Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
Maestría en Diseño Electrónico

Metodología para acelerar la verificación pre-silicio con enfoque a firmware(FW)

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **MARTIN ROBERTO LINARES ALTAMIRANO**

Director **VICTOR AVENDANO FERNANDEZ**

Tlaquepaque, Jalisco. 17 de junio de 2025.

A mis papás, por confiar y creer en mí. A mi novia, por apoyar y comprenderme en tiempos tan cambiantes e inciertos. A mi terapeuta, quien me ayudo a saber quien soy y a donde voy. A mis amigos, por haber hecho todo este proceso tan ameno. A mi asesor y mi gerente, por creer en mi idea.
A todos, gracias.

Resumen

Las actuales tendencias en desarrollo de System on Chip(SoC) y sistemas multi-dado, han traído un nivel de complejidad nunca visto, generando grandes desafíos para el diseño y la verificación en etapas de pre-silicio. Dichos desafíos han sido incorporar en las pruebas del ambiente de verificación pre-silicio, flujos de software(SW) y/o firmware(FW) que normalmente corren en etapas de emulación y/o prototipado. Esto complica el desarrollo y ejecución de los ambientes de verificación, aumentando los tiempos de simulación y aumentando las dependencias y el manejo de estas. Parte de la problemática se deriva a que todo el diseño esta siendo simulado, desde el sistema de cómputo, los diferentes buses, memorias, periféricos e incluso las interfaces y puertos de debug, sin contar las adecuaciones que se tengan que hacer para cargar el programa en la simulación de pre-silicio.

Una forma de reducir este problema es hacer uso de los modelos funcionales de los diseños, escritos en SystemC que permiten replicar el funcionamiento a alto nivel de los SoCs, desde la validación de una arquitectura, hasta ejecutar SW y sistemas operativos.

Implementar una interfaz entre el modelo funcional en SystemC y el diseño en el ambiente de verificación permite la ejecución de pruebas con contenido de SW estimulando directamente el RTL al que está enfocada la misma. Reduciendo complejidades del ambiente, dependencias y facilitando opciones de debug.

Contenido

Introducción

Los actuales *SoCs* y los sistemas multi-dado han presentado grandes desafíos para la verificación pre-silicio, ya sea por la complejidad de integrar diferentes *IPs*, subsistemas y su correcta configuración o por los acelerados tiempos de desarrollo que se están necesitando. Esto último ha requerido de comenzar a correr *software* y *firmware* en las etapas de pre-silicio, practica que normalmente se realiza en etapas de emulación o post-silicio.

Si bien es posible ejecutar software en simulaciones y ambientes de pre-silicio, ya que se tiene el *RTL* del *CPU*, subsistema de cómputo, etc. La manera de hacerlo cambia drásticamente a diferencia de emularlo, prototiparlo o incluso tenerlo en post-silicio. Uno de los primeros puntos a abordar cuando se tienen pruebas basadas en *SW*, es la correcta integración del binario del programa en el sistema de memorias del diseño, ya que tiene que ser compatible con el formato en el cual el modelo de memoria en *RTL* funciona y a nivel sistema se tienen que hacer las adecuaciones para que el contenido del programa este en el nivel de memoria correcto, ya sea *boot-code*, sistema operativo, inicializaciones, programa de usuario, etc.

Esto expone otra limitación de este tipo de pruebas en ambientes de pre-silicio, las interfaces de *debug* del *SoC*, pese a que estén verificadas y completamente funcionales, debido a que se encuentran simuladas en *RTL*, seria complicado hacer uso de ellas para depurar el programa que está corriendo. Por lo que la opción inmediata sería el aprovechar la infraestructura de mensajes que se tiene en el ambiente de verificación y las formas de onda que se generan con la simulación. Si bien este método proporciona granularidad y un nivel de detalle extraordinario sobre el funcionamiento lógico del diseño, el tiempo que se emplea en identificar un problema ya sea de *hardware* o *software* puede ser elevado.

Otra alternativa a poder ejecutar flujos de *SW* y comenzar el desarrollo de este en etapas tempranas, es el uso de modelos funcionales a alto nivel, normalmente escritos en *SystemC* o proveídos por diferentes tipos de herramientas de virtualización. Estos modelos son capaces de simular el *SoC* desde un enfoque funcional, pueden simular desde el comportamiento de un bus hasta cargar sistemas operativos completos y correr diferentes programas de *software*.

Estos modelos funcionales proporcionan diferentes facilidades para el desarrollo temprano de *software*, desde no necesitar ninguna adaptación extra a los binarios que la compilación del programa genera hasta la virtualización de las diferentes interfaces de *debug* para la conexión con la variedad de herramientas enfocadas al mismo.

No obstante, este tipo de modelos no proporciona ningún tipo de verificación sobre el funcionamiento lógico del diseño. La simulación de estos modelos es puramente funcional y aporta valor a validar a alto nivel la arquitectura deseada y/o en desarrollo. Por lo que el *software* que se desarrolla usando estos modelos se podría considerar incompleto.

Implementar una interfaz que pueda conectar los dos tipos de simulación y aprovechar las ventajas de ambos genera beneficios para el desarrollo del *SoCs* en las etapas de pre-silicio. Dicha interfaz permitiría correr *SW* desde la parte del modelo en *SystemC* tomar las transacciones que están destinadas a los subsistemas y/o periféricos al que el programa está enfocado y dirigirlas al ambiente de verificación donde se encuentra el *RTL* del bloque en cuestión.

Con esto, ambas simulaciones estarían corriendo en paralelo y comunicándose mediante *Transaction-Level Model*(*TLM*)[IEEE-23], que permite la descripción y modelado de componentes y sistemas con un alto nivel de abstracción. Para facilitar la integración e implementación de *TLM*, se hace uso de la librería de *UVM Connect* [Verification Academy-15], la cual provee métodos y tipos de datos para satisfacer la conectividad entre los modelos y componentes de *SystemVerilog* y *SystemC*. Permitiendo simplificar el proceso de conexión, tipos de paquetes e incluso, tener un control sobre las diferentes fases de simulación del ambiente de *UVM* [Accelera-15].

Es importante mencionar que esta la metodología propuesta para la implementación de esta interfaz asume que el ambiente de verificación esta implementado utilizando *UVM*, *Universal Verification Methodology*. Parte de la razón de esto, es la facilidad que se tiene con los ambientes basados en *UVM* de poder escalar y expandir la funcionalidad de este.

