo nivel conformando una pila de protocolos.

Los objetos de tipo componente son creados en tiempo cero de simulación, después que la jerarquía del DUV ha sido elaborada y las variables estáticas han sido inicializadas, pero antes que el modelo de simulación RTL comience a consumir tiempo.

Los objetos de tipo dato, en contraste con los objetos de tipo componente, pueden ser creados en cualquier momento. A diferencia del árbol de instancias de componentes, los objetos de tipo dato no son automáticamente organizados en una estructura predeterminada.

3.2.2. Utilidades

Las piezas claves de las utilidades de UVM que permiten implementar un *testbench* estructurado reusable de forma eficiente son: fases y control de ejecución, métodos de configuración y métodos de *Factory*. Adicionalmente UVM provee una utilidad de control jerárquico de reporte de mensajes.

3.2.2.1. Fases

La BCL fuerza un conjunto de convenciones relacionadas al ciclo de vida del *testbench* conocido como fases. Cada fase es ejecutada por la infraestructura de UVM, un método correspondiente

a cada fase es automáticamente invocado en cada uno de los objetos de tipo componente. Los usuarios no necesitan preocuparse por este mecanismo, sólo deben sobrescribir los métodos de las fases en sus propias clases derivadas para proveer funcionalidad personalizada de forma que sus componentes de verificación se comporten como lo desean.

Cada componente implementa el mismo conjunto de fases, que son ejecutadas en un orden predefinido durante la simulación a fin de sincronizar correctamente el comportamiento de los componentes.

Las fases se pueden clasificar de la siguiente manera: fase de construcción de la jerarquía de componentes, fase de conexión de los puertos de los componentes y fase de ejecución de la simulación. En la fase de construcción un componente crea sus subcomponentes, por ejemplo el *Agent* crea el *Sequencer* y el *Driver*. En la fase de conexionado conecta sus subcomponentes usando los puertos de comunicación estándar. Por ejemplo, el *Agent* conecta su *Sequencer* con su *Driver*. En la fase de ejecución el *Test* corre la sequencia en el *Sequencer* y el *Driver* adquiere *Sequence Items* del *Sequencer* para luego convertirlos en movimiento de señales para estimular el DUV.

3.2.2.2. UVM Factory

La BCL incorpora un mecanismo de *Factory* ya implementado que permite:

- Controlar la asignación de objetos en todo el entorno o en objetos específicos.
- Sustituir los datos usados para los estímulos como también los componentes de la infraestructura.

En la practica, el mecanismo de *Factory* de UVM permite a un *Test* hacer, en tiempo de ejecución, mediante el uso de métodos de sobreescritura, sustituciones de un tipo de componente o transacción en el mecanismo de *Factory*. Todo esto, antes de que se construya el *Environment* del *testbench*. Luego, durante la fase de construcción, el *Environment* será creado con los tipos de componentes sustituidos por el *Test* en el mecanismo de *Factory* sin necesidad de modificar el código del *Environment* ni el de sus componentes.

Por ejemplo, se podría desear hacer pruebas con dos tipos de transacciones, *trans1* y *trans2*, donde *trans2* es una extension de *trans1*, pero incluye datos de aleatorización adicionales. Debido a que el *Sequencer*, *Driver* y *Monitor* son todas clases que están parametrizadas para un tipo especifico de transacción, los desarrolladores del *Test* pueden simplemente sustituir el tipo *trans2* por el tipo *trans1* en el mecanismo de *Factory* al comienzo del *Test*. En un escenario típico, cuando el *Test* construye el *Environment*, el *Environment* construye el *Agent*, y el *Agent* construye el *Sequencer*, el *Driver* y el *Monitor* parametrizados con *trans1*, los componentes

parametrizados del *Agent* se parametrizaran con el tipo almacenado en lugar de *trans1* en el mecanismo de *Factory*, el cual es ahora el tipo compatible *trans2*. No hay necesidad de mantener dos copias del *Sequencer*, del *Driver* y del *Monitor*, y por lo tanto tampoco necesidad de tener un segundo *Agent* que utilice las segundas copias del *Sequencer*, del *Driver* y del *Monitor*. Por lo tanto es posible correr los viejos *Tests* que utilizaban el tipo *trans1* y utilizar exactamente la misma estructura de *testbench* para correr nuevos *Tests* utilizando el tipo *trans2*.

Como un segundo ejemplo, se podría desear usar el mismo tipo de transacción, pero enviando el dato al DUV de forma serial en lugar de forma paralela. La versión serial del DUV podría usar exactamente la misma estructura de *testbench* pero con un *Driver* que envíe transacciones como datos seriales y un *Monitor* que lea las transacciones de salida como datos seriales. Para implementar este *Environment*, el *Test* puede sustituir una segunda versión del *Driver* y del *Monitor* en el mecanismo de *Factory* para que cuando esos componentes sean construidos, la versión serial de los mismos sean instanciados sin necesidad de volver a codificar la totalidad del *Environment*.

La utilización del mecanismo de *Factory* provisto por UVM evita el esfuerzo de crear un *Factory* avanzado o implementar métodos de *Factory* en la definición de las clases. Facilita la reutilización y el ajuste, desde la óptica del usuario del *testbench*, de los componentes de verificación predefinidos. Una de las mayores ventajas de contar con un mecanismo de *Factory* es que es transparente para el desarrollador de *Tests* y reduce el conocimiento de POO necesario para poder utilizarlo provechosamente.

Para mayor información sobre la correcta utilización del mecanismo de *Factory* de UVM para la implementación de *testbenches* reusables referirse a [13].

3.2.2.3. Interconección de componentes

Distintos datos deben ser pasados de componente a componente, por ejemplo de un *Sequencer* a un *Driver*. Los *ports* y *exports* de UVM son usados para comunicación de alto nivel en lugar de comunicación a nivel de señales. Un componente puede enviar un *Sequence Item* a través de un *port*, o recibir un item a través de un *export*.

La conexión entre componentes es inevitablemente de aplicación específica y no puede ser conocida por el autor original de un VC. UVM supera esta problemática mediante la implementación en SV de TLM [9]. La indirección provista por TLM permite que los VC sean implementados para pasar datos a través de sus *ports* y *exports* de TLM, sin tener en cuenta los detalles de otros VCs que pueden estar conectados a éstos. La conexión de *ports* a *exports* es diferida hasta la fase de conexión de la BCL. El método que implementa esta fase es llamado de forma *bottom-up* en cada uno de los componentes del árbol de jerarquía. Esta llamada sucede

luego de que la etapa de construcción de cada componente se haya ejecutado y por consiguiente el árbol de componentes esté completo.

Se espera que los usuarios y desarrolladores de VCs sobrescriban el método de la fase de conexionado, *connect_phase*, en cada componente que implementan. Este método debe hacer todas las conexiones de *port* a *export* necesarias entre los componentes hijos y sus descendientes. Esta semántica común de comunicación TLM permite intercambiar los componentes sin afectar el resto del entorno.

3.2.2.4. Base de datos de configuración y base de datos de recursos

Las bases de datos de configuración y la base de datos de recursos de UVM [12] proveen tablas de acceso global para parámetros y tipos de datos definidos por el usuario.

Estas bases de datos son usadas para configurar el *testbench* mientras es construido.

El código de inicio del test del usuario debe poblar dichas bases de datos, generalmente la base de datos de configuración, la cual luego puede ser consultada por cada componente justo antes de construir sus propios hijos. De esta manera cada componente toma la responsabilidad de construir sus propios hijos de acuerdo a datos de configuración específicos del test almacenados en dicha base de datos.

3.2.2.5. Aleatorización para la generación de estímulos

Para sacar el máximo provecho de la generación de estímulos mediante CRV es importante, no sólo que cada sequence item sea apropiadamente aleatorizado, sino también que se puedan aleatorizar secuencias de actividades coordinadas en puertos individuales y en múltiples puertos del DUV. Además, la aleatorización provista por el desarrollador del VC en su clase base puede no ser del todo apropiada para una tarea de verificación especifica, por lo que es probable que los usuarios del VC necesiten la flexibilidad para agregar, caso a caso, restricciones a la aleatorización especificas a la aplicación.

3.2.2.6. Generación automática de código mediante macros

Hay muchas tareas rutinarias de codificación que deben ser realizadas cuando se crea una nueva clase derivada de una de la UVM BCL. Algunas de estas tareas, trabajosas y propensas a error, se prestan para su automatización mediante macros.

3.2.3. Justificación de su elección

Para llevar a cabo la implementación de un *testbench* estructurado que cumpla con las propiedades de reusabilidad y que permita aumentar el nivel de abstracción para facilitar el desarrollo de las pruebas se selecciona el *framework* UVM.

Dicha metodología se elige para la implementación debido a que promueve la arquitectura de *testbenches* basados en componentes reusables presentada en el capítulo anterior, proveyendo diversas utilidades que permiten llevar esto a la práctica de forma eficiente.

Adicionalmente UVM provee las siguientes ventajas, las cuales se tuvieron en cuenta para su elección:

- Dicho *framework* esta validado en los principales simuladores. (*Incisive Enterprise Simulator* de *Cadence Design Systems*, *ModelSim* y *QuestaSim* de *Mentor Graphics*, VCS de *Synopsys*)
- UVM escala eficientemente de nivel de bloque a nivel sistema.
- Soporta múltiples lenguajes. (SV, e *Verification Language*)
- Es el estándar de la industria hoy en día.
- Existe un mercado de VIP, donde la mayoría de estos son compatibles con UVM.

Por lo tanto, se elige UVM para experimentar con el desarrollo de un ambiente de verificación debido a que es la metodología estándar utilizada por la industria y el mercado de VIPs, está basada en el paradigma de la programación orientada a objetos, y da soporte directo a los conceptos presentados de *testbenches* estructurados.

3.3. Primer Caso de Estudio: Módulo de Buffer FIFO

Para enfocar el esfuerzo en los conceptos de la arquitectura de *testbench* estructurado así como en los conceptos de UVM necesarios para su implementación, con independencia de la complejidad del DUV, se propone implementar la infraestructura de verificación de bloque para un módulo de *buffer* FIFO.

Un *buffer* FIFO es un componente muy utilizado en los diseños digitales para interfacear bloques, tiene un protocolo de funcionamiento claro, y es rico en términos de posibles secuencias de pruebas que se le pueden inyectar para su verificación, haciéndolo de esta forma un caso interesante en términos de generación de secuencias de pruebas, implementación de *Driver* y creación del modelo de referencia, permitiendo de esta forma abordar los principales conceptos de la verificación basada en simulación.

3.3.1. Especificaciones del DUV

El DUV es una FIFO sincrónica, de un solo reloj, en la cual los datos son almacenados en una memoria RAM. En el Código 3.1 se presenta el encabezado de su implementación VHDL donde cada dato es de *dw* bit de ancho y el número máximo de elementos que el *buffer* puede almacenar es $MAX = 2^{addrw}$.

Las señales *rst* (reset), *wr_en* (escritura) y *rd_en* (lectura) controlan la operación de la FIFO, y las señales *empty* (vació) y *full* (lleno) definen su estado.

Los puertos de datos de entrada, *din*, y de salida, *dout*, se utilizan, junto con las señales *wr_en* y *rd_en*, para escribir datos al final de la cola o para leer datos desde el frente de la misma, respectivamente. Cada operación de lectura elimina el elemento almacenado en el frente de la cola. La operación de *reset* vacía la cola. Adicionalmente, las escrituras y lecturas simultáneas están permitidas.

```
1  entity fifo is
2    generic (
3      addrw : natural;
4      dw    : natural);
5    port (
6      clk    : in std_logic;
7      rst    : in std_logic;
8      rd_en  : in std_logic;
9      wr_en  : in std_logic;
10     empty  : out std_logic;
11     full   : out std_logic;
12     din    : in std_logic_vector(dw-1 downto 0);
13     dout   : out std_logic_vector(dw-1 downto 0)
14   );
15 end fifo;
```

Código 3.1: Encabezado *VHDL* de la entidad FIFO.

Las señales de estado y de datos de salida se actualizan en el mismo ciclo de reloj que se lleva a cabo cada operación, estando disponibles para su lectura en el ciclo siguiente. Esto se puede ver en la Figura 3.1 que muestra las especificaciones de temporizado para la siguiente secuencia de operaciones: *Reset, NoOp, Write(D1), Write(D2), Read, Read.*

3.3.2. Arquitectura del Entorno de Verificación

Se plantea el desarrollo de un *testbench* UVM (ver Figura 3.2) para el módulo de *buffer* FIFO formado por un Agente como componente principal. El mismo contiene un *Sequencer*, un *Driver* y un Monitor, y puede operar de forma activa instanciando los tres componentes o de forma pasiva instanciando solo el Monitor.

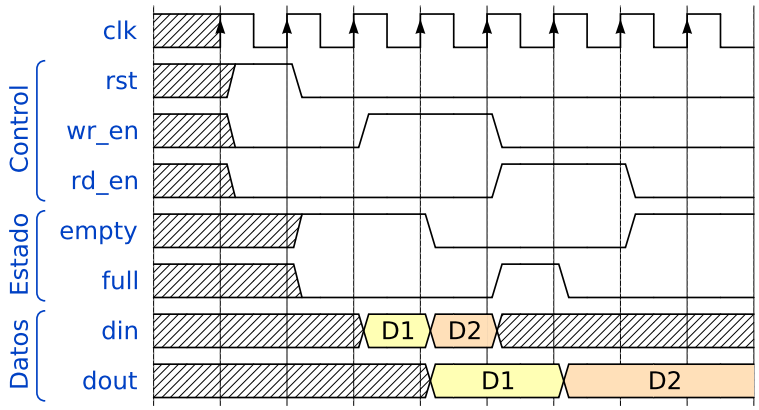

Figura 3.1: Especificaciones de temporizado de señales para una FIFO con capacidad para 2 elementos (MAX=2) para la siguiente secuencia de operaciones: *Reset, NoOp, Write(D1), Write(D2), Read, Read.*

Adicionalmente el Agente contiene un *Scoreboard*, el cual implementa un modelo de referencia de la FIFO, y un medidor de cobertura funcional.

El *Driver* implementado puede inyectar cualquiera de las cuatro operaciones definidas en las especificaciones del DUV (reset, escritura, lectura, y escritura y lectura paralelas). Las secuencias son listas de operaciones de este tipo de comandos. Adicionalmente los elementos de las secuencias pueden utilizarse para generar secuencias aleatorias de operaciones.

Se implementa un BFM en la interfaz del *testbench*, el cual se encarga de los detalles de bajo nivel y será comandada por el *Driver*.

Finalmente el Agente estará instanciado en el Entorno, y este último será creado por la Prueba, la cual definirá las secuencias que correrán en el *Driver* del Agente y configurará el Entorno mediante el uso de un objeto de configuración que centraliza las distintas opciones.

El *testbench* fue implementado y depurado usando *Synopsys VCSMx* y *DVE* version H-2013 con su biblioteca pre compilada de UVM-1.1d. Para agilizar el reiterativo proceso de compilación, debugeo y ejecución se implementó un archivo *Makefile* [28] (Apéndice .1.1) con los respectivos comandos de la herramienta.

Para compilar y correr el *testbench* se utiliza el archivo *Makefile* de la siguiente manera:

```
# make all_v1
```

En las siguientes subsecciones se explican las cuestiones más relevantes de todos los elementos que componen el *testbench* implementado (ver Figura 3.2).

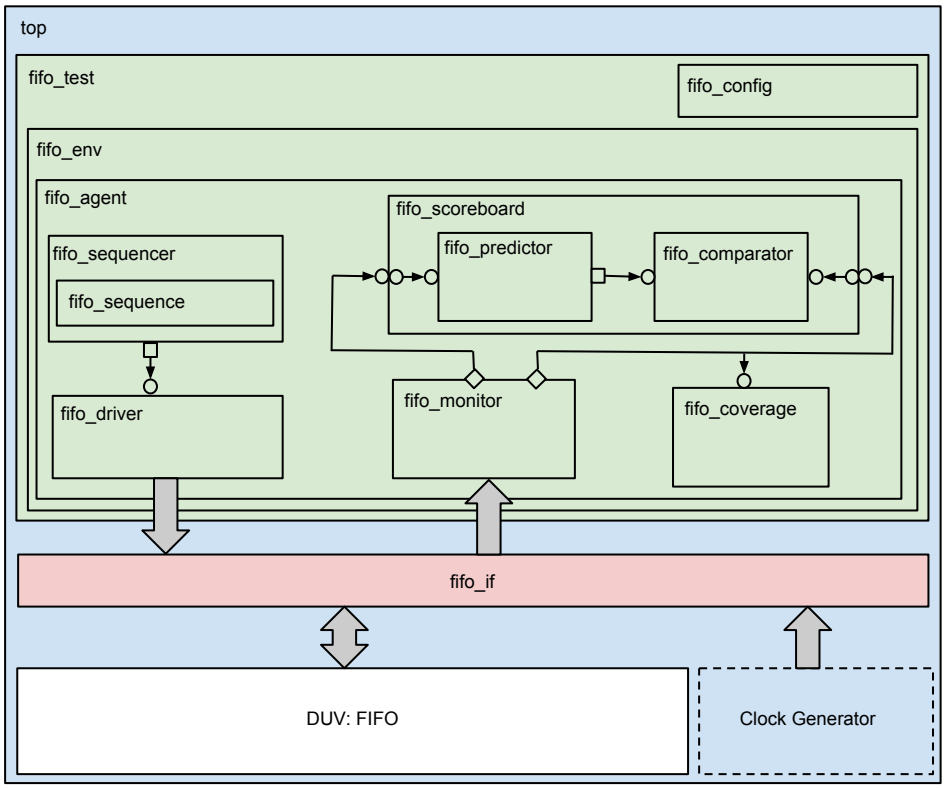

Figura 3.2: Arquitectura del entorno de verificación de la FIFO.

3.3.2.1. Módulo top

La entidad principal del programa de verificación implementado en SV se denomina top, y es un módulo estático de SV que instancia al DUV y a la interfaz, los conecta y define la señal de reloj. Para poder utilizar UVM, este módulo importa el paquete *uvm_package* el cual provee las funcionalidades y recursos básicos del *framework*.

El paquete *uvm_package* provee el método *run_test* que es usado para iniciar la ejecución del *testbench* UVM. Este paquete también brinda acceso a la base de datos de configuración y recursos de UVM, las cuales proveen una ubicación de memoria compartida donde las configuraciones y los recursos compartidos son almacenados y pueden luego ser accedidos por los distintos componentes del *testbench*. El módulo top registra la interfaz en la base de datos de configuración para permitir que el *testbench* basado en clases pueda accederla para luego conectarse al DUV.

3.3.2.2. Interfaz fifo_if

La interfaz facilita la comunicación entre los componentes del *testbench*, basados en clases de SV, y el DUV, a menudo implementado en HDL como este *buffer* FIFO implementado en VHDL.

La interfaz de la FIFO esta implementada como un BFM, es decir, además de definir el formato y encapsular las lineas de conexión (ver Código 3.2), encapsula el protocolo de comunicación, es decir las operaciones que se pueden realizar sobre la FIFO. Las mismas podrían haber estado implementadas por el *Driver*, pero se decidió hacerlas sobre la interfaz ya que son tareas de bajo nivel, asociadas al protocolo del DUV. El *Driver* luego utiliza estas operaciones para implementar sus análogas de forma más clara y sencilla.

```
1  interface automatic fifo_if(input bit CLK);
2  logic RST;
3  logic RD_EN;
4  logic WR_EN;
5  logic FULL;
6  logic EMPTY;
7  logic [31:0] DATA_IN;
8  logic [31:0] DATA_OUT;
```

Código 3.2: Estas son las señales encapsuladas por la interfaz. Dichas señales van por un lado conectadas al DUV, y por otro, el *testbench* las estimula y mide utilizando las tareas definidas en la interfaz.