En el primer capítulo de este documento se detalla y explica la implementación entre el modelo de *SystemC* y el ambiente de verificación con *UVM*. Los componentes adicionales que estos implementan, su funcionalidad individual y la manera en la que es integrado en los modelos y ambientes de verificación.

El segundo capítulo mencionará la manera en la que dichos componentes interactúan, la simulación es creada, se ejecuta y como ocurre el flujo de información entre el modelo de *SystemC* y el ambiente de verificación.

Es importante mencionar que este documento se centrara en el caso donde el modelo de *SystemC* corre el programa y las transacciones hacia el subsistema y/o periférico al que está enfocado el programa son ejecutadas y aplicadas en el ambiente de verificación, aun cuando en ciertas partes se especifiquen funcionalidades donde la transacción se origina en el ambiente de verificación y se aplica en el modelo de *SystemC*.

1. Componentes de la metodología

La principal propuesta de esta metodología es la habilidad de poder ejecutar flujos y/o programas en componentes para los cuales se enfocan los mismos, haciendo uso tanto de los modelos funcionales escritos en *SystemC*, como los ambientes de verificación que proveen de infraestructura altamente capaz de estimular y monitorear el correcto funcionamiento de los diseños, unidades, sistemas, etc.

Desde la perspectiva de un ambiente de verificación, uno de los principales objetivos de verificar un *SoC* es el correcto funcionamiento del mapa de memoria, sus niveles, diferentes secciones y tipos de acceso. Esto asegura que los diferentes buses e interfaces de los periféricos y subsistemas responden correctamente a los diferentes tipos de transacción que estos soporten.

Este mismo concepto de mapa de memoria es también usado en *software*, ayudando a definir las direcciones de memoria de los periféricos, las diferentes regiones del código y datos, permitiendo el desarrollo de un software más legible y escalable.

Es gracias a este mapa de memoria entonces que aquello que es definido por el flujo del *software*, puede ser aplicado a la lógica del hardware. De esta misma manera, esta metodología explota tanto el concepto como la aplicación del mapa de memoria de ambos lados de la solución. Ya que tanto para el modelo en *SystemC* y en el *RTL*, el mapa de memoria representa los rangos de direcciones a los cuales cada tipo de transacción es aplicada desde el subsistema de cómputo o *CPU*.

De esta manera definimos dos componentes que interactúan con el mapa de memoria en cada uno de los tipos de simulación. Estos componentes son los encargados de tomar las transacciones de los respectivos sistemas de bus en las respectivas simulaciones y mandarla a la simulación donde se desea que sea procesada.

Por último, es necesario hacer la implementación de una interfaz de co-simulacion entre el ambiente de verificación y la simulación del modelo de *SystemC*. Esto para que puedan estar sincronizadas y compartir propiamente los diferentes recursos y señales, relojes, *resets* e interrupciones, que ambas lleguen a necesitar.

Para este trabajo se hace uso de un ambiente de verificacion basado en *UVM* y el modelo del *SoC* es implementado utilizando *Virtualizer Studio*[Synopsys-25], el cual provee la construcción del modelo mediante bloques pre-definidos que aceleran la creación de los modelos para *SoCs*, ya sean basados en *ARM*, *RISC-V* o *ARC*.

1.1. Componentes del ambiente de verificación

Un ambiente de verificación basado en *UVM* hace uso de por lo menos, un *driver*, un agente, secuencia y sus respectivos *sequence items*. Estos componentes representan el mínimo necesario para poder estimular el diseño bajo pruebas, donde en las secuencias es el punto de origen de estos gracias a la manipulación de los diferentes tipos de *sequence items* que se tengan.

Para el caso donde se desea la conexión entre el modelo en *SystemC* y el ambiente de verificación, sabemos que todo tipo de accesos a los periféricos se generan desde el *core* del *SoC* y son propagadas a través de los buses que conforman el mismo. Es este tráfico de transacciones en los buses que debemos reemplazar en el ambiente de verificación de las secuencias y *sequence items* a el componente que recibe las transacciones del bus correspondiente del modelo del *SoC* en *SystemC*.

Este componente hace uso de los *sockets* en *TLM*, implementando un socket de tipo *target* para recibir las transacciones provenientes del modelo en *SystemC*, transformarlas al formato del *sequence item* el cual es usado para que el *driver* aplique los estímulos al *DUT* y finalmente reorganizar la transacción y regresarla al modelo del *SystemC*.

Para lograr esto, es necesario modificar el agente del ambiente de verificación y cambiar la configuración del *driver*. Así mismo es necesario implementar métodos de conversión del paquete transacción de *TLM* al *sequence item* que consume el *driver*.

1.1.1 TLM Target

TLM Target es el componente el cual recibe del modelo en *SystemC* todas las transacciones que son dirigidas al ambiente de verificación. Este componente tiene como principal objetivo el de recibir la transacción del lado de *SystemC*, convertirla a un objeto de tipo *uvm_sequence_item*

que requiera el *driver* al cual está destinada la transacción, enviar dicho objeto al *driver*, recibir su respuesta para finalmente completar la transacción original que fue recibida.

La implementación clave del componente radica en la *task* de *SystemVerilog b_transport*. La cual es llamada por el socket de tipo Initiator para poder ejectuar la transacción deseada. Esta *task* tiene como argumentos *uvm_tlm_gp* y *uvm_tlm_time*[Verification Academy-15], donde el primero es el que contiene todos los datos de la transacción, como tipo de acceso, dirección, datos, estado de la transacción entre otros. Este objeto es usado para crear un nuevo objeto de tipo *sequence item* que el *driver* usa para aplicar los estímulos, para hacer esto se hace uso de otro componente llamado *TLM Translator*.

1.1.2 TLM Translator

El *TLM Translator* es el componente usado para poder tomar un objeto *uvm_tlm_gp* y crear un objeto de tipo *uvm_sequence_item* a ser consumido por el *driver*, de igual manera el componente puede tomar un *uvm_sequence_item* y convertirlo a un *uvm_tlm_gp*. El componente también ofrece métodos para reportar los campos y atributos del objeto de una manera más legible para el usuario.