El BFM implementado en la interfaz provee las siguientes operaciones: *no_op, write, read, read_and_write, reset* y *monitor*. Este enfoque demostró imponer una clara interacción con el DUV ocultando los detalles de bajo nivel de comunicación, sincronización y temporizado. En el Código 3.3 se puede ver la tarea *write*. La construcción de SV denominada *clocking block* fue utilizada, tal como se recomienda en [8], para manejar en un lugar centralizado el temporizado a nivel de señales y evitar condiciones de carrera durante la lectura y escritura de las señales de la interfaz. La utilización del *clocking block* se puede observar en la linea 2 del Código 3.3, en la cual se utiliza *@ master_cb* en lugar del tradicional *@ (posdedge clk)*.

```
1  task automatic write(input bit[31:0] data_in);
2    @ master_cb;
3    `uvm_info("FIFO_IF", $sformatf("WRITE:␣%h", data_in), UVM_MEDIUM)
4    master_cb.DATA_IN <= data_in;
5    master_cb.RST <= 0;
6    master_cb.RD_EN <= 0;
7    master_cb.WR_EN <= 1;
8  endtask
```

Código 3.3: Tarea *write* que modela la operación de escritura de la FIFO. El *testbench* la utiliza para estimular el DUV abstrayéndose de las cuestiones de bajo nivel.

3.3.2.3. Clase fifo_config

Este objeto es usado para centralizar y pasar información de configuración a cada componente del *testbench*. El componente *fifo_test* lo crea, inicializa sus atributos y luego lo almacena en la base de datos de configuración de UVM para que sea accesible por resto de los componentes del *testbench*. Luego cada componente se auto configura usando la información almacenada en el objeto de configuración que retira de dicha base de datos. Un puntero a la interfaz también es pasado al *Driver* y al Monitor utilizando esta metodología.

3.3.2.4. Clase Componente fifo_test

El componente *fifo_test* inicializa la infraestructura del *testbench*. En su fase de UVM *build phase* setea el objeto de configuración descrito, el cual parametriza la estructura general del *testbench* y lo almacena en la base de datos de configuración de UVM. Estas configuraciones involucran: habilitar o deshabilitar los mensajes de debug para cada componente, establecer el modo activo o pasivo del agente y habilitar la creación del *Scoreboard* y del *Coverage Collector*. En esta fase, el test también instancia al *Environment*, el cual instancia al agente. En su fase de UVM *run phase* el test selecciona e inicia una secuencia de prueba en el *Sequencer* como se ve en el Código 3.4.

```
1  task run_phase(uvm_phase phase);
2   fifo_sequence seq;
3   phase.raise_objection(this, "Start␣base_fifo_sequence");
4   seq = fifo_sequence::type_id::create("seq");
5   // Aleatorizar la secuencia.
6   assert( seq.randomize() );
7   // Comenzar a ejecutar la secuencia en el Sequencer del Agente.
8   seq.start(env.agent1.sequencer);
9   phase.drop_objection(this, "Finished␣base_fifo_sequence");
10 endtask
```

Código 3.4: *run phase* del componente *fifo_test*

3.3.2.5. Clase Componente fifo_agent

En su fase de construcción (Código 3.5) y de acuerdo al objeto de configuración almacenado en la *Base de Datos de Configuración* de UVM, el *Agent* crea sus componentes: el *Sequencer*, *Driver*, Monitor, *Scoreboard* y *Coverage Collector*. Si el Agente es activo, debe crear al *Sequencer* y al *Driver*, de otra manera, si es pasivo, debe soló instanciar al Monitor. Esta diferencia se puede observar en la Figura 3.3. En la etapa de UVM *connect phase* conecta los puertos de los componentes de acuerdo a la estructura del *Agent*, por ejemplo, conecta el *Driver* al *Sequencer*.

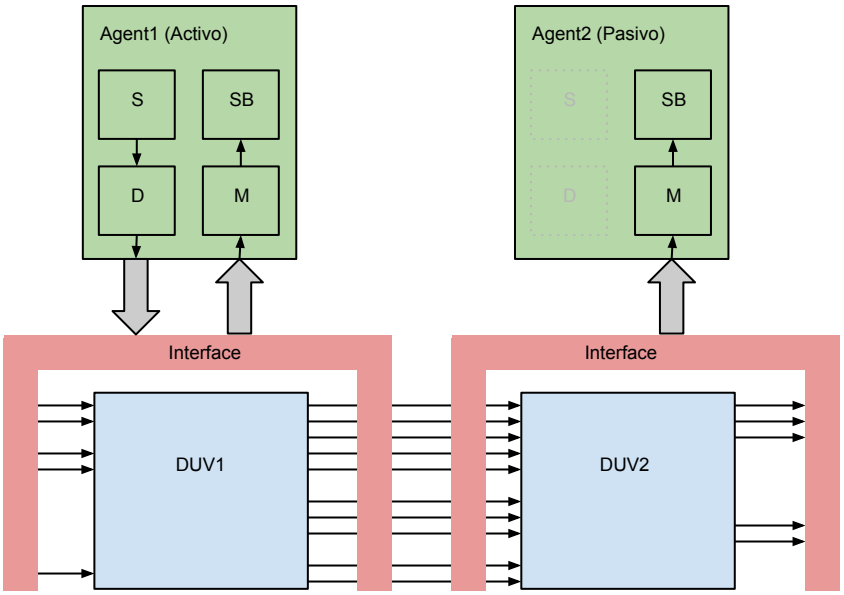

Figura 3.3: Arquitectura del *Agent* en modo pasivo y en modo activo.

```
1  function void build_phase(uvm_phase phase);
2   if (settings.active == UVM_ACTIVE) begin
3    driver=fifo_driver::type_id::create("driver",this);
4    sequencer=fifo_sequencer::type_id::create("sequencer",this);
5   end
6   if (settings.has_functional_coverage == 1) begin
7    coverage = fifo_coverage::type_id::create("coverage",this);
8   end
9   monitor=fifo_monitor::type_id::create("monitor",this);
10  if (settings.has_checker == 1) begin
11   scoreboard=fifo_scoreboard::type_id::create("scoreboard",this);
12  end;
13 endfunction
```

Código 3.5: Build phase del componente *fifo_agent*.

3.3.2.6. Clase Dato fifo_item

Esta clase modela el elemento básico que compone a las secuencias de prueba. Cada elemento contiene un conjunto de atributos que se utilizan para modelar una operación sobre el *buffer* FIFO, sus datos asociados y su estado como se define en la especificación del DUV. Cada uno de sus atributos que modelan comandos o datos de entrada están habilitados para ser aleatorizados. Esto se puede ver en el Código 3.6.

```
1  // Comandos:
2  rand logic read;
3  rand logic write;
4  logic reset;
5  // Estado:
6  logic full;
7  logic empty;
8  // Datos:
9  rand logic [dw-1:0] data_in;
10 logic [dw-1:0] data_out;
```

Código 3.6: Atributos de la clase *fifo_item*.

Esta clase implementa un conjunto de métodos *set* para definir directamente al objeto como una operación de *no_op, write, read, read_and_write* o *reset* con el propósito de utilizarlos para crear pruebas dirigidas. La operación *set_write* se muestra en el Código 3.7.

```
1 function void set_write(logic [dw-1:0] data_in);
2   this.write = 1;
3   this.read = 0;
4   this.reset = 0;
5   this.data_in = data_in;
6 endfunction
```

Código 3.7: Método *set_write* para configurar el ítem como una operación *write*.

3.3.2.7. Clase Dato fifo_sequence

Las secuencias son vectores de prueba de alto nivel. Para definir secuencias de pruebas dirigidas se utilizan los métodos *set* de la clase *fifo_item*, esta organización simplifica la creación de secuencias, mejora su legibilidad y facilita sus modificaciones. Pruebas aleatorias con restricciones también pueden ser fácilmente definidas. Una simple secuencia de prueba dirigida, *Reset, noOperation, Write(1), Write(2), Read, Read*, se muestra en el Código 3.8. Esta secuencia de transacciones genera el comportamiento observado en la forma de onda de la Figura 3.1 de las especificaciones del DUV si la misma es utilizada para estimular una FIFO con capacidad para 2 elementos.

También se pueden definir fácilmente secuencias de pruebas aleatorias implementando un bucle que cree *fifo_item*s, los aleatorice y los envie al *Driver*.

```
1 task body();
2  fifo_item i_rst1, i_noOp1, i_wr1, i_wr2, i_rd1, i_rd2;
3  // Reset
4  i_rst1 = fifo_item::type_id::create("i_rst1");
5  start_item(i_rst1);
6  i_rst1.set_reset();
7  finish_item(i_rst1);
```

```
8   // No Operation
9   i_noOp1 = fifo_item::type_id::create("i_noOp1");
10  start_item(i_noOp1);
11  i_noOp1.set_no_op();
12  finish_item(i_noOp1);
13  // Write 1
14  i_wr1 = fifo_item::type_id::create("i_wr1");
15  start_item(i_wr1);
16  i_wr1.set_write(1);
17  finish_item(i_wr1);
18  // Write 2
19  i_wr2 = fifo_item::type_id::create("i_wr2");
20  start_item(i_wr2);
21  i_wr2.set_write(2);
22  finish_item(i_wr2);
23  // Read 1
24  i_rd1 = fifo_item::type_id::create("i_rd1");
25  start_item(i_rd1);
26  i_rd1.set_read();
27  finish_item(i_rd1);
28  // Read 2
29  i_rd2 = fifo_item::type_id::create("i_rd2");
30  start_item(i_rd2);
31  i_rd2.set_read();
32  finish_item(i_rd2);
33 endtask
```

Código 3.8: Secuencia básica de prueba dirigida (*Reset, noOperation, Write(1), Write(2), Read, Read).*

3.3.2.8. Clase Componente fifo_driver

El componente *Driver* esta a cargo de estimular el DUV. Su tarea principal, *run_phase* (ver Código 3.9), ejecuta un bucle que iterativamente solicita y obtiene *fifo_items* del *Sequencer* y llama a la correspondiente tarea del BFM en la interfaz para generar los vectores de prueba apropiados a nivel de señal.

```
1  task run_phase(uvm_phase phase);
2   fifo_item item;
3   forever begin
4    // Obtener el proximo item proveniente del Sequencer
5    seq_item_port.get_next_item(item);
6    case (item.get_item_type())
7     "RESET"          : bfm.reset();
8     "READ"           : bfm.read();
9     "WRITE"          : bfm.write(item.data_in);
10    "WRITE_AND_READ": bfm.read_and_write(item.data_in);
11    "NO_OP"          : bfm.no_op();
12   endcase
13   seq_item_port.item_done();
```

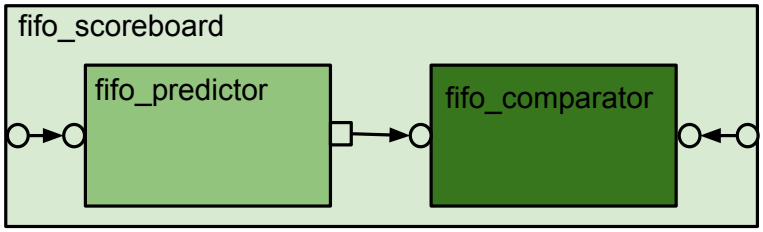

Figura 3.4: Arquitectura del *Scoreboard*.

```
14    end
15  endtask
```

Código 3.9: Tarea principal del *Driver*.

3.3.2.9. Clase Componente fifo_monitor

El Monitor detecta las señales del DUV a través de una tarea especial provista por el BFM de la interfaz. Cuando el Monitor detecta una operación en la interfaz (*read, write, read&write, reset*) crea un *fifo_item*. Dadas las especificaciones de temporizado del DUV, el Monitor tiene que esperar un ciclo de reloj para poder observar los resultados. De esta manera el elemento se ensambla con la operación capturada en el ciclo n y su resultado capturado en el ciclo $n+1$. Finalmente, se envía este *fifo_item* al *Scoreboard* y al Colector de Cobertura.

3.3.2.10. Clase Componente fifo_scoreboard

El componente *Scoreboard* se compone de un *Predictor* y un *Comparator* como se observa en la Figura 3.4. El *Predictor* implementa el modelo de referencia del *buffer* FIFO mediante una clase de SV. El modelo de referencia esta basado en las especificaciones y sólo modela los aspectos funcionales del DUV, desestimando cuestiones temporales. El Monitor del *Agent* produce un *fifo_item* para el ciclo n, sensando las señales de la interfaz, y se lo envía al *Comparator* y al *Predictor*, este último toma los campos de operación del ítem del ciclo n (descartando el resultado sensado del DUV, ya que es esto lo que se desea calcular mediante el modelo de referencia), evalúa el item mediante el modelo de referencia produciendo un resultado de referencia para el ciclo $n + 1$. El *Comparator* sincroniza la elementos provenientes del *Predictor* con los provenientes del Monitor de forma de comparar ítem del mismo ciclo. El *Predictor* los compara y genera el correspondiente mensajes de error si los valores difieren.

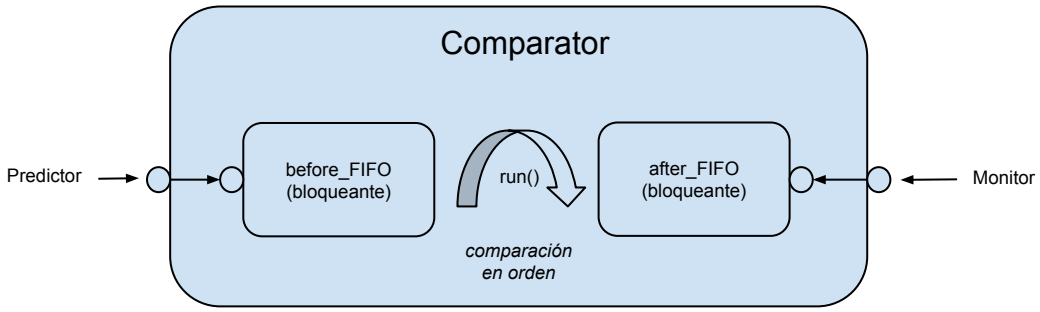

Figura 3.5: El *Comparator* utiliza dos colas bloqueantes para poder sincronizar la comparación de los ítem provenientes del *Predictor* y del Monitor respectivamente. Esto permite que se comparen de forma coordinada el elemento n producido por el DUV con el elemento n producido por el modelo de referencia.

3.3.2.11. Clase componente fifo_coverage

El colector de cobertura también recibe los ítem provenientes del Monitor y los registra en *coverpoints* y *covergroups* de SV, los cuales implementan el modelo de cobertura funcional correspondiente. Este tipo de cobertura es una métrica de caja negra porque no existe acceso a las señales internas del DUV.

En el siguiente capítulo se presenta una implementación completa de un modelo de cobertura funcional para el módulo *buffer* FIFO utilizando los respectivos *coverpoints* y *covergroups* de SV. La clase de SV que lo implementa se puede ver en el Código 4.1 del siguiente capítulo.

3.3.3. Resultados de Simulación

Una vez compilado y depurado el *testbench* se prosiguió a utilizarlo para simular el DUV. Para esto se inyectaron las secuencias de prueba implementadas en la clase *fifo_sequence* (subsección 3.3.2.7) utilizando el entorno de verificación implementado y el respectivo archivo *Makefile* (ver Apéndice .1.1) para ejecutarlo.

Para demostrar, de forma concreta, el correcto funcionamiento del entorno de verificación se presenta el resultado de simulación para una secuencia de prueba particular inyectada al *buffer* FIFO instanciado para poder almacenar hasta 8 datos (*addrw=3*) de 32 bits cada uno (*dw=32*):

SeqBasica: *Reset, NoOperation, Write(1), Write(2), Read, Read*

Esta secuencia de operaciones estimula varias de las funcionalidades básicas del diseño, las operaciones de *Reset, Read, Write* y el comportamiento de la señal de estado *empty*. La implementación en UVM de esta secuencia se puede observar en el Código 3.8.

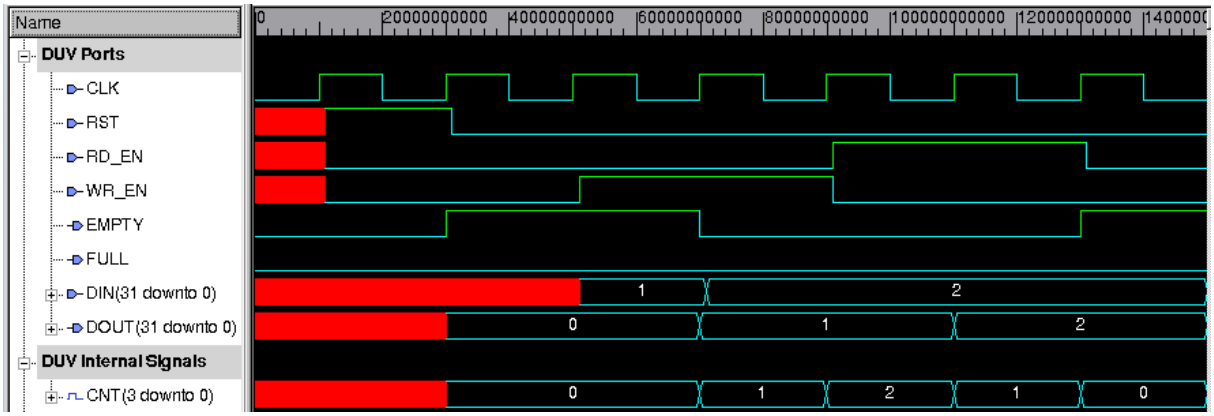

Figura 3.6: Formas de onda generadas por *Synopsys DVE* para la simulación correspondiente a la estimulación del DUV con la secuencia de operaciones *Reset, NoOp, Write(1), Write(2), Read, Read*. La señal interna *CNT* denota la cantidad de elementos almacenados en la FIFO.

Se elige esta secuencia de prueba con el objetivo de ilustrar de forma clara el correcto funcionamiento del entorno de verificación, mostrando la correspondencia entre la secuencia de prueba implementada en UVM (Código 3.8) y la forma de ondas que representa el funcionamiento del dispositivo al ser estimulado por la misma (Figura 3.6), permitiendo mapear claramente cada transacción o item de UVM en la forma de onda correspondiente. También se presenta la salida por consola de la simulación, en la cual se puede observar el funcionamiento del *Scoreboard*.

En la Figura 3.6 se puede observar la forma de ondas en la herramienta *Synopsys DVE* producto de la simulación generada por *Synopsys VCSMx*. En la misma se observan los puertos del DUV los cuales están conectados a la interfaz del *testbench*. Se puede observar que la forma de onda del simulador para esta secuencia de prueba se corresponde con la de la forma de ondas de las especificaciones presentada en la Figura 3.1 (salvo por el comportamiento de la señal de *full* que se activa en esta última debido a que representa una FIFO con solo dos elementos).

En la Salida por Consola 3.10 se presenta la salida por consola de la simulación correspondiente a la estimulación con la secuencia definida. Se puede observar el funcionamiento del *Scoreboard*, más precisamente del *Comparator*, el cual imprime la comparación entre los valores provenientes del DUV registrados por el Monitor y los valores esperados generados por el *Predictor* o modelo de referencia.

Se puede observar un mensaje del *Comparator* por cada operación modelada en la secuencia de prueba, salvo por la operación *NoOperation*, la cual no es evaluada por el *Scoreboard*, arrojando un total de 5 mensajes, coincidentes con las 5 operaciones de la secuencia.

Luego se puede observar que se reporta la cantidad total de coincidencias (*matches*) y diferen-

cias (*mismatches*) entre estos valores. En la etapa de depuración del *testbench* resultó útil cortar la simulación automáticamente cuando se detectaba una diferencia entre el valor del DUV y el valor esperado, de forma de encontrar más fácilmente la causa del error.