1.2. Componentes del modelo de SystemC

El modelo de *SystemC* integra dos componentes que son clave para poder hacer la conexión con el ambiente de verificación y poder hacer el envío de datos y señales de un ambiente a otro. Estos dos componentes son el *uvm_interceptor* y la *co_simulation_interface*. El primero se encarga de conectarse a los buses del *SoC* e integrarse al mapa de memoria para poder recibir todas las transacciones que correspondan a dicha sección del mapa de memoria. En conjunto con . La *co_simulation_interface* es la encargada de definir las conexiones de señales como relojes, *resets*, interrupciones, etc. Entre el modelo de *SystemC* y el *testbench* de *UVM*, adicionalmente este bloque permite definir a *Virtualizer Studio*, la herramienta usada en esta metodología, la elaboración y compilación del ambiente de *UVM*, así como los argumentos que requiere durante la ejecución de este.

1.2.1 Uvm_interceptor

Este componente de *SystemC* es el que sirve como puente entre el modelo del *SoC* ejecutándose en *Virtualizer Studio* y el ambiente de verificación pre-silicio con el *RTL* a verificar. Esto se logra implementando un modelo de memoria simple, con un tamaño, rango de direcciones y ancho de direcciones y datos configurables. Complementando a este modelo de memoria, se expone una interfaz de memoria que nos permite conectarnos al mapa de memoria del *SoC* y los diferentes buses que lo componen. Es aquí donde se implementa la otra parte de los *sockets* de *TLM*, en este caso el socket de tipo *Initiator*, el cual redirige todas las transacciones de memoria que llegan al *uvm_interceptor* al ambiente de verificación de *UVM*.

1.2.2 Co-Simulation Interface

Para lograr que la simulación en *Virtualizer Studio* y la del ambiente de verificación se ejecuten en paralelo y compartiendo recursos, se implementa este bloque provisto ya por *Virtualizer Studio*. En él se especifican las señales y buses que van a compartir ambas simulaciones, aquí es donde se especifican los relojes que se conectaran desde del modelo de *SystemC* hasta el *testbench* del ambiente, lo cual permite que ambos tengan el mismo reloj. Adicionalmente este módulo provee los mecanismos necesarios para manejar ambas simulaciones, permitiendo definir la lista de archivos del ambiente de verificación, las opciones de elaboración y compilación, así como las opciones de ejecución de simulación.

1.2.3 Uvmc_connectors

Los *uvmc_connectors* son los componentes que permiten hacer la conexión con la contraparte del *TLM Socket* al que se requiere conectar del lado del ambiente de verificación. Estos solo requieren tener definido un identificador, el tipo de conector que estará en el lado de *UVM* y el ancho de los datos que se van a manejar en la transacción. Adicionalmente, exponen una interfaz de memoria que sirve como intermediario entre la conexión del *uvm_interceptor* y el *TLM_Target* de *UVM*. Para este ambiente se implementa un solo *uvmc_connector*.

2. Integración de componentes y flujo de la simulación

Por si solos estos componentes no aportan a la solución que esta metodología busca, es cuando están integrados en sus respectivos ambientes y la simulación es ejecutada que podemos ver los beneficios de su integración. Previamente se habían mencionado el concepto de sockets y que era el principal mecanismo por el cual se logra la comunicación y manejo de las transacciones entre las dos simulaciones.

Sin embargo, este proceso requiere de la configuración respectiva en cada punto, así como de hacer las inicializaciones y conexiones correctas del lado del ambiente de *UVM*. Asimismo, es necesario hacer adaptaciones o modificaciones a dos componentes clave del ambiente, principalmente el agente y el *driver*.

Primero se explicará el proceso de integración en el ambiente de verificación de *UVM* y las definiciones extras que se necesitan por parte de los *sockets* de *TLM*, para después desarrollar la conexión que existe entre el ambiente y el modelo de *SystemC* utilizando *Virtualizer Studio*.

Finalmente, es necesario entrar en detalle de la secuencia y tiempo de vida de una transacción que se origina desde el modelo y termina en el ambiente de verificación.

2.1. Integrando con el ambiente de verificación

Para el ambiente de verificacion es importante establecer la comunicación entre el *TLM Target* y el *driver* de la interfaz que deseamos ejercitar con las transacciones generadas desde el modelo de *SystemC*. Ya que no necesitamos las secuencias o el *sequencer*, podemos prescindir de crear y conectar dichos objetos en el ambiente.

Tomando como ventaja que en el agente es donde el *sequencer* y el *driver* se conecta, es en este mismo componente de *UVM* que se hace la creación y conexión del *TLM Target* con el *driver*. Como se explica en el capítulo anterior, el *TLM Target* hace uso de *TLM FIFOs* para hacer la transferencia del *sequence_item* generado a través del *uvm_tlm_gp*.

2. INTEGRACIÓN DE COMPONENTES Y FLUJO DE LA SIMULACIÓN

Ya que el *sequencer* no se estará utilizando para que el *driver* este recibiendo los diferentes *uvm_sequence_items* que aplica a la interfaz, es necesario modificar el *driver* existente agregando dos puertos para la conexión con la *TLM FIFO*, uno para recibir la transacción del *TLM Target* y otro para enviar la transacción procesada de vuelta al *TLM Target*. Estos puertos, reemplazan las llamadas que típicamente haría el *driver* al *sequencer* para obtener el siguiente *sequence_item* disponible para consumir.

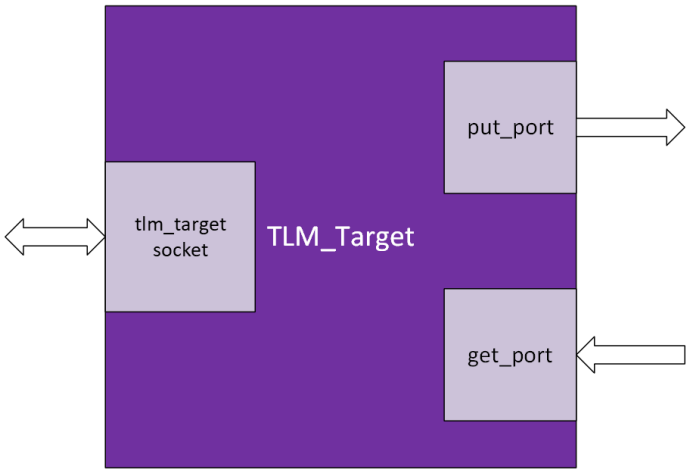

Fig 2-1 Diagrama a bloques del *TLM Target* mostrando los puntos importantes de conexión tanto para el modelo en *SystemC* y los puertos para el ambiente de verificacion

El agente entonces, instancia al *TLM Target* y al *driver* del ambiente y los conecta usando las dos *TLM FIFO* para la dirección de transacción *TLM Target* a *Driver* y *Driver* a *TLM Target*. Ya que se espera que se esté procesado una transacción al mismo tiempo, el tamaño de las *TLM FIFO* se maneja como 1.

Una vez definido la creación y conexión de los componentes del agente, es necesario hacer la conexión del socket de *TLM* que existe en *TLM Target*. Debido al orden de creación de componentes y el orden de las fases de *UVM*, es necesario mandar a llamar el método de *connect* de la librería de *UVM Connect*, en el cual se especifica el tipo de dato que se manejara entre los *sockets*, *uvm_tlm_gp*, y la jerarquía en el ambiente de *UVM* donde es instanciado el *socket*, así como el identificador que el *uvmc_connector* estará buscando del lado de *SystemC*.

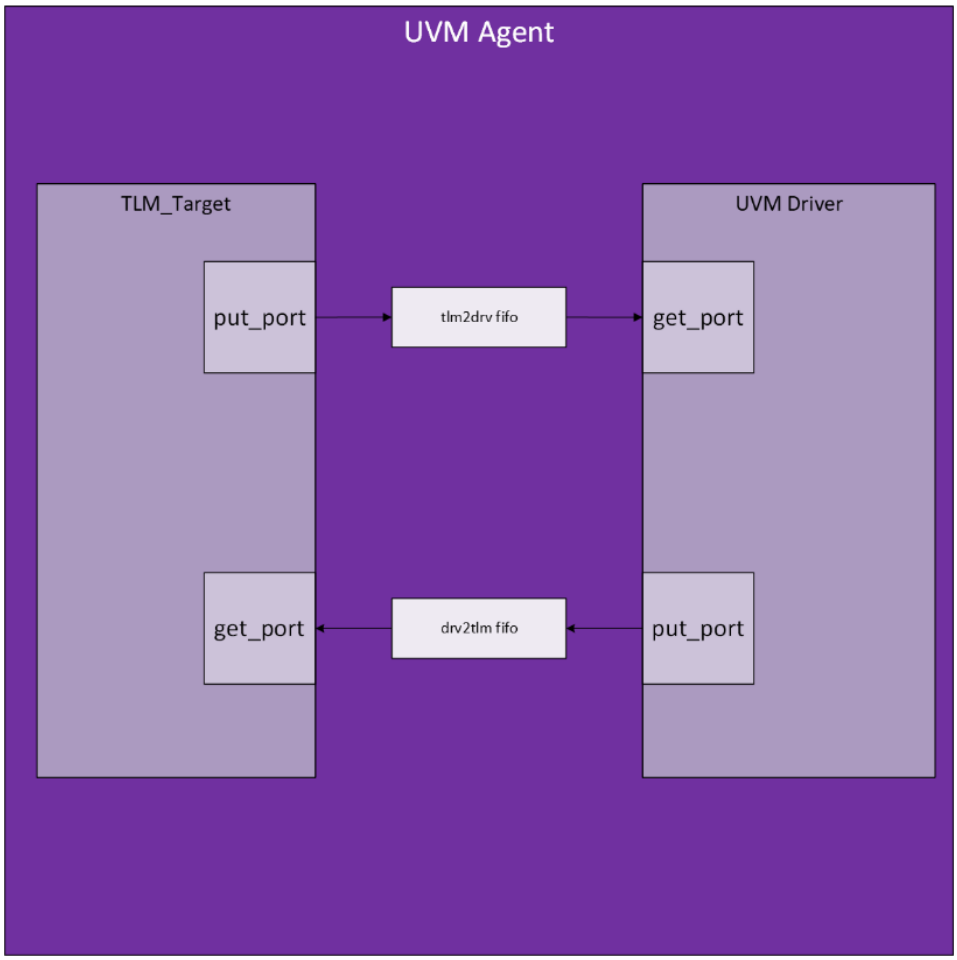

Fig 2-2 Diagrama a bloques del Agente del ambiente de verificacion. Muestra con detalle la conexión que se da entre el *TLM_Target* y el *Driver* a través de las *FIFOs*

Finalmente, solo queda definir el *uvm_test* que será ejecutado por la interfaz de co-simulacion. Debido a que la gran mayor parte del tiempo de ejecución el modelo de *SystemC* es quien está llevando el control de la simulación, solo se define una prueba de *UVM* sin contenido, solo es necesario hacer la llamada respectiva del *rise_objection* para poder iniciar la prueba de *UVM*.