También en la salida por consola se presenta la topologia o arquitectura del *testbench*. En esta sección se pueden observar todos los componentes instanciados así como su jerarquía y tipo de dato. También se detallan los puertos de comunicación de cada componente. Se puede observar que la información de la arquitectura del *testbench* concuerda con la presentada anteriormente (Arquitectura del Entorno de Verificación 3.3.2)

Finalmente, en la salida por consola, se resume la cantidad de mensajes, según la importancia y el tipo, de los cuales se pueden observar los 7 producidos anteriormente por el *Comparator*

```
Chronologic VCS simulator copyright 1991-2013
Contains Synopsys proprietary information.
Compiler version H-2013.06; Runtime version H-2013.06;  Jun 23 14:53 2015
----------------------------------------------------------------
UVM-1.1d.VCS
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
----------------------------------------------------------------

UVM_INFO @ 0: reporter [RNTST] Running test fifo_test...

UVM_INFO source/fifo_comparator.sv(62) @ 50000000000: uvm_test_top.env.agent1.scoreboard.comparator [
     FIFO_COMPARATOR] Predictor: (OP=RESET | DATA_IN=xxxxxxxx | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=1) does match
     dut: (OP=RESET | DATA_IN=xxxxxxxx | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=1)
UVM_INFO source/fifo_comparator.sv(62) @ 90000000000: uvm_test_top.env.agent1.scoreboard.comparator [
     FIFO_COMPARATOR] Predictor: (OP=WRITE | DATA_IN=1 | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=0) does match dut: (OP
     =WRITE | DATA_IN=1 | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=0)
UVM_INFO source/fifo_comparator.sv(62) @ 110000000000: uvm_test_top.env.agent1.scoreboard.comparator [
     FIFO_COMPARATOR] Predictor: (OP=WRITE | DATA_IN=2 | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=0) does match dut: (OP
     =WRITE | DATA_IN=2 | DATA_OUT=xxxxxxxx | FULL=0 | EMPTY=0)
UVM_INFO source/fifo_comparator.sv(62) @ 130000000000: uvm_test_top.env.agent1.scoreboard.comparator [
     FIFO_COMPARATOR] Predictor: (OP=READ | DATA_IN=xxxxxxxx | DATA_OUT=1 | FULL=0 | EMPTY=0) does match dut: (OP=
     READ | DATA_IN=xxxxxxxx | DATA_OUT=1 | FULL=0 | EMPTY=0)
UVM_INFO source/fifo_comparator.sv(62) @ 150000000000: uvm_test_top.env.agent1.scoreboard.comparator [
     FIFO_COMPARATOR] Predictor: (OP=READ | DATA_IN=xxxxxxxx | DATA_OUT=2 | FULL=0 | EMPTY=1) does match dut: (OP=
     READ | DATA_IN=xxxxxxxx | DATA_OUT=2 | FULL=0 | EMPTY=1)

UVM_INFO /opt/synopsys/H-2013.06/vcs_mx_vH-2013.06/etc/uvm/base/uvm_objection.svh(1268) @ 150000000000: reporter [
     TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase

UVM_INFO @ 150000000000: uvm_test_top.env.agent1.scoreboard.comparator [FIFO_COMPARATOR] Matches:    5
UVM_INFO @ 150000000000: uvm_test_top.env.agent1.scoreboard.comparator [FIFO_COMPARATOR] Mismatches: 0

UVM_INFO @ 150000000000: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------------------
Name                         Type                         Size  Value
----------------------------------------------------------------------
uvm_test_top                 fifo_test                    -     @468
  env                        fifo_env                     -     @483
    agent1                   fifo_agent                   -     @492
      coverage               fifo_coverage                -     @651
        analysis_imp         uvm_analysis_imp             -     @659
      driver                 fifo_driver                  -     @501
        rsp_port             uvm_analysis_port            -     @518
        seq_item_port        uvm_seq_item_pull_port       -     @509
      monitor                fifo_monitor                 -     @669
        ap_cov               uvm_analysis_port            -     @704
```

```
      ap_sb_after          uvm_analysis_port           -     @695
      ap_sb_before         uvm_analysis_port           -     @686
    scoreboard             fifo_scoreboard             -     @678
      after_export         uvm_analysis_export         -     @722
      before_export        uvm_analysis_export         -     @713
      comparator           fifo_comparator             -     @731
        after_export       uvm_analysis_export         -     @873
        after_fifo         uvm_tlm_analysis_fifo #(T)  -     @811
          analysis_export  uvm_analysis_imp            -     @855
          get_ap           uvm_analysis_port           -     @846
          get_peek_export  uvm_get_peek_imp            -     @828
          put_ap           uvm_analysis_port           -     @837
          put_export       uvm_put_imp                 -     @819
        before_export      uvm_analysis_export         -     @864
        before_fifo        uvm_tlm_analysis_fifo #(T)  -     @758
          analysis_export  uvm_analysis_imp            -     @802
          get_ap           uvm_analysis_port           -     @793
          get_peek_export  uvm_get_peek_imp            -     @775
          put_ap           uvm_analysis_port           -     @784
          put_export       uvm_put_imp                 -     @766
      predictor            fifo_predictor              -     @740
        analysis_imp       uvm_analysis_imp            -     @748
        results_ap         uvm_analysis_port           -     @882
    sequencer              fifo_sequencer              -     @528
      rsp_export           uvm_analysis_export         -     @536
      seq_item_export      uvm_seq_item_pull_imp       -     @642
      arbitration_queue    array                       0     -
      lock_queue           array                       0     -
      num_last_reqs        integral                    32    'd1
      num_last_rsps        integral                    32    'd1
----------------------------------------------------------------

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    9
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[FIFO_COMPARATOR]     7
[RNTST]     1
[TEST_DONE]     1
$finish called from file "/opt/synopsys/H-2013.06/vcs_mx_vH-2013.06/etc/uvm/base/uvm_root.svh", line 438.
$finish at simulation time         150000000000

           V C S   S i m u l a t i o n   R e p o r t
Time: 150000000000 ns
CPU Time:      0.340 seconds;       Data structure size:   0.3Mb
Tue Jun 23 14:53:51 2015
```

Salida por Consola 3.10: Resultados de la simulación ejecutada en *Synopsys VCSMx*. La salida corresponde a la secuencia de operaciones *Reset, NoOp, Write(1), Write(2), Read, Read*. Se puede observar el funcionamiento del *Scoreboard* y el reporte de la arquitectura del *testbench*.

3.3.4. Análisis

Inicialmente, se pensó que la arquitectura del Agente activo debía definir un canal de comunicación entre el *Sequencer* y el *Scoreboard*, más precisamente, el *Predictor*. Esto se pensó de forma de enviarle al *Predictor* el mismo ítem que era enviado al *Driver* para ser inyectado al

DUV. El objetivo era obtener el valor predicho por el modelo de referencia y luego compararlo con el valor de salida del DUV, comprobando de esta forma la correctitud de la FIFO. Luego se comprendió que si el Agente debía chequear la correctitud del DUV operando también en modo pasivo, este enfoque no funcionaría. Esto se debe a que el *Sequencer* no existe cuando el Agente es creado en modo pasivo, por lo que no le llegarían al *Predictor* los ítem con las operaciones que le estarían llegando al DUV, por lo que no se podrían comparar las salidas si el Agente operara en este modo, perdiendo la capacidad de verificación. Esto definió la actual arquitectura del Agente, con el Monitor comunicándose solo con el *Scoreboard*.

Se planteó la opción de desarrollar dos Agentes, uno que haga las operaciones de productor, *write* y otro Agente que haga las operaciones de consumidor, *read*. De esta forma se intanciarían dos agentes y se debería colocar el *Scoreboard* y el medidor de cobertura en el Entorno, alimentando a estos componentes con los elementos producidos por los monitores de ambos Agentes. Adicionalmente a este esquema habría que agregarle un *Virtual Sequencer*, que es un componente de UVM utilizado para coordinar los *Sequencers* de varios Agentes, de forma de poder crear secuencias de escrituras y lecturas significativas. Este esquema además implicaría el doble de esfuerzo, ya que habría que codificar dos Agentes, cada uno con su *Driver* y su Monitor particular, habría que hacer todo el conexionado, coordinando los *Sequencers* con el *Virtual Sequencer*, y teniendo que implementar un *Scoreboard* más complejo que tenga que coordinar y sincronizar la entrada de dos agentes, para poder alimentar correctamente al modelo de referencia.

Se observó que las mayores dificultades se presentaron al implementar los componentes encargados de interactuar con el DUV a nivel de señal, en este caso, el BFM implementado en la interfaz de SV, y el Monitor.

3.4. Segundo Caso de Estudio: Módulo de EEPROM I2C

La experiencia adquirida con el *testbench* para el *FIFO buffer* fue aprovechada para el desarrollo de un *testbench* UVM para la verificación a nivel de unidad de un módulo EEPROM I2C esclavo. La arquitectura del *testbench* es similar a la del caso de estudio previo, permitiendo reutilizar la mayoría de la estructura y los VC ya implementados con mínimas modificaciones.

En el desarrollo anterior se identificó que, pese a que la FIFO tiene un protocolo de interacción sencillo, la implementación de las cuestiones de bajo nivel del *testbench*, es decir el manejo de la estimulación a nivel de señal que involucra aspectos de temporizado y sincronización, son los más complejos de desarrollar.

Para enfocarse en esto se decidió elegir un DUV que emplee un protocolo estándar de comunicación utilizado en la industria, de forma tal de poder desarrollar un *Driver* más complejo,

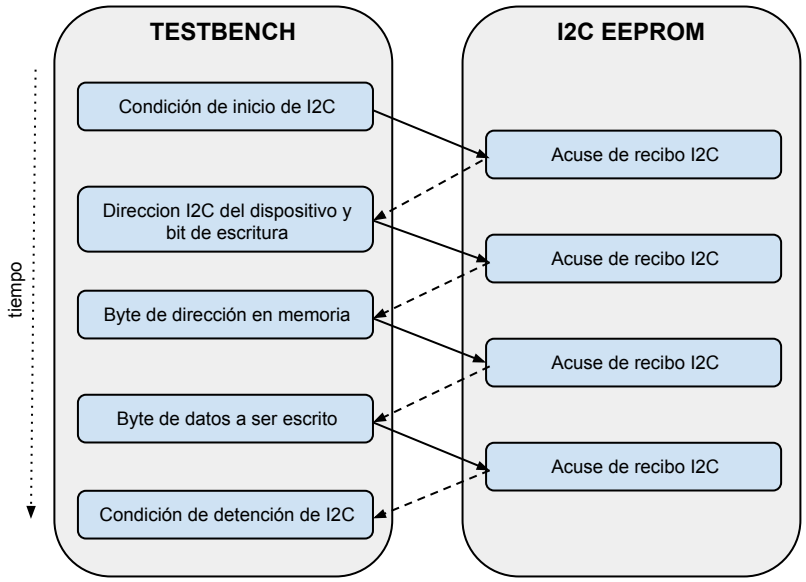

Figura 3.7: El flujo del protocolo de comunicación para escribir un byte en una dirección de memoria específica de la EEPROM

y ganar experiencia en el proceso. En función de esto se decidió desarrollar un *testbench* para verificar el módulo EEPROM I2C.

3.4.1. Especificaciones del DUV

El DUV es una EEPROM I2C 24C01 [45] de 1 K-bit de memoria con palabras de 8 bits de ancho. Se trata de un módulo esclavo I2C [39] implementado como una entidad VHDL *Open Source* con dirección de dispositivo 1010000 y 7 bits de modo de direccionado I2C.

Para escribir un byte en la EEPROM I2C esclava los mensajes I2C mostrados en la Fig. 3.7 deben ser enviados y recibidos por el dispositivo maestro I2C (*testbench*).

Para leer un byte de una dirección de memoria en la EEPROM se le deben enviar los siguientes mensajes del protocolo I2C:

1. Condición de Inicio de I2C,
2. Dirección I2C del dispositivo y bit de escritura I2C,
3. Byte de dirección en EEPROM,
4. Condición de inicio de I2C y

5. Dirección I2C del dispositivo y bit de lectura I2C.

En primer lugar se debe escribir la dirección de memoria EEPROM de la cual se desea leer a través de un mensaje I2C (paso 2 y 3), a continuación, se debe enviar un mensaje de lectura I2C (paso 5) para leer un byte de la dirección de memoria previamente definida.

3.4.2. Arquitectura del Entorno de Verificación

Se plantea el desarrollo de un *testbench* UVM (ver Figura 3.8) para la EEPROM I2C formado por un Agente como componente principal. El mismo contiene los elementos para facilitar la estimulación, esto es, un *Sequencer* y un *Driver* con el respectivo BFM.

En este desarrollo no se incluyeron los componentes de análisis, es decir, ni un Monitor, ni un *Scoreboard* ni un componente para medir la cobertura, ya que se decidió enfocar el desarrollo de los componentes de estimulación que trabajan a bajo nivel, esto es, el *Driver* y su BFM.

El *testbench* fue implementado y depurado usando *Synopsys VCSMx* y *DVE* versión H-2013 con su biblioteca precompilada de UVM-1.1d. Para agilizar el reiterativo proceso de compilación, depuró y ejecución se implementó un archivo *Makefile* (apéndice .1.2) con los respectivos comandos de la herramienta. La herramienta permitió el trabajo con lenguajes mixtos, es decir, el diseño en VHDL y el *testbench* en SV.

Para compilar y correr el *testbench* se utiliza el archivo *Makefile* de la siguiente manera:

```
# make all
```

Solo los componentes específicos y más relevantes del *testbench* son presentados en las siguientes subsecciones, detallando decisiones de diseño e implementación relacionadas al protocolo I2C. Las modificaciones más relevantes, respecto al *testbench* anterior, involucran el BFM que ahora implementa el protocolo I2C.

3.4.2.1. Interfaz i2c_if

En el protocolo I2C, *SDA* y *SCL* son ambas señales bidireccionales, conectadas a una tensión de alimentación positiva a través de una fuente de corriente o resistencia *pull-up*. Cuando el bus está libre, ambas líneas están en alta impedancia. Este comportamiento se modela en la interfaz usando el tipo de datos de SV *str1* (triestado con pull-up) y *scl_oe* y *sda_oe* como sus respectivas variables de control como se muestra en el Código 3.11. El *testbench* estimula el DUV a través de la interfaz utilizando las variables de control *scl_oe* y *sda_oe*.

```
1  interface i2c_m_if(inout tri1 sda);
2   // Senales del bus I2C
3   tri1  scl;
```

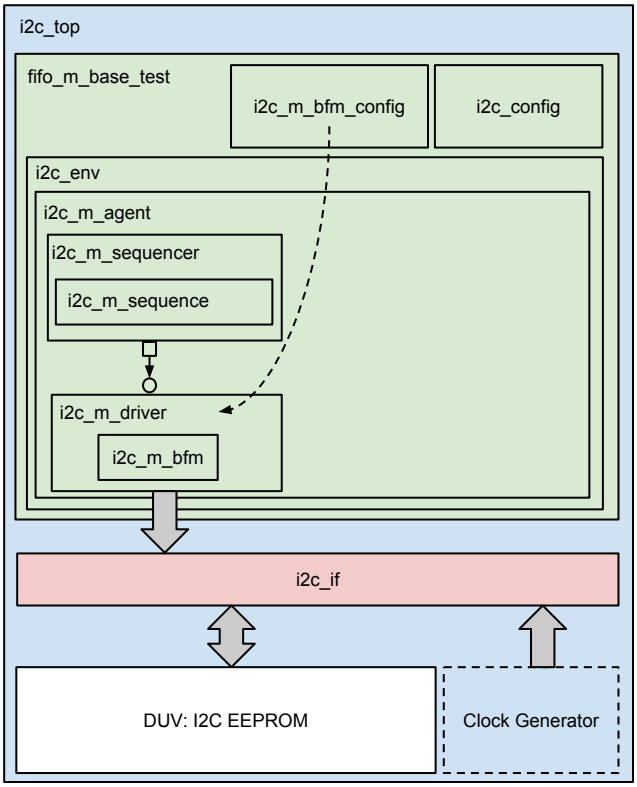

Figura 3.8: Arquitectura del entorno de verificación del I2C EEPROM. Se puede observar que los componentes *Driver* y Monitor utilizan el objeto *i2c_m_bfm_config* para configurar cuestiones de bajo nivel, como temporizado, frecuencia de reloj, etc, según el modo de velocidad del protocolo I2C seleccionado en el *Test*.

```
4   // Control de los buffers tri-estado
5   logic scl_oe;
6   logic sda_oe;
7   // Buffers tri-estado
8   assign scl = scl_oe ? 1'bz : 1'b0;
9   assign sda = sda_oe ? 1'bz : 1'b0;
10 endinterface
```

Código 3.11: Las lineas bidireccionales *SDA* y *SCL* modeladas con resistencias *pull-up* en la interfaz. Si *scl_oe* tiene valor alto entonces *scl* tiene alta impedancia. Si *scl_oe* tiene valor bajo entonces *scl* tiene valor bajo.

3.4.2.2. Clase i2c_m_bfm

Esta clase modela el BFM utilizado por el *Driver*. La misma implementa los detalles de bajo nivel del protocolo I2C mediante tres tareas principales que interactúan con la interfaz de acuerdo con la información del *i2c_m_item* que recibe:

- *busIdle(i2_m_item tr):* Mantiene el bus ocioso por el correspondiente número de ciclos de reloj.
- *initialValues():* Pone las lineas *SDA* y *SCL* en alta impedancia.
- *burstReadWrite(i2c_m_item tr):* Sigue el protocolo I2C para escribir o leer datos del dispositivo I2C deseado. El pseudo algoritmo que explica cómo funciona se muestra en el Código 3.12.

```
1  // Generar la condicion de inicio I2C.
2  i2cStartCondition();
3  // Enviar la direccion del dispositivo I2C y el bit de lectura/escritura.
4  i2cAddressRWbit();
5  if (item.OperationType = WRITE)
6   // Enviar el buffer de datos.
7   then i2cByteWrite(item)
8   // Recibir los datos.
9   else i2cByteRead()
10 if (item.genStop = TRUE)
11  // Generar la condicion de detencion de I2C.
12  then i2cStopCondition()
```

Código 3.12: El pseudo algoritmo para la tarea *burstReadWrite*.

Estas tareas utilizan los parámetros de temporizado obtenidas del objeto *i2c_m_bfm_config* (explicado al final de esta sección) para establecer el sincronizado de las señales.

3.4.2.3. Clase Dato i2c_m_item

Estos elementos representan las operaciones que pueden ser realizadas por un dispositivo maestro I2C sobre un bus de este mismo protocolo.

Los campos principales de esta clase son: un campo para la dirección I2C del dispositivo con el cual se desea interactuar a través de esta transacción; un campo que especifica el tipo de operación (*READ, WRITE, RESET, IDLE*); un campo para el dato que se va a leer o escribir; un campo para modelar la cantidad tiempo de inactividad en el bus si se trata de una operación *IDLE*; y un campo para decidir si la señal de detención I2C sera generada al final de la transacción.

Se implementan tres métodos para definir fácilmente estos elementos con el objetivo de facilitar la creación de secuencias de pruebas dirigidas:

- *setBusIdle(time idleTime):* hace que el ítem represente un tiempo de inactividad en el bus.
- *setSoftReset():* hace que el ítem represente una operación de I2C denominada software reset, en la cual se hace un llamado general de I2C (0000 0000) y se envía 0000 0110 (06h) como segundo byte. Esta funcionalidad es opcional en el protocolo por lo que no todos los dispositivos contestan a este comando, por lo tanto no se espera respuesta.
- *setReadData(int address, int lenght, int genStop):* hace que el ítem represente una operación I2C de lectura de *length* bytes de longitud sobre el dispositivo I2C con la dirección especificada. *genStop* establece si se genera una condición de detención I2C después del último byte leído.
- *setWriteData(int address, bit8 dataInBuff[], int genStop):* hace que el ítem represente una operación de escritura I2C de *length* bytes sobre el dispositivo I2C con la dirección especificada. *genStop* establece si se genera una condición de detención I2C después del último byte escrito.