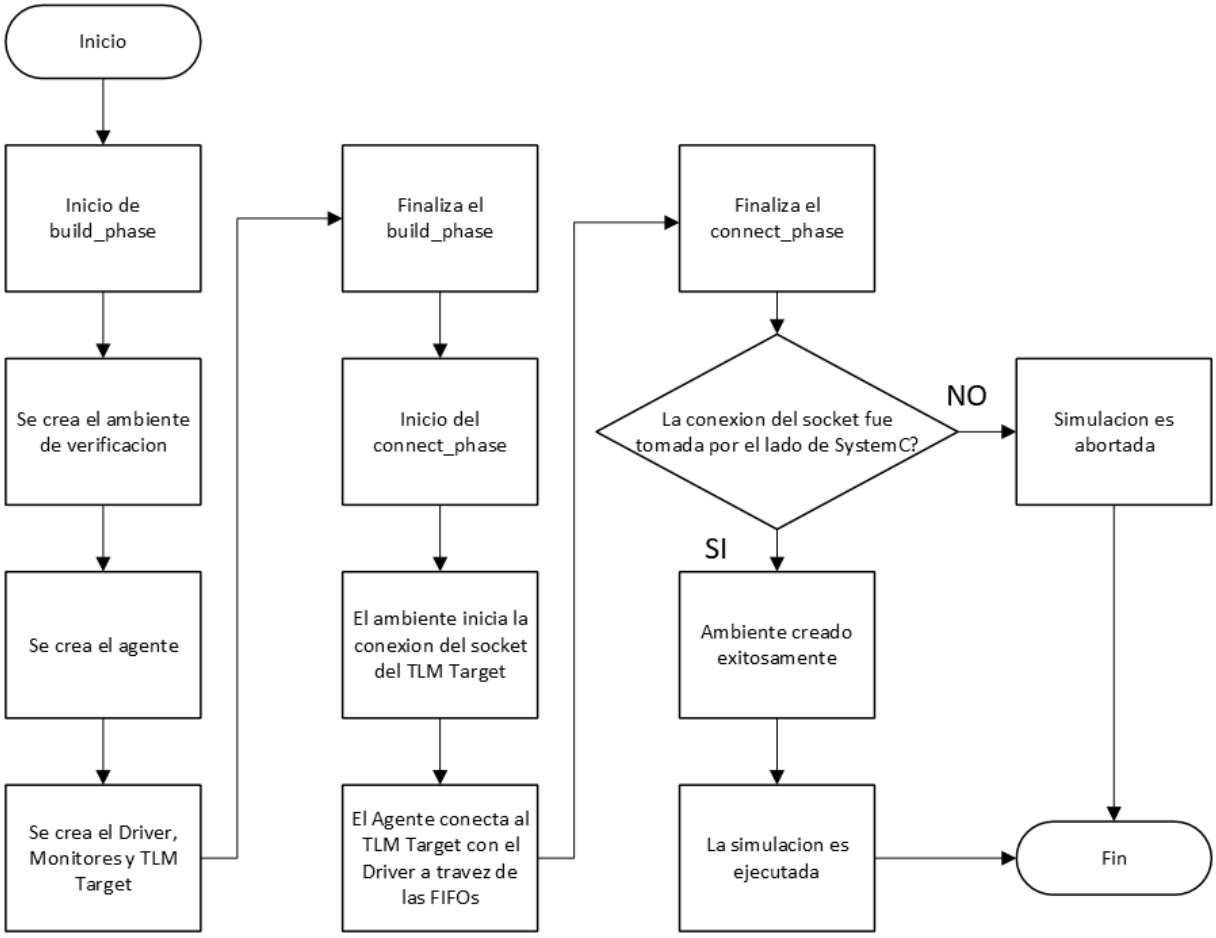

Fig 2-3 Diagrama de flujo detallando el proceso de creación y conexión del *TLM Target* en el ambiente de verificacion dentro de las diferentes fases de *UVM*

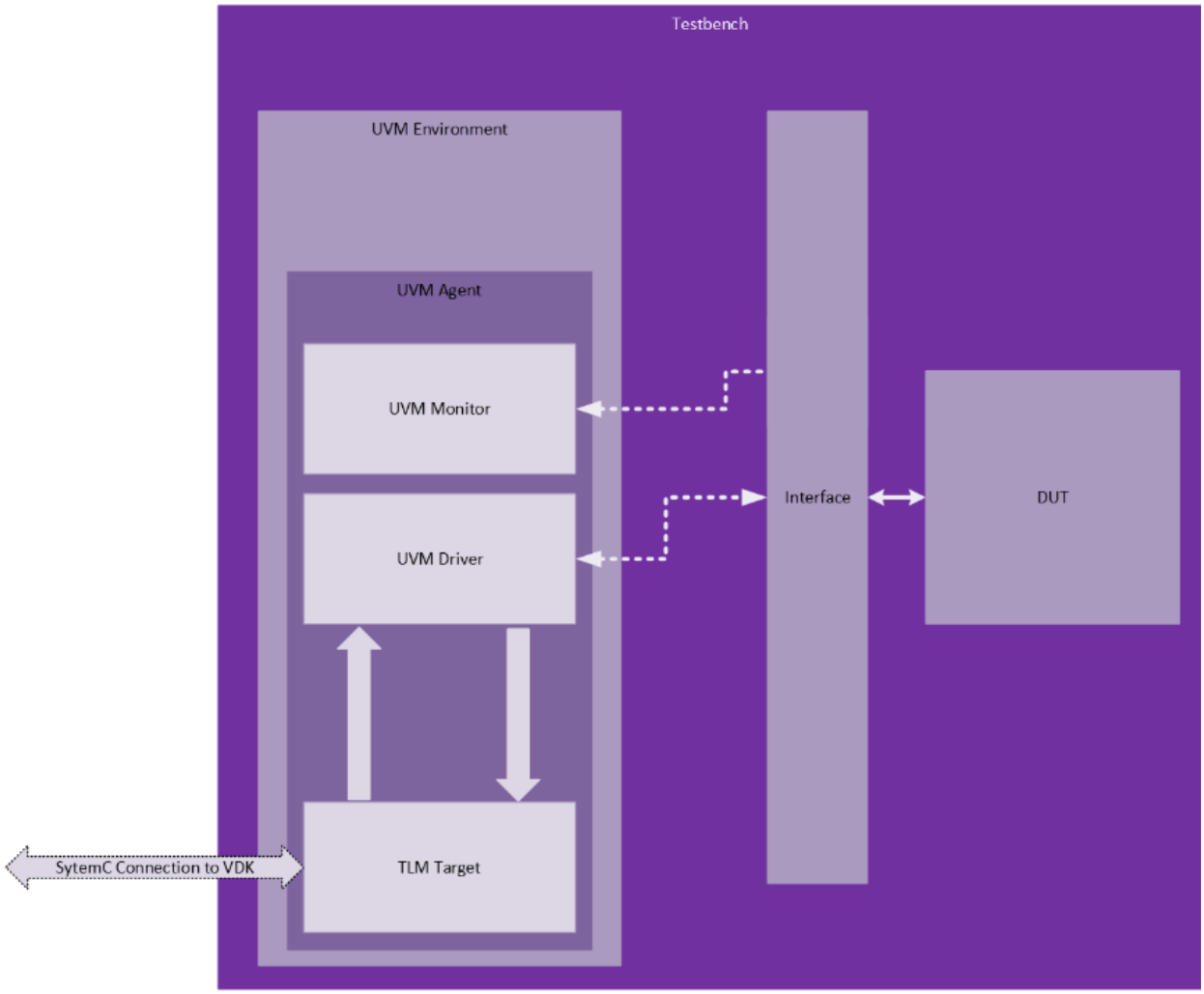

Fig 2-4 Diagrama a bloques a alto nivel de la integración completa del *TLM Target* en un ambiente de verificacion basado en *UVM*

2.2. Integrando con el modelo de SystemC, usando Virtualizer Studio

En la integración con el modelo de *SystemC* es importante recalcar el uso de *Virtualizer Studio* como herramienta que, facilita la creación de modelos en *SystemC* utilizando módulos listos para usarse y crear módulos con funcionalidad especial tales como el *uvm_interceptor*. Para este trabajo se asume que se tiene un modelo de *SoC* generado desde *Virtualizer Studio*, *VDK*, el cual tiene su reloj, *reset*, mapa e interfaces de memoria definidos, de manera que cuando estos sean mencionados en la integración con los componentes de la metodología, no serán descritos.

Adicionalmente, se asume que ya existe el respectivo *toolchain* y librerías de *SW* para poder escribir programas para el mismo.

Para la integración con el modelo de *SystemC* modificando el *VDK*, no existe un orden predeterminado para que este funcione correctamente. Sin embargo, si es altamente recomendable primero, crear la interfaz de co-simulacion, definir el reloj y *reset* a conectar con el *testbench* y definir todos los archivos y configuración para el ambiente de verificación de *UVM*.

Todos los componentes son integrados desde la *GUI* de *Virtualizer Studio*, en la perspectiva de *VDK Creation*. Para simplificar la edición del *VDK* e integración de los componentes de la metodología, se crea un grupo de bloques dentro del *VDK*. Para este caso este grupo se encuentra al nivel de los periféricos del *VDK*.

Cuando se agrega la interfaz de co-simulacion es necesario agregar la entrada de reloj que necesitara el ambiente de *UVM*, esto se logra desde el apartado de Interfaces con la opción de *Add Interface* y se hace la configuración inicial como se muestra en la imagen.

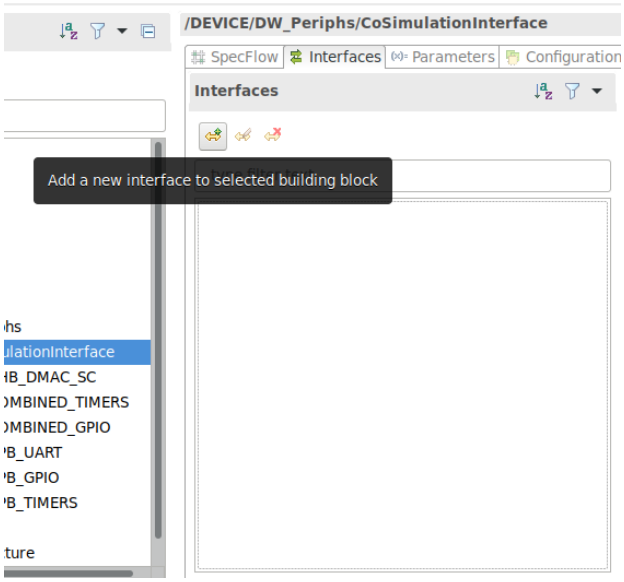

Fig 2-5 Sección de ventana en *Virtualizer Studio* para agregar el *clock* hacia el ambiente de verificacion

Lo siguiente es conectar este reloj con el reloj del sistema del *VDK*, la conexión con el reloj del *VDK* se especifica en el apartado de *External Connections*, para la sección de *Internal Connections* se recomienda hacer una configuración de *Stub Selected Interfaces*. Esto es ejemplificado por las figuras 2-7 y 2-8.

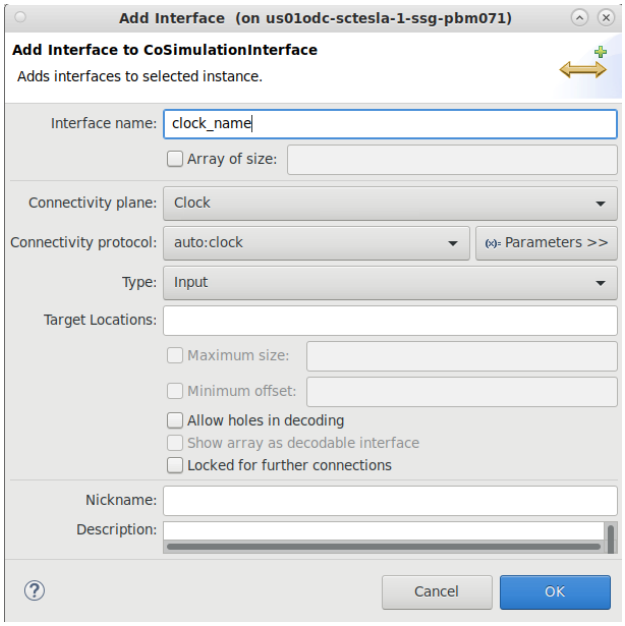

Fig 2-6 Ventana para configuración del *clock interface* que será agregado al *CoSimulationInterface* desde *Virtualizer Studio*.

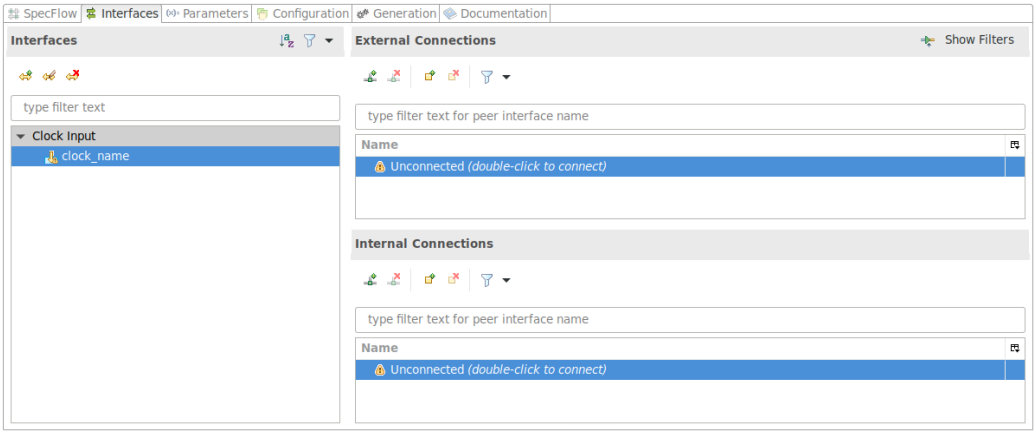

Fig 2-7 Sección de la ventana donde se realizan las conexiones entre las diferentes interfaces de reloj en el modelo, se observa la interfaz agregada en por la ventana de la figura anterior

En la pestaña de configuración es donde se especifican todos los archivos fuentes que componen el ambiente de verificación, el módulo top donde este ambiente es instanciado, la lista de directorios, la referencia al puerto de reloj que se creó anteriormente y las opciones de ejecución de simulación.