3.4.2.4. Clase Dato i2c_m_sequence

Secuencias de pruebas dirigidas pueden ser fácilmente implementadas usando los métodos *set* de la clase ítem descrita anteriormente. Un ejemplo se muestra en el Código 3.13, en la cual se escriben dos bytes en ráfaga a partir de la dirección 0 de la EEPROM y luego se leen los dos bytes, también en ráfaga a partir de la dirección 0. Primero se crea un ítem a través del método *Factory* de UVM, luego la dirección I2C de 7 bits (1010000) del dispositivo EEPROM se carga en una variable, luego una lista de bytes es creada (0, 1, 2), donde el primer elemento (0) representa la dirección de memoria dentro de la EEPROM y el resto de los elementos los bytes

que se escribirán. También se establece una variable para decidir si se generará una señal de detención I2C al finalizar la escritura de los bytes. Después de eso, el ítem se establece mediante el método *setWriteData* para modelar una transacción de escritura con esa información. Luego se genera un tiempo de inactividad en el bus (*Idle*), se escribe la dirección sobre la cual se desea leer en la EEPROM, y finalmente se leen de forma consecutiva los cinco bytes escritos previamente.

```
1  task body();
2      i2c_m_item i0, i1, i2, i3;
3      bit8 dataInBuff[$];
4      int device_address;
5      int genStop;
6
7      // Direccion de la EEPROM
8      device_address = 7'b1010000;
9
10     // Escribir una rafaga de 2 bytes consecutivos a partir de la direccion 0 dentro
           de la EEPROM.
11     i0 = i2c_m_item::type_id::create("i0");
12     start_item(i0);
13     genStop = 1;
14     // El primer dato de dataInBuff es la direccion dentro de la EEPROM.
15     dataInBuff = {0, 1, 2};
16     i0.writeData(device_address, dataInBuff, genStop);
17     finish_item(i0);
18
19     // Poner el bus en estado Idle.
20     i1 = i2c_m_item::type_id::create("i1");
21     start_item(i1);
22     i1.busIdle(100000);
23     finish_item(i1);
24
25     // Establecer la direccion dentro de la EEPROM en 0.
26     i2 = i2c_m_item::type_id::create("i2");
27     start_item(i2);
28     genStop = 1;
29     dataInBuff = {0};
30     i2.writeData(device_address, dataInBuff, genStop);
31     finish_item(i2);
32
33     // Leer una rafaga de 2 bytes consecutivos a partir de la direccion escrita en la
           operacion anterior.
34     i3 = i2c_m_item::type_id::create("i3");
35     start_item(i3);
36     genStop = 1;
37     i3.readData(device_address, 2, genStop);
38     finish_item(i3);
39 endtask
```

Código 3.13: Una secuencia de prueba dirigida que escribe 2 bytes consecutivos comenzando en la dirección 0, hace un *Idle* y luego lee los 2 bytes de forma consecutiva.

3.4.2.5. Clase Componente i2c_m_driver

El desarrollo del *Driver* fue simplificado por el uso del BFM implementado por la clase *i2c_m_bfm*. El *Driver* recibe el ítem desde el *Sequencer* y en función del tipo de ítem (*READ, WRITE, IDLE*) llama al método apropiado del BFM para que se encargue de los detalles de bajo nivel de temporizado y sincronización de señales.

3.4.2.6. Clase i2c_m_bfm_config

Este objeto se utiliza para configurar el BFM. Establece, de forma centralizada, la configuración I2C de temporizado, como frecuencia de reloj, tiempos de *hold* para las condiciones de inicio y detención del protocolo, tiempos de *setup* y *hold* de los datos, etc. Los valores de estos parámetros se pueden obtener a partir de la página 48 de [39] para todos los modos de velocidad del protocolo I2C.

Este objeto es creado e inicializado por el componente *Test*. Se configura seleccionado el modo de velocidad del protocolo I2C (*Standart, Fast* o *FastPlus)* y a partir de esa información el objeto deriva la configuración de temporizado que será usada por el BFM. Luego este objeto es colocado en la base de datos de configuración de UVM para permitir que el BFM lo adquiera.

3.4.3. Resultados de Simulación

Una vez compilado y depurado el *testbench* se prosiguió a utilizarlo para simular el DUV. Para esto se inyectaron las secuencias de prueba implementadas en la clase *i2c_m_sequence* (subsección 3.4.2.4) utilizando el entorno de verificación implementado y el respectivo archivo *Makefile* (ver Apéndice .1.2) para ejecutarlo.

Para demostrar, de forma concreta, el correcto funcionamiento del entorno de verificación se presenta el resultado de simulación del DUV para una secuencia de prueba particular. La misma se describe a continuación:

1. **Escritura I2C (dirección, datos):** Escribir, en modo ráfaga, a partir de la dirección 0 de la EEPROM un 1 y luego un 2. Para esto, se selecciona primero el dispositivo I2C (EEPROM), se hace una escritura I2C con la dirección de memoria a partir de la cual se desea escribir (0) y luego se hacen dos escrituras I2C consecutivas escribiendo un 1 y luego un 2.

2. **Idle I2C**: Poner el bus en estado *Idle* durante 10000 unidades de tiempo.

3. **Escritura I2C (dirección):** Establecer en la EEPROM la dirección de memoria 0 a partir de la cual se desea leer. Se selecciona el dispositivo I2C (EEPROM), se hace una

escritura I2C indicando a la memoria que se desea apuntar a la dirección 0.

4. **Lectura I2C:** Leer, en modo ráfaga, en la EEPROM a partir de la dirección establecida en el paso anterior. Para esto, se selecciona primero el dispositivo I2C (EEPROM), y luego se hacen dos lecturas I2C en ráfaga a partir de la dirección 0.

Esta secuencia de operaciones estimula varias de las funcionalidades básicas del diseño, las operaciones de escritura I2C, lectura I2C e *Idle*. La implementación en UVM de esta secuencia se puede observar en el Código 3.13.

Se elige esta secuencia de prueba con el objetivo de ilustrar de forma clara el correcto funcionamiento del entorno de verificación, mostrando la correspondencia entre la secuencia de prueba implementada en UVM (Código 3.13) y la forma de ondas que representa el funcionamiento del dispositivo al ser estimulado por la misma (Figura 3.9), permitiendo mapear claramente cada transacción o item de UVM en la forma de onda correspondiente. También se presenta la salida por consola de la simulación, en la cual se puede observar el funcionamiento del *Driver* y el BFM.

En la Figura 3.9 se puede observar la forma de ondas en la herramienta *Synopsys DVE* producto de la simulación generada por *Synopsys VCSMx*. En la misma se observan las señales de la interfaz, los puertos del DUV, así como también sus señales internas. Particularmente se puede observar la señal *THIS_STATE* la cual indica en que fase del protocolo I2C se encuentra la EEPROM. Siguiendo el progreso de esta señal en la forma de onda se puede visualizar claramente como la EEPROM evoluciona según el protocolo I2C acorde a la secuencia de prueba inyectada. Se puede observar cómo primero se identifica el dispositivo I2C (*ID_CODE*), luego se escribe la dirección a la cual se desea apuntar (*WR_ADDR*), y finalmente se observa la escritura de un 1 y de un 2 (*WR_DATA*). Luego se observa el *IDDLE* durante la cantidad de tiempo especificada. Se identifica el dispositivo I2C nuevamente (*ID_CODE*), se escribe la dirección a la que se desea apuntar para la lectura (*WR_ADDR*), se vuelve a identificar el dispositivo (*ID_CODE*), y se prosiguen a hacer las dos lecturas (*RD_DATA*). El cambio de estado que se observa después de cada cambio de estado de esta señal es el acuse de recibo del protocolo I2C enviado por la EEPROM al dispositivo maestro I2C. Adicionalmente se puede observar la correspondencia entre la forma de onda y la secuencia de prueba de UVM (Código 3.13). Finalmente se puede comprobar cómo se corresponde el comportamiento de la simulación con el comportamiento del protocolo I2C presentado en la subsección 3.4.1.

En la Salida por Consola 3.14 se presenta el resultado de la simulación correspondiente a la estimulación con la secuencia definida. Se puede observar el comienzo de la secuencia (*[I2C_M_SEQUENCE]*), el funcionamiento del *Driver* (*[I2C_DRIVER]*) y el BFM (*[I2C_BFM]*). El *Driver* reporta un mensaje por cada operación que contiene la secuencia

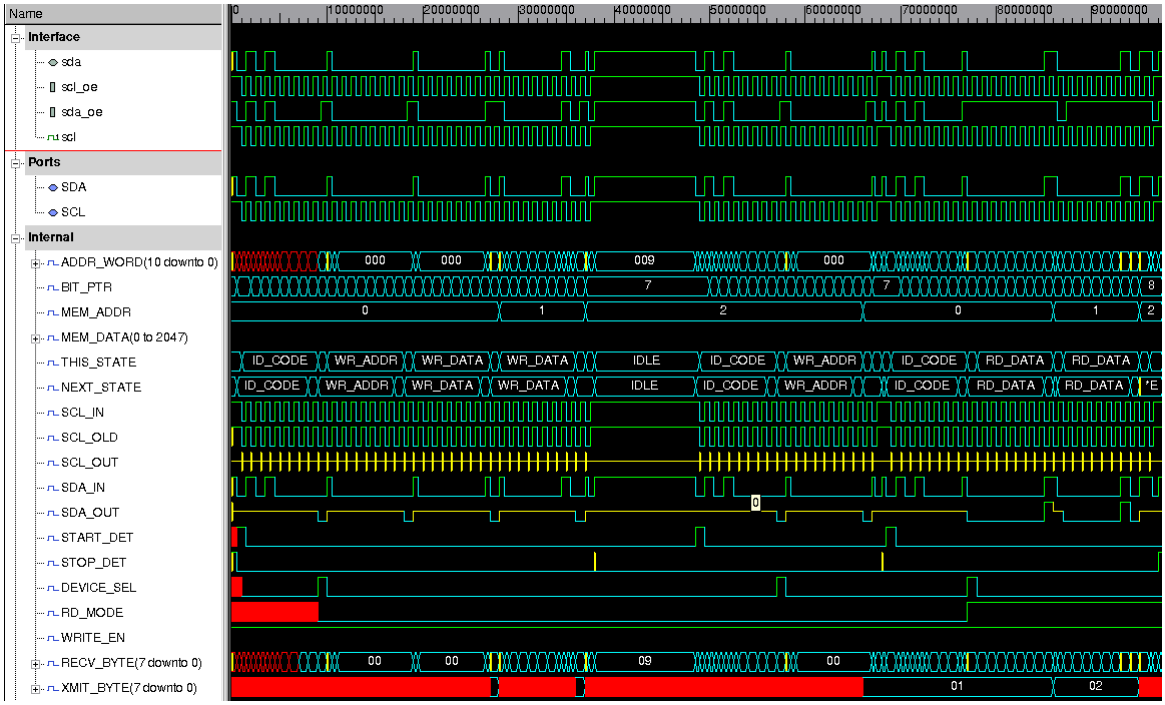

Figura 3.9: Formas de onda generadas por *Synopsys DVE* para la simulación correspondiente a la estimulación de la EEPROM I2C con la secuencia de operaciones presentada en esta subsección.

de prueba. Se puede ver el tipo de operación analizando el campo *TrType* de este mensaje. A su vez, el *Driver* procesa cada item mediante el uso del BFM el cual descompone el item de alto nivel en su respectiva codificación del protocolo I2C. De esta manera se puede observar como se modela, inicialmente, la secuencia de prueba a alto nivel, luego cada item de la misma es procesado por el *Driver* mediante el uso del BFM, el cual se encarga de procesar la operación a bajo nivel siguiendo cada paso del protocolo I2C. Se puede observar claramente el correcto funcionamiento de la EEPROM I2C para esta secuencia de prueba, en la cual se escribe un 1 y un 2 en la operación de escritura, y luego se leen esos mismos valores en la operación de lectura.

También en la salida por consola se presenta la topologia o arquitectura del *testbench*. En esta sección se pueden observar todos los componentes instanciados, así como su jerarquía y tipo de dato. También se detallan los puertos de comunicación de cada componente. Se puede observar que la información de la arquitectura del *testbench* concuerda con la presentada anteriormente (sección 3.4.2). El BFM no aparece en este último reporte ya que no es una clase que herede de la clase base UVM *Component* por lo que el *framework* no lo reporta.

```
Chronologic VCS simulator copyright 1991-2013
Contains Synopsys proprietary information.
```

```
Compiler version H-2013.06; Runtime version H-2013.06;  Jun 24 14:25 2015
----------------------------------------------------------------
UVM-1.1d.VCS
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
----------------------------------------------------------------

UVM_INFO @ 0: reporter [RNTST] Running test i2c_m_base_test...

UVM_INFO tb/i2c_m_bfm.sv(191) @ 0: reporter [I2C_BFM] Initial Values: sda=1 scl=1

UVM_INFO tb/i2c_m_sequence.sv(46) @ 0: uvm_test_top.env.m_agent0.sequencer@@seq [I2C_M_SEQUENCE] Starting body

UVM_INFO tb/i2c_m_driver.sv(54) @ 500000: uvm_test_top.env.m_agent0.driver [I2C_DRIVER] TrType=WRITE | Address
     =1010000 | DataBuff=[0][1][2] | genStop=1 | TrNum=0 | dataLength=3

UVM_INFO tb/i2c_m_bfm.sv(113) @ 500000: reporter [I2C_BFM] Burst Read Write: TrType=WRITE | Address=1010000 |
     DataBuff=[0][1][2] | genStop=1 | TrNum=0 | dataLength=3
UVM_INFO tb/i2c_m_bfm.sv(118) @ 500000: reporter [I2C_BFM] Start condition
UVM_INFO tb/i2c_m_bfm.sv(125) @ 1475000: reporter [I2C_BFM] Send device address and read/write bit
UVM_INFO tb/i2c_m_bfm.sv(43) @ 1475000: reporter [I2C_BFM] Byte Write = 160 = 10100000
UVM_INFO tb/i2c_m_bfm.sv(130) @ 10345000: reporter [I2C_BFM] id_ack
UVM_INFO tb/i2c_m_bfm.sv(134) @ 10345000: reporter [I2C_BFM] Write: Send data buffer
UVM_INFO tb/i2c_m_bfm.sv(43) @ 10475000: reporter [I2C_BFM] Byte Write = 0 = 0
UVM_INFO tb/i2c_m_bfm.sv(43) @ 19475000: reporter [I2C_BFM] Byte Write = 1 = 1
UVM_INFO tb/i2c_m_bfm.sv(43) @ 28475000: reporter [I2C_BFM] Byte Write = 2 = 10

UVM_INFO tb/i2c_m_driver.sv(54) @ 38500000: uvm_test_top.env.m_agent0.driver [I2C_DRIVER] TrType=IDLE | Address=0 |
      DataBuff= | genStop=0 | TrNum=0 | dataLength=0

UVM_INFO tb/i2c_m_bfm.sv(186) @ 38500000: reporter [I2C_BFM] Bus Idle

UVM_INFO tb/i2c_m_driver.sv(54) @ 48500000: uvm_test_top.env.m_agent0.driver [I2C_DRIVER] TrType=WRITE | Address
     =1010000 | DataBuff=[0] | genStop=1 | TrNum=0 | dataLength=1

UVM_INFO tb/i2c_m_bfm.sv(113) @ 48500000: reporter [I2C_BFM] Burst Read Write: TrType=WRITE | Address=1010000 |
     DataBuff=[0] | genStop=1 | TrNum=0 | dataLength=1
UVM_INFO tb/i2c_m_bfm.sv(118) @ 48500000: reporter [I2C_BFM] Start condition
UVM_INFO tb/i2c_m_bfm.sv(125) @ 49475000: reporter [I2C_BFM] Send device address and read/write bit
UVM_INFO tb/i2c_m_bfm.sv(43) @ 49475000: reporter [I2C_BFM] Byte Write = 160 = 10100000
UVM_INFO tb/i2c_m_bfm.sv(130) @ 58345000: reporter [I2C_BFM] id_ack
UVM_INFO tb/i2c_m_bfm.sv(134) @ 58345000: reporter [I2C_BFM] Write: Send data buffer
UVM_INFO tb/i2c_m_bfm.sv(43) @ 58475000: reporter [I2C_BFM] Byte Write = 0 = 0

UVM_INFO tb/i2c_m_driver.sv(54) @ 68500000: uvm_test_top.env.m_agent0.driver [I2C_DRIVER] TrType=READ | Address
     =1010000 | DataBuff=[0][0] | genStop=1 | TrNum=0 | dataLength=2

UVM_INFO tb/i2c_m_bfm.sv(113) @ 68500000: reporter [I2C_BFM] Burst Read Write: TrType=READ | Address=1010000 |
     DataBuff=[0][0] | genStop=1 | TrNum=0 | dataLength=2
UVM_INFO tb/i2c_m_bfm.sv(118) @ 68500000: reporter [I2C_BFM] Start condition
UVM_INFO tb/i2c_m_bfm.sv(125) @ 69475000: reporter [I2C_BFM] Send device address and read/write bit
UVM_INFO tb/i2c_m_bfm.sv(43) @ 69475000: reporter [I2C_BFM] Byte Write = 161 = 10100001
UVM_INFO tb/i2c_m_bfm.sv(130) @ 78345000: reporter [I2C_BFM] id_ack
UVM_INFO tb/i2c_m_bfm.sv(145) @ 78345000: reporter [I2C_BFM] Read: receive data buffer
UVM_INFO tb/i2c_m_bfm.sv(94) @ 86000000: reporter [I2C_BFM] Byte Read = 1 = 1
UVM_INFO tb/i2c_m_bfm.sv(94) @ 95000000: reporter [I2C_BFM] Byte Read = 2 = 10

UVM_INFO tb/i2c_m_sequence.sv(83) @ 97500000: uvm_test_top.env.m_agent0.sequencer@@seq [I2C_M_SEQUENCE] Finished
     body

UVM_INFO tb/i2c_m_base_test.sv(88) @ 97500000: uvm_test_top [I2C_BASE_TEST] Finished run_phase

UVM_INFO /opt/synopsys/H-2013.06/vcs_mx_vH-2013.06/etc/uvm/base/uvm_objection.svh(1268) @ 97500000: reporter [
     TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase

UVM_INFO @ 97500000: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------------
```

```
Name                        Type                    Size  Value
----------------------------------------------------------------
uvm_test_top                i2c_m_base_test         -     @463
  env                       i2c_env                 -     @482
    m_agent0                i2c_m_agent             -     @490
      driver                i2c_m_driver            -     @498
        rsp_port            uvm_analysis_port       -     @513
          recording_detail  uvm_verbosity           32    UVM_FULL
        seq_item_port       uvm_seq_item_pull_port  -     @505
          recording_detail  uvm_verbosity           32    UVM_FULL
        recording_detail    uvm_verbosity           32    UVM_FULL
      sequencer             i2c_m_sequencer         -     @522
        rsp_export          uvm_analysis_export     -     @529
          recording_detail  uvm_verbosity           32    UVM_FULL
        seq_item_export     uvm_seq_item_pull_imp   -     @623
          recording_detail  uvm_verbosity           32    UVM_FULL
        recording_detail    uvm_verbosity           32    UVM_FULL
        arbitration_queue   array                   0     -
        lock_queue          array                   0     -
        num_last_reqs       integral                32    'd1
        num_last_rsps       integral                32    'd1
      recording_detail      uvm_verbosity           32    UVM_FULL
    recording_detail        uvm_verbosity           32    UVM_FULL
----------------------------------------------------------------

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   47
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[I2C_BFM]    26
[I2C_DRIVER]     4
[I2C_M_SEQUENCE]     2
[RNTST]     1
[TEST_DONE]     1
[UVMTOP]     1
[VCS_TR_AUTO]     1
$finish called from file "/opt/synopsys/H-2013.06/vcs_mx_vH-2013.06/etc/uvm/base/uvm_root.svh", line 438.
$finish at simulation time             97500000


           V C S   S i m u l a t i o n   R e p o r t
Time: 1515000000 ps
CPU Time:      0.320 seconds;       Data structure size:   0.3Mb
Wed Jun 24 14:26:11 2015
```

Salida por Consola 3.14: Resultado de la simulación ejecutada en *Synopsys VCSMx*. La salida corresponde a la secuencia de operaciones presentada en esta subsección. En la misma se puede observar el funcionamiento del *Driver*, el BFM y el reporte de la arquitectura del *testbench*.

3.4.4. Análisis

El conocimiento adquirido en la implementación del *testbench* del *buffer* FIFO fue aprovechado para implementar el entorno de verificación de la EEPROM I2C, disminuyendo el tiempo de desarrollo notablemente, ya que la arquitectura utilizada previamente pudo ser re utilizada sin mayores dificultades.

El aumento del nivel de abstracción provisto por la arquitectura facilitó el manejo de la complejidad, permitiendo enfocarse en la funcionalidad de la prueba a nivel de la secuencia sin perderse en los detalles de bajo nivel del protocolo I2C.

En un desarrollo futuro se planea incluir los componentes de análisis al *testbench*, es decir un Monitor y un *Scoreboard* que contenga un modelo de referencia de la memoria implementado mediante un arreglo asociativo re utilizando la arquitectura del *testbench* del *buffer* FIFO y la clase item utilizado por el entorno de verificación actual.

Para controlar el correcto funcionamiento del protocolo y asegurar que el mismo sea cubierto en su totalidad se plantea el desarrollo de un componente, que verifique el protocolo mediante el uso de aserciones de *SystemVerilog*, conocido como *Protocol Checker* el cual estaría implementado en la interfaz del *testbench*.

El sistema desarrollado se puede utilizar para estimular cualquier dispositivo I2C, dando un soporte de alto nivel a las pruebas derivadas de modelo de cobertura utilizada, permitiendo abstraerse de los detalles del protocolo, y enfocarse en las funcionalidades que se desean probar.

Capítulo 4

Desarrollo de un Modelo de Cobertura Funcional

En este capítulo se presenta la problemática del crecimiento del espacio de prueba de los diseños modernos y la subsecuente dificultad para generar modelos de cobertura adecuados para los mismos a partir de sus especificaciones. En función de esto, se introduce un método empírico de caja negra para generar un modelo de cobertura funcional para diseños dominados por el control. Este método está basado en la utilización de un modelo abstracto del DUV para facilitar la extracción de conjuntos de secuencias de prueba, los cuales definen un modelo de cobertura funcional. Dada la complejidad de los posibles espacios de prueba generados, las conocidas técnicas de *Testing de Software*, partición en clases de equivalencia y análisis de valores limites, son aplicadas para reducirlos. Adicionalmente, para expresar las secuencias de pruebas equivalentes extraídas del modelo, se desarrolla una notación formal. Este método es aplicado para la verificación del módulo de *buffer* FIFO presentado en el capítulo anterior, utilizando el *testench* desarrollado previamente. Finalmente, se analizan los puntos más relevantes de la aplicación de esta técnica.

4.1. Problemática

Como se mencionó en la introducción de este trabajo, la verificación funcional es hoy en día el cuello de botella del flujo de diseño digital y la simulación de eventos discretos sigue siendo la técnica de verificación más utilizada, principalmente debido a que es aplicable a sistemas grandes y complejos [26]. El espacio de posibles estímulos para diseños modernos presenta una complejidad que lo hace inviable de cubrir de forma exhaustiva dentro de un cronograma de verificación típico, en consecuencia, es fundamental identificar los patrones de estímulos clave

para ejercitarlos de forma eficiente.

En la mayoría de los diseños modernos se desprenden requerimientos funcionales que definen un espacio de cobertura que esta más allá de la capacidad de simulación en un tiempo realista. Esto se debe a que ejercitar exhaustivamente, inclusive un dispositivo de tamaño modesto, puede requerir de un numero exorbitante de vectores de prueba. Por ejemplo, si un dispositivo tiene N entradas y M *flip-flops*, según la complejidad del mismo, se pueden llegar a requerir $(2^N)^M$ vectores de prueba para ejercitarlo completamente. Un dispositivo modesto puede tener 10 entradas y 100 *flip-flops* (un poco menos de tres registros de 32-bits). Este dispositivo requeriría $(2^{10})^{100}$ o 2^{1000} vectores para ejercitarlo completamente. Si quisiéramos simular este dispositivo a 1000 vectores por segundo, esto tomaría aproximadamente $3{,}4 \times 10^{308}$ años.

Para lidiar con esta problemática, las tendencias actuales en verificación emplean un modelo de cobertura funcional [35] para medir la calidad y orientar el progreso del proceso de verificación. En la técnica de verificación dirigida por la cobertura el objetivo principal es implementar un conjunto de pruebas para acertar el máximo número de puntos de cobertura con el menor esfuerzo. Las combinaciones y secuencias de vectores de entrada son diseñados con el objetivo de verificar por completo la funcionalidad del diseño con un aceptable grado de confianza. Esta completitud se evalúa mediante el diseño de vectores de entrada para estimular el DUV y registrando automáticamente su impacto sobre ciertas metas o métricas de cobertura.

Los casos de prueba se pueden generar de forma dirigida o con rutinas aleatorias. En la generación de estímulos dirigidos, las secuencias de prueba están especialmente diseñadas para ejercitar cada característica que tiene que ser probada. La debilidad de esta técnica reside en que el ingeniero de verificación debe utilizar su intuición para dirigir la búsqueda de posibles errores y por lo tanto se le puede pasar por alto controlar la existencia de algunos errores poco frecuentes o difíciles de intuir debido a que los mismos surgen a causa de la ocurrencia de múltiples eventos paralelos difíciles de predecir por una persona. El paradigma de Programación Imperativa de los lenguajes utilizados para implementar *testbenches* resulta adecuado para codificar este último tipo de pruebas.

Por otro lado, en la generación de estímulos aleatorios con restricciones se generan secuencias de prueba automáticamente para satisfacer varias restricciones definidas a partir de las especificaciones del diseño. Los HVLs modernos, como SV, se basan en el paradigma de Programación Declarativa para expresar estas restricciones de forma natural y generar automáticamente soluciones válidas sin especificar exactamente cómo encontrarlas. Este enfoque es más adecuado para generar secuencias de prueba poco probables, pero válidas, las cuales pueden ser difícil de predecir por el ingeniero de verificación en un esquema de pruebas dirigidas. Sin embargo, a menudo es difícil establecer correcta y eficientemente las restricciones con el fin de generar un número mínimo de secuencias de prueba.

Para sistemas complejos, la resolución de estos grandes conjuntos de restricciones se ha convertido en una limitante en el proceso de simulación debido a la gran cantidad de tiempo dedicado a la resolución de los mismos. En consecuencia, es común que una gran cantidad de tiempo se dedique a depurar y buscar formas de solucionar esta limitación. En el peor de los casos, es necesario descartar la generación de pruebas aleatorias utilizando el paradigma declarativo y reescribir la aleatorización mediante código imperativo, perdiendo así la productividad ganada. [20].

Dependiendo de las características del DUV, el enfoque dirigido o el aleatorio con restricciones puede ser preferible, o incluso una combinación de ambos. Esto último se desprende de que es útil dirigir la generación de pruebas hacia cierta región del espacio de cobertura y luego contar con aleatoridad para moverse dentro de dichas regiones, todo esto teniendo una adecuada visibilidad del proceso. Esto es difícil de lograr cuanto se trabaja con la generación de estímulos aleatorios con restricciones tradicionales, donde muchas veces es difícil dirigir las pruebas hacia ciertas regiones.

Un modelo de cobertura de buena calidad facilita la generación de un conjunto de secuencias de prueba mínimo, capaz de ejecutar condiciones complejas y poco comunes, evitando la generación de secuencias replicadas, pero logrando un alto nivel de cobertura. Por otro lado, la definición de un modelo de cobertura funcional completo y correcto es, a menudo, difícil de obtener directamente a partir de la especificaciones. Esto es, para muchos diseños es difícil de visualizar y extraer comportamientos y escenarios de uso directamente a partir de sus especificaciones.

4.2. Técnicas de Testing de Software

La industria del software proporciona una base solida para realizar pruebas de software, lo cual es útil para realizar verificación funcional de hardware. La similitud entre la sintaxis de los lenguajes de programación y los de definición de hardware, como VHDL o *Verilog*, permite aplicar la mayor parte de las técnicas que involucran cobertura de código fuente. Estas técnicas, junto a las metodologías de caja negra, las cuales examinan la funcionalidad de una aplicación sin acceder a su estructura interna, pueden ser aplicadas a una unidad de hardware digital, ya que desde el punto de vista de Ingeniería en Sistemas y *Testing de Software*, la misma es equivalente a un componente de software.

El *testing* de caja negra [33] se puede aplicar en cualquier nivel de *testing*, unidad, integración, sistema y es a menudo una buena técnica para utilizar en primera instancia. Debido a que este tipo de pruebas pueden ser construidas a partir de las especificaciones del diseño, antes de la etapa de implementación, ayudan a detectar tempranamente errores en el flujo de trabajo,

cuando la corrección es menos costosa. Las técnicas de caja negra más consolidadas se basan en la partición del espacio de prueba en clases de equivalencia [32, 23, 48] y en la derivación de pruebas a partir de modelos abstractos que representan la funcionalidad del diseño [25, 21, 41].

4.2.1. Clases de Equivalencia y Análisis de Valores Limites

El concepto de partición en clases de equivalencia propone dividir los rangos de datos de entrada y de salida de una unidad de software en particiones o clases de tal manera que todos los elementos contenidos en cada subrango sean tratados por el componente de la misma manera. En principio, los casos de prueba son diseñados para cubrir cada partición al menos una vez, comprobando secciones equivalentes del componente bajo prueba y detectando clases de errores, reduciendo así el tiempo consumido por las pruebas debido al uso de un número mínimo de casos de prueba. La técnica de clases de equivalencia se combina con el análisis de valores límites para definir los casos de prueba en los límites entre las particiones, a menudo representado por los valores mínimos y máximos para cada región de equivalencia.

4.2.2. Pruebas basadas en Modelos

La técnica de prueba basada en modelos (*Model-based Testing* [47]) utiliza modelos abstractos que capturan la funcionalidad del diseño para ayudar en la derivación de casos de prueba concretos. El nivel de detalle para cada modelo funcional debe ajustarse en función de los objetivos de la verificación, siendo el ingeniero en verificación el responsable de identificar los aspectos relevantes del DUV que necesitan ser ejercitados. Dado que los conjuntos de pruebas se derivan de modelos y no de código fuente, las pruebas basadas en modelos por lo general se ven como una forma de técnica de caja negra. Este enfoque también permite automatizar la generación de los conjuntos. Es por ello, que generalmente se requieren modelos lo suficientemente formales, de manera que permitan usar un algoritmo que derive los casos de prueba a partir de ellos (la derivación manual de los casos de prueba a partir de modelos formales también se considera *Model-based Testing*).

En cuanto a la derivación de casos de prueba a partir de un modelo abstracto de la funcionalidad del diseño, una metodología bien conocida es la prueba de transición de estados [5]. El comportamiento de los DUV se representa con máquinas de estados finito y mediante el análisis de la combinación de eventos que hacen que el modelo evolucione, se obtienen condiciones de prueba eficaces.

4.3. Técnica empírica para derivación de vectores de prueba

Se ideó una técnica empírica para CDV con el fin de generar un conjunto significativo y reducido de secuencias de prueba para la verificación de caja negra de un diseño dominado por el control. Esta técnica simplifica la creación de un modelo de cobertura funcional mediante la utilización de un modelo abstracto de la funcionalidad del DUV, el cual facilita la extracción de secuencias de prueba. La metodología presentada pretende aumentar el nivel de abstracción en el diseño de modelos de cobertura funcional permitiendo enfocarse en la identificación de secuencias útiles de operación que determinan el comportamiento del sistema, en lugar de modelar de forma tradicional cada una de las funcionalidades básicas definidas por las especificaciones.

En [36] la funcionalidad del DUV presentado es modelada con una Maquina de Estados Finitos Extendida (EFSM, sigla en inglés de *Extended Finite State Machine*) [10]. El mismo modelo de abstracción es utilizado en la técnica desarrollada en esta sección. Siendo ésta una metodología de caja negra, los estados y transiciones relevantes de la EFSM tienen que ser inferidos intuitivamente a partir de las especificaciones del DUV. Este tipo de modelo es apropiado para diseños dominados por el control (*control-dominated designs*). Un diseño dominado por el control es uno en el que la complejidad del hardware resultante está más relacionada a operaciones de entrada/salida (I/O, sigla en inglés de *Input/Output*) y accesos a memoria que con computación. Un diseño dominado por el flujo de datos (*datapath-dominated design*) se podría definir como uno en el que la característica dominante es la transformación de datos, o computación, en el cual la dinámica del sistema consiste principalmente en la generación de nuevos valores de salida a partir de la transformación de los valores de entrada. Un diseño dominado por el control está más abocado a pasar valores de un lugar a otro. Una definición común de un diseño dominado por el control es uno que se modela mediante máquinas de estado finito que interactúan entre sí. Una definición equivalente es un diseño compuesto por procesos concurrentes que se comunican entre sí. El mecanismo clave de estos diseños es el mecanismo de comunicación o protocolo. Como una cuestión practica, un diseño dominado por el control es uno dominado por sus protocolos de comunicación con uno o más procesos (ya sea internos o externos) [18].

Los errores más difíciles de detectar en los diseños dominados por el control dependen de la historia del DUV o son el resultado de múltiples eventos lógicos de control [22]. Estos errores son difíciles de descubrir usando pruebas dirigidas. Los test aleatorios pueden encontrar estos errores, pero cada una de estas condiciones es tan improbable, que encontrar un error que ocurra en la conjunción de estos casos o que dependa de casos previos, requiere de un tiempo de simulación prohibitivo.

En un sistema basado en control, el conjunto de operaciones disponibles varía acorde al

estado en el que se encuentre el sistema. Este tipo de sistemas son especificados más fácilmente usando notaciones basadas en transiciones como por ejemplo las máquinas de estado.

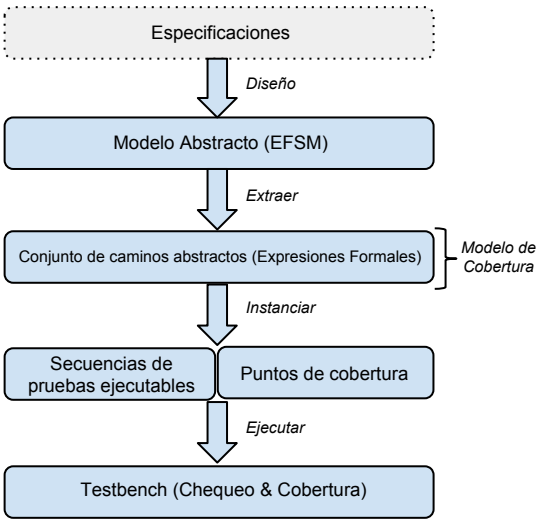

Figura 4.1: Los cuatro pasos básicos de la técnica presentada

La técnica presentada consiste de los siguientes cuatro pasos (ver Figura 4.1) que se basan en la metodología de *Model-based Testing* y en las técnicas de clases de equivalencia y análisis de valores limites:

1. El primer paso se basa en capturar la funcionalidad del diseño modelándolo mediante una EFSM.

2. El segundo paso consiste en elaborare un criterio de selección de pruebas con el fin de identificar secuencias relevantes de transiciones (caminos) a través de la máquina de estados, las cuales representan pruebas abstractas. Según el criterio de selección, pruebas equivalentes serán agrupadas en conjuntos definiendo clases de equivalencias. Este grupo de conjuntos de caminos abstractos conforman el modelo de cobertura funcional.

3. El tercer paso se basa en transformar las pruebas abstractas en pruebas ejecutables y sus correspondientes puntos de cobertura. Es decir, seleccionar un número adecuado de representantes de cada clase de equivalencia.

4. El último paso consiste en ejecutar los casos de prueba mediante un *testbench* que compruebe automáticamente el correcto funcionamiento del DUV y recolecte la información de cobertura.

En las siguientes subsecciones se describe con mayor detalle cada uno de los pasos de la técnica presentada.

4.3.1. Modelo Funcional Abstracto

El primer paso de la técnica se basa en desarrollar, a partir de las especificaciones, un modelo que capture la funcionalidad del sistema usando una EFSM. Teniendo en cuenta que se trata de una técnica de caja negra, el modelo tiene que ser creado basado en las especificaciones del diseño. La independencia entre este modelo y la implementación o visión del diseñador facilita la detección de errores. Adicionalmente, analizar las especificaciones de forma separada tiene el beneficio de encontrar fallas en los documentos de requerimientos y diseño en etapas tempranas del desarrollo.

Este modelo abstracto tiene que enfocarse sólo en los aspectos clave que deben ser verificados y debe omitir muchos de los detalles del DUV, como su estructura y características a nivel RTL. Debe incluir suficiente información para identificar cada operación válida, así como cualquier posible operación inválida que pueda ocurrir durante la operación del DUV, de forma de determinar qué secuencias verifican la funcionalidad y la robustez del diseño. Este tipo de análisis puede ser útil para completar las especificaciones funcionales cuando no hayan sido elaboradas con el nivel apropiado de detalle.

4.3.2. Pruebas Abstractas y Modelo de Cobertura

El espacio de cobertura se puede ahora analizar por medio del modelo abstracto. Se requiere de un criterio de selección de pruebas para decidir que caminos se generan a partir del modelo, ya que por lo general hay un número infinito de caminos posibles.

El principal resultado de este paso es un conjunto de pruebas abstractas que son secuencias de operaciones para evolucionar el DUV. Este conjunto representa el modelo de cobertura. Dado que el modelo utiliza una vista simplificada del DUV, estas secuencias abstractas carecen de algunos detalles que necesita el DUV por lo que no son directamente ejecutables.

La cobertura completa de transiciones (en inglés, *Full Transition Coverage*) es un buen mínimo para aspirar cuando se generan pruebas a partir de modelos de transición como EFSMs. Un recorrido de transiciones (en inglés, *Transition Tour*) es un recorrido circular de longitud mínima a través del modelo que llega a ejercitar todas las transiciones al menos una vez. Es muy posible que una secuencia particular de tres o más transiciones expongan un fallo del DUV, pero esa secuencia particular puede no ser ejercitada durante el *Transition Tour*. Es por eso que caminos explícitos deben ser extraídos del modelo basado en las especificaciones del dispositivo y la experiencia del ingeniero. Caminos equivalentes deben ser agrupados en clases equivalentes

y los casos límites deben ser tomados en cuenta.

A pesar de que el espacio de cobertura también puede implicar caminos de longitud infinita a través de la máquina de estados, para la los efectos de la verificación, secuencias de prueba de este estilo deben ser definidas con bucles finitos.

Las pruebas abstractas se definen como caminos a través de la EFSM expresados utilizando una notación formal. Dicha notación formal se explica a continuación utilizando la EFSM de la Figura 4.2 como ejemplo.

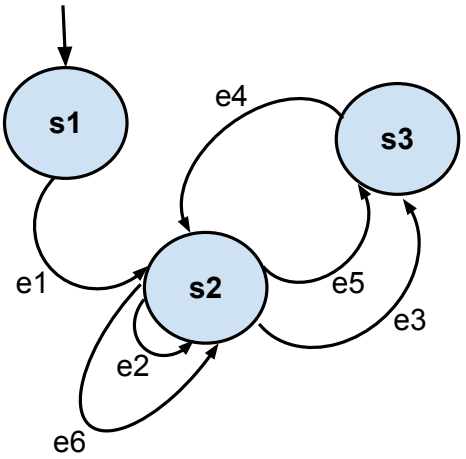

Figura 4.2: EFSM de ejemplo.