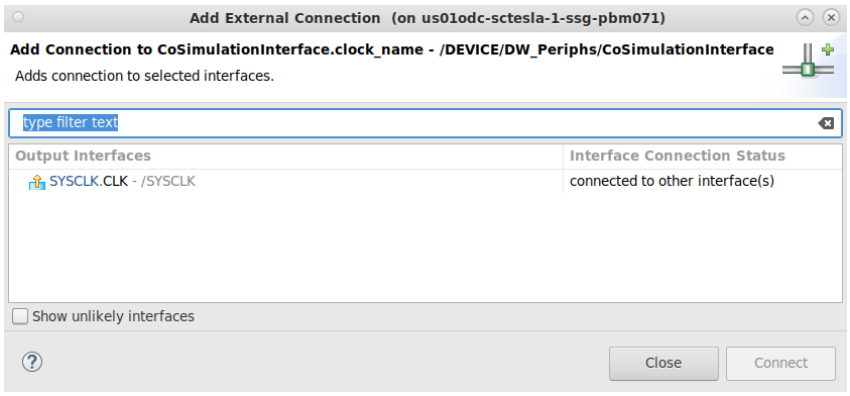

Fig 2-8 Ventana donde se muestra el ejemplo de la conexión con el reloj del modelo hacia la interfaz destinada al ambiente de verificacion

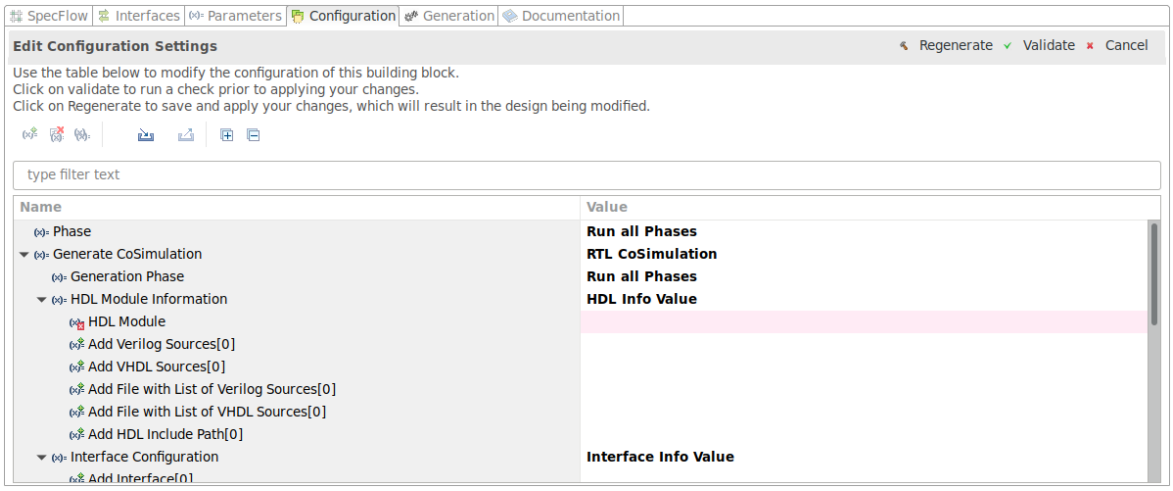

Fig 2-9 Sección de la ventana en *Virtualizer Studio* donde se realiza la configuración para integrar el ambiente de verificacion, desde archivos fuente hasta opciones de simulación

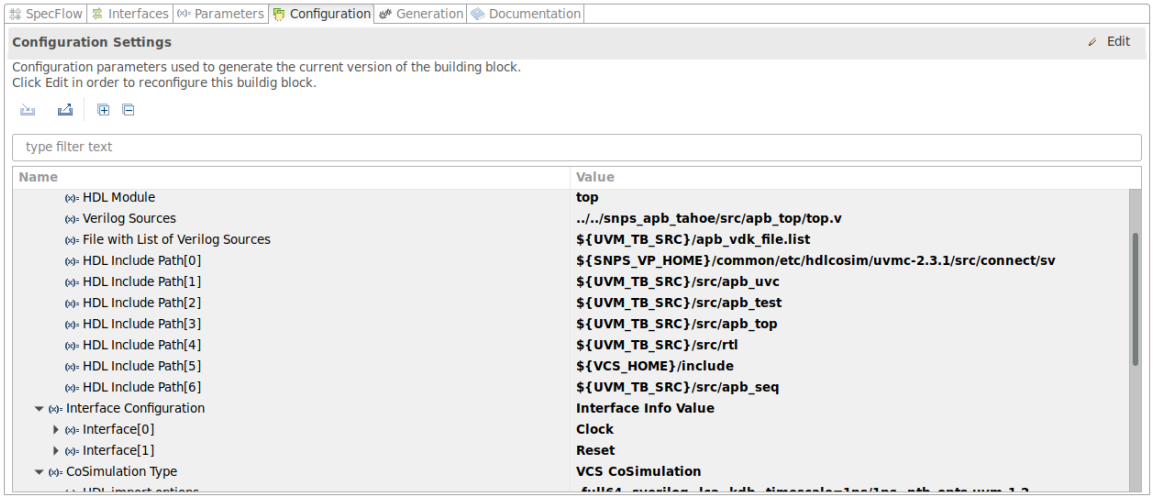

Fig 2-10 Ejemplo de especificación del ambiente de verificacion a ser usado en la interfaz de co-simulacion con el modelo en *Virtualizer Studio*

Lo siguiente es agregar el conector de *UVM Connect*, *uvmc_target_connector*, este bloque requiere de especificar la conexión del *Memory Target* hacia un *Memory Initiator*, que es el que se encuentra en nuestro *uvm_interceptor*, así como definir la referencia del identificador del conector y el tamaño del dato que manejan.