Un camino P es una tupla ordenada $i_1, i_2 \dots i_n$ donde cada elemento i es un operando $s.e$ formado por un nodo (o estado) s y un arco (o evento) e saliendo de ese nodo en la EFSM. El operador de concatenación "." se puede omitir para mayor claridad. Por ejemplo:

$$P_1 = (i_1, i_2, i_3) = ((s_1.e_1), (s_2.e_3), (s_3.e_4)) = (s_1e_1, s_2e_2, s_3e_4)$$

Los operadores son "*:" y "|". Este último, "|", es el operador de selección exclusiva para los posibles eventos de un ítem tal como se ve en el siguiente ejemplo:

$$P_2 = (s_1e_1, s_2(e_3|e_5), s_3e_4)$$

$$P_2 => (s_1e_1, s_2e_3, s_3e_4) \ o \ (s_1e_1, s_2e_5, s_3e_4)$$

Los ciclos de longitud finita en los caminos pueden ser formalmente expresados usando el operador denominado *Estrella Acotada "*:"*, con significado similar a la *Estrella de Kleene* utilizada en expresiones regulares. La *Estrella Acotada* es usada como un operador *postfix* indicando que el evento del ítem que modifica puede ser ejecutado un número acotado de veces de acuerdo a ciertas restricciones basadas en una o más inecuaciones lineales. Las variables en las inecuaciones representan el número de veces que cada evento e del ítem al cual el operador es aplicado puede ser ejecutado. Las variables son definidas usando el símbolo # para representar su cantidad, siendo los valores de las mismas siempre enteros mayores o iguales a cero. Esto se muestra en el siguiente ejemplo:

$$P_3 = (s_1e_1, s_2e_2^{*:(\#e_2<3)}, s_2e_5, s_3e_4)$$

$$P_3 => (s_1e_1, s_2e_5, s_3e_4)\ o\ (s_1e_1, s_2e_2, s_2e_5, s_3e_4)\ o\ (s_1e_1, s_2e_2, s_2e_2, s_2e_5, s_3e_4)$$

La *Estrella Acotada "*:"* también se puede usar en conjunto con el operador "|". En este caso la inecuación puede definir el número de veces que cada operando puede ser ejecutado en relación con el otro. Por ejemplo, a continuación se observa P_4, en el cual todos los caminos generados tendrán un evento e_2 más que e_6:

$$P_4 = (s_1e_1, s_2(e_2|e_6)^{*:(0<\#e_2-\#e_6<2)}, s_2e_3, s_3e_4)$$

$$P_4 => (s_1e_1, s_2e_2, s_2e_3, s_3e_4)\ o\ (s_1e_1, s_2e_2, s_2e_2, s_2e_6, s_2e_3, s_3e_4)\ o$$
$$(s_1e_1, s_2e_6, s_2e_2, s_2e_2, s_2e_3, s_3e_4)\ o\ ...$$

El modificador $\forall$ puede ser usado para restringir aun más la *Estrella Acotada "*:"* cuando ésta es usada en conjunto con el operador "|". El modificador $\forall$ es usado para establecer una relación de orden entre el número de operandos que son creados en cada una de las subsecuencias generadas por la concatenación de cada iteración del operador "**:*". La relación de orden es expresada usando una inecuación lineal que involucra los operandos a los cuales este operador es aplicado. Por ejemplo:

$$P_5 = (s_1e_1, s_2(e_2|e_6)^{*:\forall(\#e_2>\#e_6)}, s_2e_3, s_3e_4)$$

La aplicación de este operador puede ser descrita enumerando el conjunto de subsecuencias que pueden ser generadas respetando la inecuación. Eso se logra concatenando en cada paso un operando valido a la subsecuencia del paso anterior, respetando en cada caso la restricción establecida por el modificador ∀. Siguiendo con P_5:

1. (s_2e_2) donde $(\#e_2 = 1 > 0 = \#e_6)$
2. (s_2e_2, s_2e_2) donde $(\#e_2 = 2 > 0 = \#e_6)$
3. (s_2e_2, s_2e_2, s_2e_6) donde $(\#e_2 = 2 > 1 = \#e_6)$
4. $(s_2e_2, s_2e_2, s_2e_6, s_2e_2)$ donde $(\#e_2 = 3 > 1 = \#e_6)$
5. ... donde $(\#e_2 > \#e_6)$

Luego, uno de los posibles caminos que pueden ser extraídos de P_5 usando la subsecuencia generada en el paso 4) es:

$$(s_1e_1, s_2e_2, s_2e_2, s_2e_6, s_2e_2, s_2e_3, s_3e_4)$$

El modificador @ puede ser usado para restringir aun más la *Estrella Acotada "*:"* cuando éste es usado en conjunto con el operador "|". El modificador @ es usado para establecer una relación de orden entre el número de operandos que son creados en la secuencia final generada por la concatenación de cada iteración del operador "**:*". La relación de orden es expresada usando una inecuación lineal que involucra los operandos a los cuales se le aplica este operador.

El modificador ∀ puede ser usado en conjunto con el modificador @ para describir un conjunto de restricciones. El modificador ∀ define una restricción para el número de veces que cada variable puede ser generada en cada subsecuencia, forzando a que la secuencia se mantenga en un determinado estado. El operador @ establece la condición final que la secuencia debe cumplir, forzando a la secuencia a evolucionar hasta cierto estado al final de la misma. Por ejemplo:

$$P_6 = (s_1e_1, s_2(e_2|e_6)^{*:\{\forall(0<\#e_2-\#e_6<2)@(\#e_2>\#e_6)\}}, s_2e_3, s_3e_4)$$

El formalismo presentado puede ser utilizado para expresar clases de equivalencia de caminos. Cada expresión formal puede representar un conjunto de caminos que comparten cierto criterio. Esto es modelado a través de restricciones definidas por las inecuaciones usadas. El uso de los valores limites permitidos por las inecuaciones permite derivar un conjunto de caminos frontera para esa clase de equivalencia.

4.3.3. Pruebas Ejecutables y Puntos de Cobertura

El tercer paso consiste en transformar las pruebas abstractas en pruebas concretas (ejecutables) y sus correspondientes puntos de cobertura. Para cada clase de caminos equivalentes expresada el en formalismo presentado, un número determinado de representantes son seleccionados aleatoriamente. Lo mismo se hace para cada camino límite. De cada representante se extrae una secuencia de transiciones teniendo en cuenta sólo los eventos, descartando los nodos. Esta lista de eventos será la secuencia ejecutable inyectada por el *testbench*.

Cada tipo de evento es mapeado directamente a una clase de ítem de UVM, permitiendo una forma natural de expresar las secuencias de prueba a nivel de transacción mediante secuencias de UVM. Estas secuencias son inyectadas al DUV a través del *testbench*, el cual mide la cobertura funcional lograda por cada secuencia y verifica automáticamente el comportamiento del DUV. La medición de la cobertura de los caminos asegura que las secuencias ejecutables ejerciten efectivamente las pruebas tal cual se definieron, brindando una redundancia adicional, ya que se controla el comportamiento esperado de la secuencia inyectada mediante la cobertura de su respectivo camino. De esta manera el razonamiento y diseño de alto nivel de las pruebas proviene del ingeniero, mientras que los detalles de bajo nivel de las pruebas y la salida esperada del DUV provienen del *testbench*.

Finalmente, luego de cada corrida en la que se encuentren errores, el DUV debe ser revisado y corregido de acuerdo a los errores detectados, para luego volver a verificarse.

4.4. Caso de Estudio: Módulo de Buffer FIFO

En lo que resta del capítulo se presenta la aplicación de la técnica introducida a un diseño puntual, el módulo *buffer* FIFO utilizado en el capítulo anterior. El objetivo es generar un modelo de cobertura funcional adecuado para dicho diseño utilizando la técnica desarrollada. En función de esto, se describe la aplicación de cada paso de la técnica y las principales decisiones tomadas en el proceso.

Para poder experimentar con la técnica propuesta se plantea una metodología de trabajo iterativa, la cual cicla entre una etapa de diseño, a cargo de un tercero, y otra de verificación. Esto es, se comienza verificando una primera versión del DUV con la técnica propuesta. Luego, los errores que se encuentren, se reportan al diseñador, el cual corrige el DUV acorde a las fallas reportadas, generando una nueva versión del diseño. Posteriormente, se verifica la nueva version del DUV y se vuelve a repetir el ciclo de diseño y verificación. El proceso termina cuando la técnica de verificación no encuentra más fallas.

El módulo *buffer* FIFO (ver las especificaciones detalladas en la Sección 3.3.1) se elige como

caso de prueba debido a que es un diseño dominado por el control que, como se mencionó en el capítulo anterior, es rico en términos de posibles secuencias de pruebas que se le pueden inyectar para su verificación. Para la aplicación de la técnica, se instancia el *buffer* FIFO para que tenga la capacidad de almacenar hasta 8 elementos, es decir, $MAX = 8$. El módulo *buffer* FIFO es un diseño dominado por el control que puede ser modelado efectivamente mediante una máquina de estados, por lo cual es un ejemplo adecuado para la aplicación de la técnica desarrollada.

El *testbench* implementado mediante SV y UVM en el capítulo anterior (subsección 3.3.2) es utilizado como plataforma para implementar eficientemente los casos de pruebas y puntos de cobertura generados utilizando la técnica propuesta. En linea con esto, en el presente capítulo se introduce la clase *fifo_coverage* del *testbench* mencionado, la cual implementa el sistema de medición de cobertura funcional para el modelo generado con la técnica propuesta.

En las siguientes subsecciones se explica en detalle cómo se aplicaron los distintos pasos de la técnica al caso de estudio propuesto, analizando finalmente las cuestiones más relevantes de la experiencia.

4.4.1. Modelo Funcional Abstracto

Siguiendo la técnica de verificación de caja negra, el modelo de referencia de la FIFO se implementó como la EFSM presentada en la Figura 4.3, la cual tiene cuatro estados: *UNDEFINED, EMPTY, MIDDLE* y *FULL*. Éstos representan todos los posibles estados que la FIFO puede adoptar basado en la codificación de las señales de estado del DUV como se muestra en la Tabla 4.1. Los estados fueron elegidos de esta manera para definir clases de equivalencia. El estado *MIDDLE* representa cuando la FIFO tiene más de un elemento y menos que *MAX*, incluyendo todas las posibilidades intermedias. Así se reduce el número de estados de la EFSM y se simplifica la posterior generación de casos de prueba. Adicionalmente, los estados *EMPTY* y *FULL* se pueden ver como valores límites de la clase de equivalencia *MIDDLE*. El estado *UNDEFINED* modela el DUV antes de que sea inicialmente reseteado, es decir, las señales *full* y *empty* están aun indefinidas.

Estado	**empty**	**full**
UNDEFINED	U	U
EMPTY	1	0
MIDDLE	0	0
FULL	0	1

Tabla 4.1: Los estados de la FIFO codificados basados en las señales de *full* y *empty*.

Las transiciones entre los estados son las distintas operaciones que se pueden realizar so-

bre la FIFO: *Reset*, *Write*, *Read*, *Write&Read* como se ven codificadas en la Tabla 4.2. Las transiciones inválidas son modeladas en rojo en la EFSM.

Operación	rst	rd_en	wr_en
Reset	1	*	*
Write	0	0	1
Read	0	1	0
Write&Read	0	1	1

Tabla 4.2: Las operaciones de la FIFO codificadas según las señales *rst*, *rd_en* y *wr_en*.

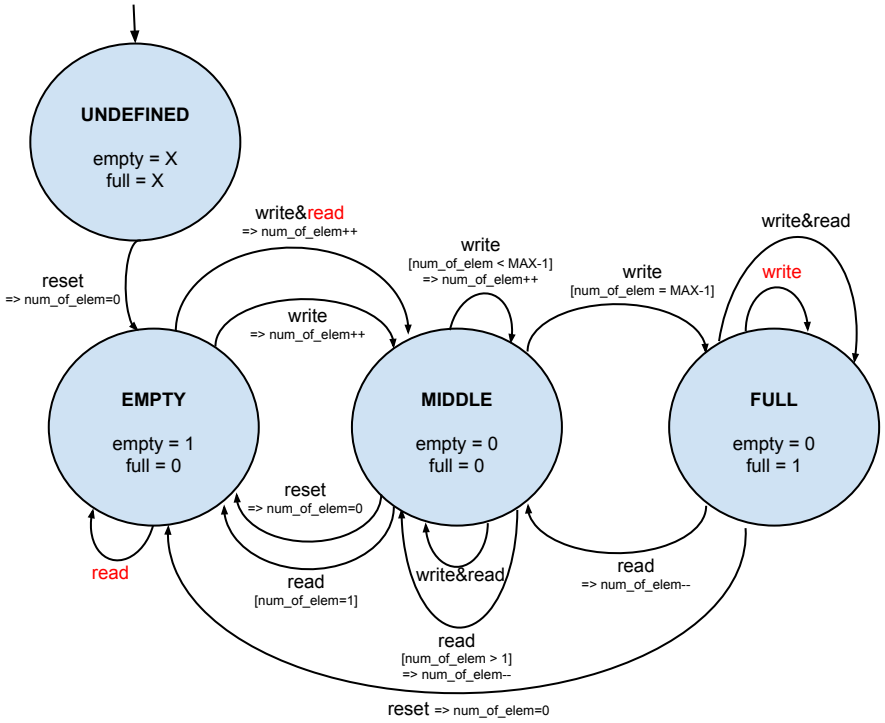

Figura 4.3: EFSM de la FIFO con tres clases de equivalencia: EMPTY, MIDDLE y FULL. La variable de condición *num_of_elem* representa la cantidad de elementos almacenados en la FIFO.

4.4.2. Pruebas Abstractas y Modelo de Cobertura

El formalismo definido previamente fue utilizado para expresar un conjunto significativo de clases de equivalencia de caminos a través de la EFSM de la FIFO. Las clases fueron seleccionadas

basadas en las especificaciones del diseño, el funcionamiento esperado y los posibles casos límites (*corner cases*). El conjunto de clases de equivalencia (ver Tabla 4.3) representan el modelo de cobertura de la FIFO, donde P_{EM} representa una clase de equivalencia de caminos entre el estado EMTPY y el estado MIDDLE, y P_{MF} representa una clase de equivalencia de caminos entre el estado MIDDLE y el estado FULL. Los caminos equivalentes con *min* y *max* en sus nombres representan caminos frontera derivados de P_{EM} y P_{MF}. Por otro lado, P_{Urst}, P_{Erst}, P_{Mrst} y P_{Frst} representan caminos equivalentes usados para probar la operación de *Reset* en los respectivos estados.

En la Figura 4.4 se muestra una extensión del modelo abstracto presentado anteriormente. En el mismo se expande la clase MIDDLE con el objetivo de mostrar los valores límites de la misma, el valor límite *Valor_Límite_EM* (casi vacío) y el valor límite *Valor_Límite_MF* (casi lleno). A continuación se describen los aspectos más relevantes de las pruebas abstractas presentadas en la Tabla 4.3 utilizando este último modelo para facilitar la explicación:

- La clase P_{EM} modela los caminos entre el estado EMPTY y el estado MIDDLE. Con $\forall(-1 \leq \#w - \#r < MAX - 1)$ se evita que la secuencia pase al estado FULL. Con $@(\#r > \#w)$ se fuerza que al finalizar la secuencia se vuelva al estado EMPTY.

- La clase P_{EMmin} ejercita el valor límite EM (casi vacío) pasando directamente del estado EMPTY al MIDDLE y de vuelta al EMPTY, sin realizar ninguna operación en el estado MIDDLE.

- La clase P_{EMmax} ejercita el valor límite MF (casi lleno). Con $\forall(0 \leq \#w - \#r < MAX - 1)$ del primer $M(r|w|wr)$ se evita que la secuencia pase al estado FULL. Con $@(\#w - \#r = MAX - 2)$ se fuerza que al finalizar la secuencia se evolucione al valor limite MF (casi lleno). Luego, en el segundo $M(r|w|wr)$ se utiliza $@(\#r - \#w = MAX - 1)$ para forzar que se vuelva al estado EMPTY.

- La clase P_{MF} modela los caminos entre el estado MIDDLE y el estado FULL. Con $\forall(0 \leq \#w - \#r < MAX)$ se evita que se vuelva al estado EMPTY. Con $@(\#w - \#r = MAX - 1)$ se fuerza que al finalizar la secuencia se evolucione al estado FULL. Esta clase no realiza operaciones sobre el estado MIDDLE al volver del estado FULL porque se asume que esas operaciones fueron cubiertas por la clase P_{EMmax}.

- La clase P_{MFmin} ejercita el valor límite MF (casi lleno) pasando directamente del estado MIDDLE al FULL y de vuelta al FULL, sin realizar ninguna operación en el estado FULL. Los modificadores del $M(r|w|wr)$ son análogos a los utilizados en el primer $M(r|w|wr)$ de la clase anterior, de forma de evitar volver al estado EMPTY y forzar que al finalizar la secuencia se evolucione al estado FULL.

Modelo de cobertura del *buffer* FIFO
$P_{EM} = (Urst, Er^{*:(\#r\geq 0)}, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(-1 \leq \#w - \#r < MAX-1) @(\#r > \#w), (\#wr \geq 0)\}}, Er^{*:(\#r\geq 0)})$
$P_{EMmin} = (Urst, E(w\|wr), Mr)$
$P_{EMmax} = (Urst, Er^{*:(\#r>0)}, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX-1) @(\#w - \#r = MAX-2), (\#wr > 0)\}}, M(r\|w\|wr)^{*:\{\forall(0 \leq \#r - \#w < MAX) @(\#r - \#w = MAX-1), (\#wr > 0)\}}, Er^{*:(\#r>0)})$
$P_{MF} = (Urst, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX), @(\#w - \#r = MAX-1), (\#wr \geq 0)\}}, F(w\|wr)^{*:\{(\#w\geq 0),(\#wr\geq 0)\}}, Fr)$
$P_{MFmin} = (Urst, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX), @(\#w - \#r = MAX-1), (\#wr \geq 0)\}}, Fr))$
$P_{MFmax} = (Urst, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX), @(\#w - \#r = MAX-1), (\#wr \geq 0)\}}, F(w\|wr)^{*:\{(\#w>0),(\#wr>0)\}}, Fr)$
$P_{Urst} = (Urst)$
$P_{Erst} = (Urst, Er^{*:(\#r\geq 0)}, Erst)$
$P_{Mrst} = (Urst, Er^{*:(\#r\geq 0)}, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX-1), @(\#w \geq \#r), (\#wr \geq 0)\}}, Mrst)$
$P_{Frst} = (Urst, Er^{*:(\#r\geq 0)}, E(w\|wr), M(r\|w\|wr)^{*:\{\forall(0 \leq \#w - \#r < MAX), @(\#w - \#r = MAX-1), (\#wr \geq 0)\}}, Frst)$

Tabla 4.3: Conjunto de secuencias de prueba abstractas que definen el modelo de cobertura de la FIFO.

- La clase P_{MFmax} llega al estado FULL, donde realiza al menos una operación de *Write* o *Write&Read* y luego vuelve al estado MIDDLE, ejercitando el estado MF (casi lleno).
- La clase P_{Mrst} ejercita la operación *Reset* en el estado MIDDLE. Con $\forall(0 \leq \#w - \#r < MAX-1)$ se evita que se pase al estado FULL. Con $@(\#w \geq \#r)$ se fuerza que al finalizar la secuencia se quede en el estado MIDDLE.

4.4.3. Implementación de Pruebas Ejecutables y Puntos de Cobertura

Después que el modelo de cobertura fue definido, un número de secuencias de pruebas concretas fueron seleccionadas como representantes de cada clase. Las mismas se presentan en la Tabla 4.4.

Las respectivas pruebas ejecutables fueron implementadas en la clase *fifo_sequence* utilizando los métodos *set* de la clase *fifo_item* tal como se ejemplifica en la Sección 3.3.2.7.

La clase *fifo_coverage*, la cual se muestra en el Código 4.1, cuenta con los mecanismos necesarios para llevar a cabo la medición de la cobertura funcional a partir de los *fifo_items* recibidos provenientes del Monitor. Para esto se implementa la función *write*, definida en la linea 112, la cual es llamada por el Monitor cada vez que le enviá un *fifo_item* al medidor de cobertura. A su vez, esta función llama al método *sample*, definido en la linea 68, el cual se

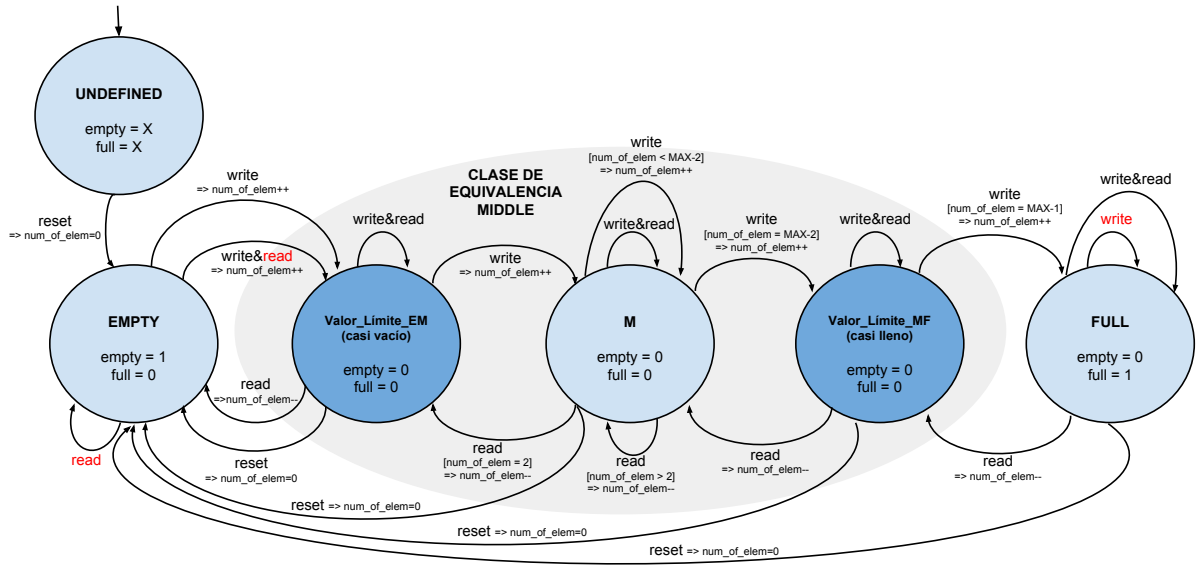

Figura 4.4: EFSM de la FIFO con la clase de equivalencia MIDDLE expandida para mostrar los valores límites, casi vacío y casi lleno.

P_{EMmin1}	$: rst, w, r$
P_{EMmin2}	$: rst, wr, r$
P_{EM1}	$: rst, w, wr, r$
P_{EM2}	$: rst, w, w^{*:(\#w=MAX-2)}, wr, r^{*:(\#r=MAX-1)}$
P_{EM3}	$: rst, w, w^{*:(\#w=2)}, wr, r^{*:(\#r=2)}$
P_{EMmax1}	$: rst, r, w, w^{*:(\#w=MAX-2)}, r^{*:(\#r=MAX-1)}, r$
P_{MFmin1}	$: rst, w, w^{*:(\#w=MAX-1)}, r$
P_{MF1}	$: rst, w, w^{*:(\#w=MAX-1)}, wr, r$
P_{MF2}	$: rst, w, w^{*:(\#w=MAX-1)}, w, r$
P_{MFmax1}	$: rst, w, w^{*:(\#w=MAX-1)}, w, wr, r$
P_{Urst1}	$: rst$
P_{Erst1}	$: rst, r, rst$
P_{Mrst1}	$: rst, w, rst$
P_{Frst1}	$: rst, w, w^{*:(\#w=MAX-1)}, rst$

Tabla 4.4: Secuencias de prueba concretas.

encarga de procesar la información proveniente del ítem recibido y actualizar los valores de las variables que modelan el modelo de cobertura. Finalmente, el método *sample* ejecuta la linea *fifo_fsm.sample()*, la cual actualiza el *covergroup* que contiene la implementación del modelo de cobertura definido en la linea 43.

Ademas de implementar el modelo de cobertura desarrollado, se implementaron tres modelos

de cobertura funcional más básicos. El primero consiste en la medición de la cobertura de comandos, es decir, controlar si se ejercitaron todas las operaciones sobre la FIFO (linea 18). El segundo consiste en medir si el diseño atraviesa todos los estados, es decir *UNDEFINED, EMPTY, MIDDLE* y *FULL* (linea 27). El tercer modelo de cobertura funcional mide si se ejercitaron todos los arcos de la EFSM presentada en la Figura 4.3, es decir, si se ejercitaron todos los comandos en cada uno de los estados. Esto último se implementó haciendo el producto cruz entre los comandos y los estados (linea 36), ignorando las operaciones *Read, Write, Write&Read* y *Reset* en el estado *UNDEFINED*.

```
1  class fifo_coverage extends uvm_subscriber#(fifo_item);
2
3    `uvm_component_utils(fifo_coverage)
4
5    typedef enum {Undefined, Empty, Middle, Full} state_t;
6    typedef enum {Write, Read, Reset, Write_and_Read} cmd_t;
7
8    fifo_item t;
9
10   state_t state;
11   cmd_t cmd;
12
13   bit first_reset;
14
15   covergroup fifo_fsm;
16
17     // COBERTURA DE COMANDOS
18     CMD: coverpoint cmd
19     {
20       bins write = { Write };
21       bins read  = { Read };
22       bins reset = { Reset };
23       bins write_and_read = { Write_and_Read };
24     }
25
26     // COBERTURA DE ESTADOS
27     STATE: coverpoint state
28     {
29       bins undefined = { Undefined };
30       bins empty     = { Empty };
31       bins middle    = { Middle };
32       bins full      = { Full };
33     }
34
35     // COBERTURA DE ARCOS
36     ARC_COV: cross STATE, CMD
37     {
38       ignore_bins ignore = binsof (STATE.undefined) && (! binsof (CMD.reset));
39     }
40
41     // ============================================================================
42     // MODELO DE COBERTURA FUNCIONAL UTILIZADO: COBERTURA DE CAMINOS
```

```
43      STATE_CMD_PATH : coverpoint state_cmd
44      {
45        bins legal[] =
46          ( U_Rst => E_Wr => M_Rd ),                                  // PEMmin1
47          ( U_Rst => E_WR => M_Rd ),                                  // PEMmin2
48          ( U_Rst => E_Wr => M_WR => M_Rd ),                          // PEM1
49          ( U_Rst => E_Wr => M_Wr[*6] => M_WR => M_Rd[*7] ),          // PEM2
50          ( U_Rst => E_Wr => M_Wr[*2] => M_WR => M_Rd[*2]),           // PEM3
51          ( U_Rst => E_Rd => E_Wr => M_Wr[*6] => M_Rd[*7] => E_Rd ), // PEMmax1
52
53          ( U_Rst => E_Wr => M_Wr[*7] => F_Rd )                       // PMFmin1
54          ( U_Rst => E_Wr => M_Wr[*7] => F_WR => F_Rd ),              // PMF1
55          ( U_Rst => E_Wr => M_Wr[*7] => F_Wr => F_Rd ),              // PMF2
56          ( U_Rst => E_Wr => M_Wr[*7] => F_Wr => F_WR => F_Rd ),      // PMFmax1
57
58          ( U_Rst ),                                                  // PUrst1
59          ( U_Rst => E_Rd => E_Rst ),                                 // PErst1
60          ( U_Rst => E_Wr => M_Rst ),                                 // PMrst1
61          ( U_Rst => E_Wr => M_Wr[*7] => F_Rst );                     // PFrst1
62        bins illegal = default sequence;
63      }
64      // ============================================================================
65
66    endgroup;
67
68    virtual function void sample(fifo_item t);
69      this.t = t;
70      // Se establece el comando.
71      case (this.t.get_item_type())
72        "RESET"           : this.cmd = Reset;
73        "READ"            : this.cmd = Read;
74        "WRITE"           : this.cmd = Write;
75        "WRITE_AND_READ"  : this.cmd = Write_and_Read;
76      endcase
77      // Se establece el estado.
78      if ((this.t.reset == 1) && (this.first_reset == 1))
79        begin
80          this.state = Undefined;
81          first_reset = 0;
82        end
83      else
84        begin
85          if ((this.t.empty == 1) && (this.t.full == 0))
86            this.state = Empty;
87          else
88            if ((this.t.empty == 0) && (this.t.full == 0))
89              this.state = Middle;
90            else
91              if ((this.t.empty == 0) && (this.t.full == 1))
92                this.state = Full;
93        end
94      // Se establece el arco.
95      if ( (this.state == Undefined) && (this.cmd == Reset) ) this.state_cmd = U_Rst;
96      if ( (this.state == Empty) && (this.cmd == Reset) ) this.state_cmd = E_Rst;
```

```
97      if ( (this.state == Empty) && (this.cmd == Read) ) this.state_cmd = E_Rd;
98      if ( (this.state == Empty) && (this.cmd == Write) ) this.state_cmd = E_Wr;
99      if ( (this.state == Empty) && (this.cmd == Write_and_Read) ) this.state_cmd = E_WR
            ;
100     if ( (this.state == Middle) && (this.cmd == Reset) ) this.state_cmd = M_Rst;
101     if ( (this.state == Middle) && (this.cmd == Read) ) this.state_cmd = M_Rd;
102     if ( (this.state == Middle) && (this.cmd == Write) ) this.state_cmd = M_Wr;
103     if ( (this.state == Middle) && (this.cmd == Write_and_Read) ) this.state_cmd =
            M_WR;
104     if ( (this.state == Full) && (this.cmd == Reset) ) this.state_cmd = F_Rst;
105     if ( (this.state == Full) && (this.cmd == Read) ) this.state_cmd = F_Rd;
106     if ( (this.state == Full) && (this.cmd == Write) ) this.state_cmd = F_Wr;
107     if ( (this.state == Full) && (this.cmd == Write_and_Read) ) this.state_cmd = F_WR;
108
109     fifo_fsm.sample();
110   endfunction
111
112   function void write(fifo_item t);
113     sample(t);
114     if (settings.coverage_debug) `uvm_info("COVERAGE", $sformatf("Coverage=%0d%␣(t=%s
            )", fifo_fsm.get_inst_coverage(), t.convert2string()), UVM_MEDIUM)
115   endfunction
116
117   function new(string name = "fifo_coverage", uvm_component parent );
118     super.new(name, parent);
119     if( !uvm_config_db#(fifo_config)::get(this, "", "fifo_config", settings))
120       `uvm_fatal("Config␣Fatal", "Can't␣get␣the␣fifo_config");
121     first_reset = 1;
122     fifo_fsm = new();
123   endfunction
124
125 endclass
```

Código 4.1: Clase *fifo_coverage*. La misma implementa el modelo de cobertura funcional desarrollado para el *buffer* FIFO utilizando las estructuras provistas por SV.

4.4.4. Resultados de Simulación

Las secuencias definidas fueron ejecutadas mediante el *testbench* implementado, siguiendo la metodología de trabajo explicada al principio de la Sección 4.4. Esto es, se comenzó verificando la version $v0$ del DUV, y luego, verificando las versiones corregidas en función de los errores observados en las versiones anteriores ($m1$ hasta $m6$). Los errores detectados por los representantes de cada clase de equivalencia en las diferentes versiones de la FIFO se pueden observar en la Tabla 4.5.

Para correr las respectivas pruebas concretas sobre las distintas versiones del DUV se utilizó el archivo *Makefile* (ver Apéndice .1.1) haciendo referencias a las versiones de la siguiente forma:

- make all_v1

Secuencia de prueba concreta	v1	m1	m2	m3	m4	m5	m6
$P_{EMmin1} : rst, w, r$	bug						
$P_{EMmin2} : rst, wr, r$	bug	bug					
$P_{EM1} : rst, w, wr, r$	bug	bug	bug	bug	bug		
$P_{EM2} : rst, w, w^{*:(\#w=MAX-2)}, wr, r^{*:(\#r=MAX-1)}$	bug	bug	bug	bug	bug	bug	
$P_{EM3} : rst, w, w^{*:(\#w=2)}, wr, r^{*:(\#r=2)}$	bug	bug	bug	bug			
$P_{EMmax1} : rst, r, w, w^{*:(\#w=MAX-2)}, r^{*:(\#r=MAX-1)}, r$	bug	bug					
$P_{MFmin1} : rst, w, w^{*:(\#w=MAX-1)}, r$	bug		bug				
$P_{MF1} : rst, w, w^{*:(\#w=MAX-1)}, wr, r$	bug	bug	bug				
$P_{MF2} : rst, w, w^{*:(\#w=MAX-1)}, w, r$	bug	bug					
$P_{MFmax1} : rst, w, w^{*:(\#w=MAX-1)}, w, wr, r$	bug		bug				
$P_{Urst1} : rst$							
$P_{Erst1} : rst, r, rst$							
$P_{Mrst1} : rst, w, rst$							
$P_{Frst1} : rst, w, w^{*:(\#w=MAX-1)}, rst$							

Tabla 4.5: Errores detectados por las secuencias concretas de prueba en las diferentes versiones del DUV.

- make all_m1
- ...
- make all_m6

4.4.5. Análisis

La técnica planteada facilitó el análisis del DUV y la posterior generación de su modelo de cobertura funcional. La misma resulto útil para poder analizar el comportamiento del DUV mediante un mayor nivel de abstracción, enfocando la atención a cuestiones de alto nivel, evitando perderse en detalles de implementación. La arquitectura del *testbench* desarrollado previamente, facilitó el mapeo de las secuencias de pruebas abstractas en secuencias ejecutables mediante el uso de los métodos de la clase *fifo_item*, definidos para facilitar la creación de pruebas dirigidas.

Una ventaja de este enfoque de dos niveles, secuencias de pruebas abstractas y secuencias de pruebas ejecutables, es que las secuencias de pruebas abstractas son independientes del lenguaje utilizado para implementar las pruebas ejecutables y la infraestructura del *testbench*, por lo que la técnica se puede utilizar con otros lenguajes o metodologías de verificación.

El uso de los otros tres modelos de cobertura funcional (cobertura de comandos, cobertura de estados y cobertura de arcos) demostró como algunos modelos básicos de cobertura funcional pueden arrojar una cobertura del 100 %, aun cuando existen escenarios que pueden generar la aparición de un error en el diseño. Por ejemplo, la cobertura de comandos arrojó una cobertura

del 100 % solo ejecutando la secuencia P_{EM1}, y la cobertura de estados arrojó también una cobertura del 100 % solo ejecutando la secuencia P_{Frst1}. Se demuestra como muchas veces un modelo de cobertura básico, implementado generalmente mediante una lista de funcionalidades, no es suficiente para estresar adecuadamente un diseño, ya que existen secuencias de pruebas complejas que dependen de la historia pasada del dispositivo las cuales deben ser cubiertas para asegurar el correcto funcionamiento del diseño y no pueden ser capturadas por un modelo de cobertura básico.

La técnica presentada en este capítulo facilitó la exploración y extracción de escenarios que no resultaban obvios analizando directamente las especificaciones del DUV. La misma provee mayor poder expresivo para definir escenarios o casos de prueba que la técnica tradicional utilizada para definir modelos de cobertura funcional (la cual se basa en el listado de funcionalidades extraídas a partir de las especificaciones).

La técnica también resultó útil para extraer y expresar escenarios que dependen de la historia pasada del dispositivo y sirvió para dirigir, de forma clara, las pruebas hacia ciertas regiones de interés.

Implementar la medición de la cobertura de los caminos permite, adicionalmente, utilizar la técnica desarrollada en este capítulo mediante la metodología de verificación CRV, generando las secuencias de prueba de forma aleatoria, esperando que las mismas cubran en algún momento los caminos implementados en el medidor de cobertura.

Capítulo 5

Conclusiones Generales

La verificación funcional es en la actualidad un tema de relevancia en la academia y la industria. Estudios recientes estiman que la mitad de todos los chips diseñados requieren fabricarse una o más veces debido a errores, siendo el 77 % de estas fallas errores funcionales [14].

Dada la cantidad de esfuerzo que exige la verificación, la falta de ingenieros de verificación de hardware, y la cantidad de código que debe producirse, no es de extrañar que, en la mayoría de los proyectos, la verificación recaiga en la ruta crítica, siendo alrededor del 70 % del presupuesto de un proyecto invertido en dicha etapa [49]. Esta, también es la razón por la que en la actualidad la verificación es blanco de nuevas herramientas y metodologías, las cuales intentan reducir el tiempo de verificación permitiendo optimizar el esfuerzo mediante el manejo de mayores niveles de abstracción y automatización.

En función de la problemática identificada se planteó el objetivo general de comprender, aplicar y analizar, desde un punto de vista teórico-practico, dos de los conceptos fundamentales de la verificación funcional de hardware basada en simulación: la arquitectura de los *testbenches* y los modelos de cobertura funcional.

En concordancia con lo anterior, el primer objetivo puntual de este trabajo consistió en desarrollar los conocimientos y las capacidades, mediante el estudio bibliográfico y la experimentación con casos de estudios, para diseñar, implementar y usar *testbenches* estructurados y reusables, utilizando tecnología del estado del arte, y en función de esto poder analizar su arquitectura en pos de facilitar y mejorar el proceso de verificación.

Mediante el estudio bibliográfico, se identificó que una de las principales propiedades con la que debe contar un *testbench* es la de reusabilidad. El mismo se debe poder reutilizar de forma horizontal, es decir en diferentes proyectos, y de forma vertical, es decir a través de diferentes niveles del mismo proyecto, desde la verificación a nivel de bloque, pasando por la verificación a nivel de integración, hasta la verificación a nivel de sistema. Para lograr esta reusabilidad, los

testbenches deben estar implementados utilizando componentes intercambiables.

Se identificó que otra propiedad fundamental con la que que debe contar un *testbench* es la capacidad para aumentar el nivel de abstracción, separando principalmente los detalles de bajo nivel relacionados a los temporizados y sincronizados de señales de la funcionalidad que se desea probar.

Para facilitar el diseño e implementación de una infraestructura de verificación que cumpla con dichas propiedades se recurrió a la metodología UVM, con el fin de implementar un *testbench* con tecnología y metodología del estado del arte.

Se diseñó e implementó un *testbench* para realizar la verificación funcional de un *buffer* FIFO. Este componente, con un protocolo de funcionamiento claro y rico en términos de posibles secuencias de prueba que se le puede inyectar, resultó interesante en términos de generación de estímulos, implementación del *Driver* y creación del modelo de referencia, permitiendo abordar de esta forma los principales conceptos de la verificación basada en simulación.

En el desarrollo del entorno para la FIFO se identificó que la implementación de las cuestiones de bajo nivel del *testbench*, es decir, el manejo de la estimulación a nivel de señal que involucra aspectos de temporizado y sincronización, resultan complejas de implementar. En función de esto se decidió desarrollar un *tesbench* UVM para un DUV que emplee un protocolo estándar de comunicación utilizado en la industria, con el objetivo de poder desarrollar un *Driver* más complejo, y ganar experiencia en el proceso. Por lo tanto, se decidió desarrollar un *testbench* para el módulo EEPROM I2C, aprovechando la experiencia adquirida con el *testbench* desarrollado para el *buffer* FIFO. La arquitectura del *testbench* fue similar a la del caso de estudio previo, permitiendo reutilizar la mayoría de la estructura y los VC ya implementados con mínimas modificaciones.

Se comprendió que para poder lograr reusabilidad dentro de la industria y favorecer la creación de mercados de VIP es recomendable utilizar metodologías estándar de verificación, las cuales brindan un *framework* y establecen un conjunto de prácticas de cómo utilizar dicho *framework* para generar componentes reusables.

Diseñar un *testbench* reutilizable que permita facilitar la tarea de verificación aumentando el nivel de abstracción resultó una tarea compleja que involucró conocimientos de distintos campos, algunos propios de la disciplina de verificación, como conceptos de cobertura, HVL, modelos de referencia y aleatorización; otros propios de la ingeniería de software, como los conceptos relacionados a la POO, como clases, patrones de diseño, métodos de *Factory*, estructuras de datos; otros conceptos de sistemas, como creación, comunicación y sincronización de procesos; y obviamente otros conceptos propios del diseño digital de circuitos digitales, como simulación de HDL, temporizados de señales, tiempos de *hold*, *rise*, etc.

El trabajo desarrollado en función de este primer objetivo fue publicado en la 8^a *Conferencia*

de Micro-Nanoelectrónica, Tecnología y Aplicaciones (CAMTA) del año 2014 bajo el titulo *UVM based testbench architecture for unit verification* [16].

El segundo objetivo de este trabajo consistió en desarrollar los conocimientos y las capacidades para definir modelos de cobertura funcional de un diseño a partir de modelos funcionales de alto nivel del mismo, facilitando la exploración del espacio de cobertura y mejorando de esta forma el proceso de creación de pruebas.

Se identificó que la simulación de eventos discretos sigue siendo la técnica de verificación más utilizada en la actualidad, debido a que es la más adecuada para aplicar en sistemas grandes y complejos [26]. Así y todo, el espacio de posibles estímulos para diseños modernos presenta una complejidad que lo hace inviable de cubrir de forma exhaustiva dentro de un cronograma de verificación típico, en consecuencia, es fundamental identificar los patrones de estímulos clave para ejercitarlos de forma eficiente.

Un modelo de cobertura funcional de buena calidad facilita la generación de un conjunto de secuencias de prueba mínimo, capaz de ejecutar condiciones complejas y poco comunes, evitando la generación de secuencias replicadas, pero logrando un alto nivel de cobertura. Por otro lado, la definición de un modelo de cobertura funcional completo y correcto es, a menudo, difícil de obtener directamente a partir de la especificaciones. Para muchos diseños resulta difícil visualizar y extraer comportamientos y escenarios de uso directamente a partir de sus especificaciones.

En función del segundo objetivo y la problemática identificada se introdujo un método empírico de caja negra para derivar un modelo de cobertura para diseños dominados por el control. Este método se basó en la definición de un modelo funcional del DUV que facilita la extracción de conjuntos de secuencias de prueba, los cuales definen un modelo de cobertura funcional. Dada la complejidad de los espacios de prueba, las conocidas técnicas de *Testing de Software*, partición en clases de equivalencia y análisis de valores limites, fueron aplicadas para reducirlos. Cabe destacar que se desarrolló una notación formal para poder expresar las secuencias de pruebas equivalentes. Este método se aplicó para la verificación del módulo de *buffer* FIFO presentado previamente en el Capítulo 3. El objetivo fue generar un modelo de cobertura funcional adecuado para este diseño utilizando la técnica desarrollada. Se describió la aplicación de cada paso de la técnica y las principales decisiones tomadas en el proceso.

El caso del *buffer* FIFO ejemplificó como muchas veces un modelo de cobertura básico implementado mediante una lista de funcionalidades no es suficiente para estresar adecuadamente un diseño, ya que existen secuencias de pruebas complejas que dependen de la historia pasada del dispositivo, las cuales deben ser cubiertas para asegurar el correcto funcionamiento del diseño y no pueden ser capturadas por una simple lista de funcionalidades.

El aumento del nivel de abstracción, mediante el uso de un modelo funcional del diseño, facilitó la generación de casos de prueba permitiendo analizar el problema desde una óptica más

elevada. De esta manera se evito perderse en los detalles propios de la implementación del diseño durante el análisis de los posibles casos de prueba, facilitando la extracción de los mismos.

El objetivo final de éste trabajo consistió en integrar los dos objetivos anteriores. Esto se logró utilizando el *testbench* implementado para el *buffer* FIFO, para que el mismo brinde soporte práctico a las pruebas extraídas del modelo de cobertura funcional desarrollado anteriormente. El *testbench* facilitó el mapeo de estos conjuntos de pruebas abstractas en pruebas ejecutables, permitiendo medir la cobertura funcional y verificar automáticamente el comportamiento del diseño al ser estimulado con dichas secuencias.

De esta manera, se abordaron exitosamente dos de los conceptos fundamentales de la verificación funcional de hardware basada en simulación, los modelos de cobertura funcional y la arquitectura de los *testbenches*.

De este trabajo se desprende que resulta favorable aumentar el nivel de abstracción, tanto para facilitar el diseño de los modelos de cobertura funcional, como para lidiar con la complejidad de los *testbenches*.

Adicionalmente, aprovechando el conocimiento sobre verificación funcional adquirido durante el desarrollo de la tesis, se realizo un trabajo para un filtro de imágenes en el cual se diseñó e implementó su respectivo *testbench* para poder realizar su posterior verificación funcional. En dicho trabajo se describe el desarrollo de un filtro de imágenes con procesamiento paralelo, desde las especificaciones del mismo hasta su síntesis en FPGA. Para este trabajo se desarrolló un entorno de verificación en *SystemVerilog* y se utilizó un modelo de referencia de alto nivel implementado en una herramienta matemática (*Octave*). El trabajo se encuentra publicado en las memorias del *VI Congreso de Microelectrónica Aplicada* (uEA 2015) bajo el título *Diseño y Verificación de un Filtro de Imágenes* [43].

Como trabajo futuro se plantea, en primera instancia, reutilizar el Agente del *testbench* del *buffer* FIFO para la verificación de integración de este dispositivo en un sistema. Buscando de esta forma probar la reusabilidad de los componentes utilizados previamente, pero esta vez, a nivel de verificación de integración.

También se propone probar esta primera aproximación de la técnica para facilitar la extracción de modelos de cobertura funcional en otros diseños, con el objetivo de validar su aplicabilidad en un conjunto más amplio de casos.

En caso de validarse dicha técnica para un conjunto más amplio de diseños, se podría automatizar el proceso, expresando los caminos en alto nivel, o definiendo los mismos sobre el grafo de estados del modelo funcional. Luego se podrían traducir los mismos, automáticamente, a secuencias ejecutables de UVM. De esta forma el ingeniero en verificación se podría abstraer aun más de los detalles de implementación, enfocando su atención en la generación de un conjunto optimo de pruebas.

Finalmente, para remarcar la importancia de esta linea de investigación en el campo de la verificación funcional se puede citar la actual tendencia de generación de estímulos basados en grafos. Esto forma parte fundamental del concepto de *Intelligent Testbench Automation* (iTBA) propuesto por *Mentor Graphics* y utilizado en su herramienta *InFact*, en el cual se define el posible espacio de pruebas mediante grafos y luego se generan, automáticamente, secuencias de pruebas ejecutables para UVM. De esta manera se intenta alcanzar un mayor nivel de cobertura en menor tiempo, facilitando la generación de pruebas y fomentando la reusabilidad y portabilidad del modelo de cobertura y generación de estímulos hacia distintos *testbenches*. En 2014 *Mentor Graphics* donó el *Graph-based Specification Format* a *Acellera*. En 2015 *Accellera* aprobó la creación del *Portable Stimulus Working Group* con el objetivo de estandarizar la generación de estímulos basados en grafos de *Mentor Graphics*, demostrando la importancia de esta linea de investigación en el campo de la verificación funcional.

Apéndice

.1. Scripts

.1.1. FIFO Makefile

```
#
# Makefile for VCS MX for UVM FIFO Testbench
# Author: Juan Ignacio Francesconi - UNS
#

all_v1: clean comp_v1 run dve
all_m1: clean comp_m1 run dve
all_m2: clean comp_m2 run dve
all_m4: clean comp_m3 run dve
all_m5: clean comp_m4 run dve
all_m6: clean comp_m5 run dve
all_m7: clean comp_m6 run dve

debug_v1: clean comp_v1 debug_uvm
debug_m1: clean comp_m1 debug_uvm
debug_m2: clean comp_m2 debug_uvm
debug_m4: clean comp_m3 debug_uvm
debug_m5: clean comp_m4 debug_uvm
debug_m6: clean comp_m5 debug_uvm
debug_m7: clean comp_m6 debug_uvm

comp_v1: library vhdl_v1 rest
comp_m1: library vhdl_m1 rest
comp_m2: library vhdl_m2 rest
comp_m4: library vhdl_m3 rest
comp_m5: library vhdl_m4 rest
comp_m6: library vhdl_m5 rest
comp_m7: library vhdl_m6 rest

library:# Library creation
        mkdir -p WORK
        chmod -R 755 WORK

vhdl_v1:# VHDL Analysis for FIFO v1
        vhdlan -q -f vhdl_files_v1
```

```
36
37  vhdl_m1:# VHDL Analysis for FIFO mutant1
38          vhdlan -q -f vhdl_files_m1
39
40  vhdl_m2:# VHDL Analysis for FIFO mutant2
41          vhdlan -q -f vhdl_files_m2
42
43  vhdl_m3:# VHDL Analysis for FIFO mutant4
44          vhdlan -q -f vhdl_files_m3
45
46  vhdl_m4:# VHDL Analysis for FIFO mutant5
47          vhdlan -q -f vhdl_files_m4
48
49  vhdl_m5:# VHDL Analysis for FIFO mutant6
50          vhdlan -q -f vhdl_files_m5
51
52  vhdl_m6:# VHDL Analysis for FIFO mutant7
53          vhdlan -q  -xlrm -f vhdl_files_m6
54  # -xlrm: Enable vhdl features beyond those described in lrm.
55
56  rest:   # Verilog Analysis
57          vlogan -q -ntb_opts uvm -sverilog
58          vlogan -q -ntb_opts uvm -sverilog -f sv_files
59
60          # Elaboration
61          vcs -cm line+branch -ntb_opts uvm -debug_all top -l fifo_comp.log
62
63  run:
64          # Simulation ( Can add +UVM_TR_RECORD +UVM_LOG_RECORD +UVM_PHASE_RECORD )
65          ./simv -cm line+branch +UVM_NO_RELNOTES -ucli -i fifo.do -l fifo_run.log
66
67  dve:
68          # Coverage analysis with dve
69          dve -cov -dir simv.vdb &
70
71          # Waveform viewer
72          dve -vpd fifo_dump.vpd -script fifo_wave.tcl &
73
74          # Unified Report Generator
75          # urg -dir simv.vdb -report text
76
77  debug_uvm:
78          ./simv -gui +UVM_TR_RECORD +UVM_NO_RELNOTES -l fifo_run.log
79
80  clean:
81          \rm -rf simv* *.vpd *.dump csrc *.sim *.mra *.log ucli.key session* *.db vcs.
                key urgReport *.h log *.txt scsim* WORK/*  text
```

Código 1: Archivo Makefile para FIFO testbench

.1.2. I2C Makefile

```
# ====================================================
# File name: Makefile
# Description: Makefile for VCS MX for UVM I2C Testbench
# Author: Juan Francesconi - UNS
# Date: 2013
# ====================================================

all: clean comp run dve

debug: clean comp debug_uvm

comp: library vhdl rest

library:# Library creation
        mkdir -p WORK
        chmod -R 755 WORK

vhdl:   # VHDL Analysis for i2c_eemprom
        vhdlan -q -f vhdl_files

rest:   # Verilog Analysis
        vlogan -q -ntb_opts uvm -sverilog
        vlogan -q -ntb_opts uvm -sverilog -f sv_files

        # Elaboration
        vcs -cm line+branch -ntb_opts uvm -debug_all i2c_top -l i2c_elaboration.log

run:
        # Simulation ( Can add +UVM_TR_RECORD +UVM_LOG_RECORD +UVM_PHASE_RECORD )
        ./simv -cm line+branch +UVM_NO_RELNOTES -ucli -i i2c.do -l i2c_run.log

dve:
        # Coverage analysis with dve
        dve -cov -dir simv.vdb &

        # Waveform viewer
        dve -vpd fifo_dump.vpd -script i2c_wave.tcl &

        # Unified Report Generator
        # urg -dir simv.vdb -report text

debug_uvm:
        ./simv -gui +UVM_TR_RECORD +UVM_NO_RELNOTES -i i2c_debug.do -l i2c_run.log

clean:
        \rm -rf simv* *.vpd *.dump csrc *.sim *.mra *.log ucli.key session* *.db vcs.
            key urgReport *.h log *.txt scsim* WORK/*  text
```

Código 2: Archivo Makefile para I2C testbench

Bibliografía

[1] Accellera: *UVM 1.1 Class Reference*. 2011. http://accellera.org/images/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf.

[2] Accellera: *UVM 1.1 User Guide*. 2011. http://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.1.pdf.

[3] Aharon, A., B. Dorfman, E. Gofman, M. Leibowitz, V. Schwartzburd y A. Bar-David: *Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator*. IBM Syst. J., 30(4):527–538, Oct. 1991, ISSN 0018-8670. http://dx.doi.org/10.1147/sj.304.0527.

[4] Bailey, B., G. Martin, M. Grant y T. Anderson: *Taxonomies for the Development and Verification of Digital Systems*. Springer, 2005, ISBN 9780387240190. http://books.google.com.ar/books?id=i04n_I4EOMAC.

[5] Beizer, B.: *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990, ISBN 0-442-20672-0.

[6] Benjamin, M., D. Geist, A. Hartman, Y. Wolfsthal, G. Mas y R. Smeets: *A Study in Coverage-driven Test Generation*. En *Proceedings of the 36th Design Automation Conference*, págs. 970–975, 1999.

[7] Bergeron, J., E. Cerny, A. Hunter y A. Nightingale: *Verification Methodology Manual for SystemVerilog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005, ISBN 0387255389.

[8] Bromley, J. y K. Johnston: *Taming Testbench Timing - Time's Up for Clocking Block Confusion*. En *Proceedings of the 2012 Austin SNUG*. Synopsys, 2012. http://www.synopsys.com/news/pubs/snug/2012/austin/fb3_paper_bromley.pdf.

[9] Cai, L. y D. Gajski: *Transaction Level Modeling: An Overview.* En *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, págs. 19–24, New York, NY, USA, 2003. ACM, ISBN 1-58113-742-7. http://doi.acm.org/10.1145/944645.944651.

[10] Cheng, K. T. y A. S. Krishnakumar: *Automatic Functional Test Generation Using The Extended Finite State Machine Model.* En *30th Conference on Design Automation*, págs. 86–91, June 1993.

[11] Cohen, B.: *SVA in a UVM Class-based Environment.* Verification Horizons, 7, 2013.

[12] Cooper, V. R. y P. Marriott: *Demystifying the UVM Configuration Database.* En *Proceedings of the 2014 Design & Verification Conference & Exhibition (DVCon)*, 2014.

[13] Cummings, C.: *The OVM/UVM Factory & Factory Overrides - How They Work - Why They Are Important.* En *Proceedings of the 2012 Silicon Valley SNUG.* Synopsys, 2012. http://www.synopsys.com/news/pubs/snug/siliconvalley2012/ma3_cummings_paper.pdf.

[14] Foster, H.: *Evolving Verification Capabilities.* En *Verification Academy.* Mentor Graphics, 2014. https://verificationacademy.com/courses/evolving-verification-capabilities.

[15] Foster, H., A. Krolnik y D. Lacey: *Assertion-based Design.* Kluwer Academic, 2003, ISBN 9781402074981. http://books.google.com.ar/books?id=zqBvPwAACAAJ.

[16] Francesconi, J., J. Agustin Rodriguez y P. Julian: *UVM Based Testbench Architecture for Unit Verification.* En *Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, págs. 89–94, July 2014.

[17] Fujita, M., I. Ghosh y M. Prasad: *Verification Techniques for System-Level Design.* Systems on Silicon. Elsevier Science, 2010, ISBN 9780080553139. http://books.google.com.ar/books?id=MmLEIDcQUh8C.

[18] Fummi, F., U. Rovati y D. Sciuto: *Functional Design for Testability of Control-dominated Architectures.* ACM Trans. Des. Autom. Electron. Syst., 2(2):98–122, Abr. 1997, ISSN 1084-4309.

[19] Gamma, E., R. Helm, R. Johnson y J. Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[20] Grosse, D., R. Wille, R. Siegmund y R. Drechsler: *Contradiction Analysis for Constraint-based Random Simulation.* En *Forum on Specification, Verification and Design Languages*, págs. 130–135, Sept 2008.

[21] Hemmati, H., L. Briand, A. Arcuri y S. Ali: *An Enhanced Test Case Selection Approach for Model-based Testing: An Industrial Case Study.* En *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, págs. 267–276, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-791-2. http://doi.acm.org/10.1145/1882291.1882331.

[22] Ho, R., C. Yang, M. Horowitz y D. Dill: *Architecture Validation for Processors.* En *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, págs. 404–413, June 1995.

[23] Kang, I. y I. Lee: *An Efficient State Space Generation for Analysis of Real-time Systems.* SIGSOFT Softw. Eng. Notes, 21(3):4–13, May. 1996, ISSN 0163-5948. http://doi.acm.org/10.1145/226295.226297.

[24] Katrowitz, M. y L. M. Noack: *I'M Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DEC Chip 21164 Alpha Microprocessor.* En *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, págs. 325–330, New York, NY, USA, 1996. ACM, ISBN 0-89791-779-0. http://doi.acm.org/10.1145/240518.240580.

[25] Lakey, P. B.: *Model-based Specification and Testing Applied to the Ground-Based Midcourse Defense (GMD) System: An Industry Report.* SIGSOFT Softw. Eng. Notes, 30(4):1–7, May. 2005, ISSN 0163-5948. http://doi.acm.org/10.1145/1082983.1083291.

[26] Lam, W. K.: *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005, ISBN 0131433474.

[27] Lichtenstein, Y., Y. Malka y A. Aharon: *Model Based Test Generation for Processor Verification.* En *Proceedings of the sixth annual conference on innovative applications of artificial intelligence*, págs. 83–94, 1994.

[28] Mecklenburg, R.: *Managing Projects with GNU Make (Nutshell Handbooks).* O'Reilly Media, Inc., 2004, ISBN 0596006101.

[29] Meyer, A.: *Principles of Functional Verification.* Elsevier Science, 2003, ISBN 9780080469942. http://books.google.com.ar/books?id=qaIiX3hYWL4C.

[30] Mintz, M. y R. Ekendahl: *Hardware Verification with System Verilog: An Object-Oriented Framework*. International Federation for Information Processing. Springer, 2007, ISBN 9780387717401. http://books.google.com.ar/books?id=Mawfv_drBJoC.

[31] Moundanos, D., J. Abraham y Y. Hoskote: *Abstraction Techniques for Validation Coverage Analysis and Test Generation*. IEEE Transactions on Computers, 47(1):2–14, Jan 1998, ISSN 0018-9340.

[32] Murphy, C., G. Kaiser y M. Arias: *Parameterizing Random Test Data According to Equivalence Classes*. En *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, págs. 38–41, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-881-7.

[33] Myers, G. J. y C. Sandler: *The Art of Software Testing*. John Wiley & Sons, 2004, ISBN 0471469122.

[34] Noy, A.: *System and Method for Applying Flexible Constraints*, 2003. http://www.google.com/patents/US6519727, US Patent 6,519,727.

[35] Piziali, A.: *Functional Verification Coverage Measurement and Analysis*. Springer Publishing Company, Incorporated, 1st ed., 2007, ISBN 0387739920, 9780387739922.

[36] Reid, S.: *An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing*. En *Proceedings of the Fourth International Software Metrics Symposium*, págs. 64–73, Nov 1997.

[37] A. Rose, S. Swan, J. Pierce, and J.M Fernandez: *OSCI TLM Standard Whitepaper: Transaction Level Modeling in SystemC*. Accellera Systems Initiative Inc. http://www.accellera.org/downloads/standards/systemc/accept_license/accepted_download/TLM-1.0.tar.gz.

[38] Cadence Design Systems y Mentor Graphics: *OVM User Guide*. 2011. http://ovmworld.s3.amazonaws.com/downloads/ovm-2.1.2.tar.gz.

[39] NXP Semiconductors: *I2C-bus specification and user manual*. 2012.

[40] van der Schoot, H., S. Anoop, G. Ankit y S. Krishnamurthy: *A Methodology for Hardware-Assisted Acceleration of OVM and UVM Testbenches*. Verification Horizons, 7, 2011.

[41] Sarma, M., P. V. R. Murthy, S. Jell y A. Ulrich: *Model-based Testing in Industry: A Case Study with Two MBT Tools*. En *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, págs. 87–90, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-970-1. http://doi.acm.org/10.1145/1808266.1808279.

[42] Shen, J. y J. Abraham: *An RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation*. Journal of Electronic Testing, 16(1-2):67–81, 2000, ISSN 0923-8174.

[43] Soto, M. F., J. Francesconi, G. Pachiana y M. D. Federico: *Diseño y Verificación de un Filtro de Imágenes*. En *Proc. The VI Aplicated Micro-electronics Congress (uEA)*, San Justo, Buenos Aires, Argentina, May 2015.

[44] Spear, C.: *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd ed., 2008, ISBN 0387765298, 9780387765297.

[45] STMicroelectronics: *M24C01-R 1 Kbit serial I2C bus EEPROM Datasheet*. 2013. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00071904.pdf.

[46] Tasiran, S. y K. Keutzer: *Coverage Metrics for Functional Validation of Hardware Designs*. IEEE Des. Test, 18(4):36–45, Jul. 2001, ISSN 0740-7475. http://dx.doi.org/10.1109/54.936247.

[47] Utting, M. y B. Legeard: *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007, ISBN 0123725011, 9780080466484.

[48] Vasconcelos Oliveira, K. de, A. Perkusich, K. Costa Gorgonio, L. Dias da Silva y A. Falcao Martins: *Using Equivalence Classes for Testing Programs for Safety Instrumented Systems*. En *IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, págs. 1–7, Sept 2013.

[49] Vasudevan, S.: *Effective Functional Verification: Principles and Processes*. Springer, 2006, ISBN 9780387326207. http://books.google.com.ar/books?id=cEInngEACAAJ.

[50] Wang, J., J. Shao, Y. Li y J. Ding: *Survey on Formal Verification Methods for Digital IC*. En *Fourth International Conference on Internet Computing for Science and Engineering (ICICSE)*, págs. 164–168, Dec 2009.

[51] Wile, B., J. Goss y W. Roesner: *Comprehensive Functional Verification: The Complete Industry Cycle*. Systems on Silicon. Elsevier Science, 2005, ISBN 9780080476643. `http://books.google.com.ar/books?id=btl\_OX3kJ7MC`.

[52] Yossi Lichtenstein an, Y. M. y A. Aharon: *Coverage-oriented verification of Banias*. En *Proceedings of the sixth annual conference on innovative applications of artificial intelligence*, págs. 280–285, June 2003.