Por último, agregamos el *uvm_interceptor*. Este componente es propio de la metodología por lo cual es necesario de integrar indicando a *Virtualizer Studio* donde se encuentra la ubicación de este. El *uvm_interceptor* dentro del *VDK* solo requiere de la conexión con sus diferentes tipos de interfaces de memoria, las cuales son 3, dos de tipo *Target* y una de tipo *Initiator*. Las dos de tipo Target se dividen en *vdk_socket*, que es donde se integra al mapa de memoria del *SoC* y *uvm_target*, que no es cubierto en este trabajo, la interfaz de tipo *Initiator* es la que se conecta a nuestro *uvmc_target_connector* y es la que se encarga de ejecutar las transacciones que sean destinadas para el ambiente de *UVM*.

Para facilitar la integración definimos primero el punto de conexión en el mapa de memoria configurando el *vdk_socket*, aquí especificamos la conexión con el resto del *VDK* y a su vez, los rangos en los que existirá para el mapa de memoria, así como el tamaño de este.

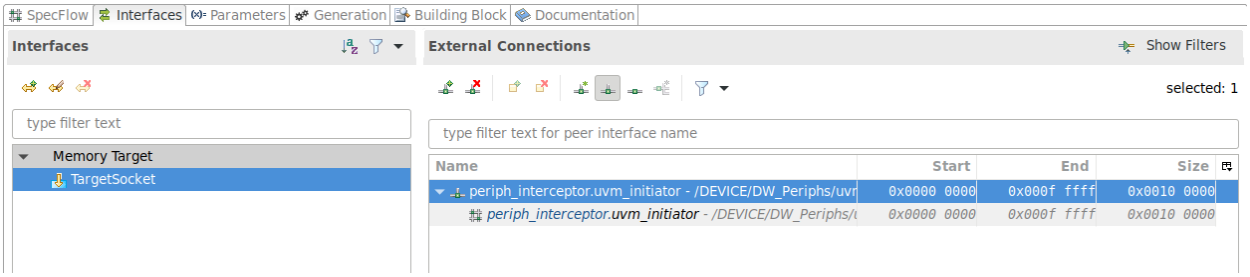

Fig 2-11 Ventana de configuración del *uvmc_target_connector*, mostrando la conexion de su interfaz de *Memory Target* con su correspondiente *Memory Initiator* en el *uvm_interceptor*

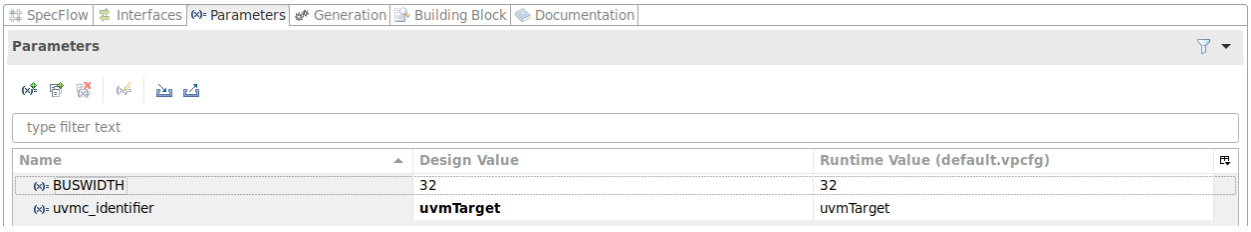

Fig 2-12 Ventana de configuración del *uvmc_target_connector*, es aquí donde se define el ancho del bus que se desea transmitir, así como del identificador que se usara para la conexión con el ambiente de verificacion

2. INTEGRACIÓN DE COMPONENTES Y FLUJO DE LA SIMULACIÓN

Una vez que tengamos esta conexión definida pasamos a generar la conexión entre el *uvmc_target_connector* y el *Initiator* de memoria que tenemos en el *uvm_interceptor*. Aquí podemos definir el mismo rango del mapa de memoria para la interfaz o dejarlo por los valores predeterminados que se dan por el tamaño de la memoria en el mismo.

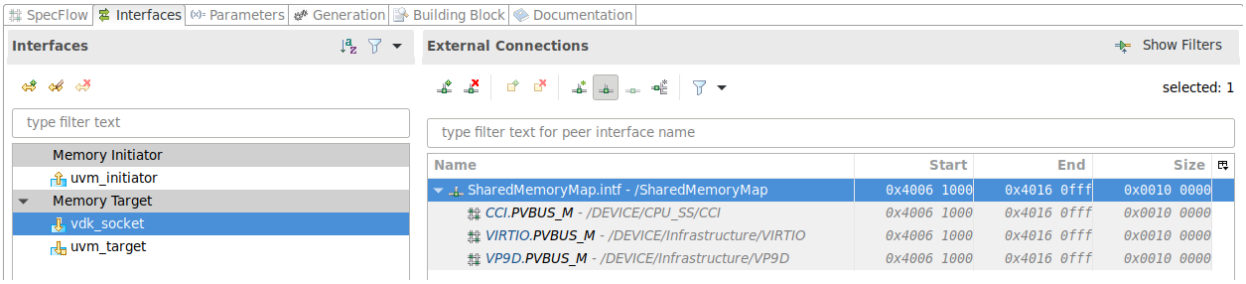

Fig 2-13 Sección de ventana en *Virtualizer Studio* donde se muestran las diferentes interfaces de memoria del *uvm_interceptor* y su respectiva conexión con el mapa de memoria del modelo

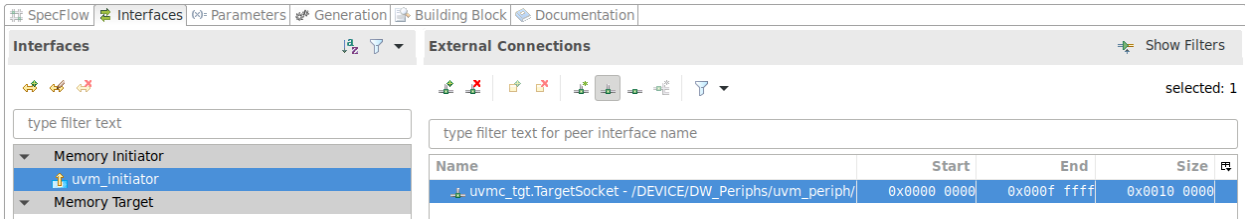

Fig 2-15 Sección de la ventana de *Virtualizer Studio* donde se hacen las conexiones entre el *uvm_interceptor* y el *uvmc_target_connector*, estas conexiones no forman parte del mapa de memoria del modelo, pero si del ambiente de verificacion

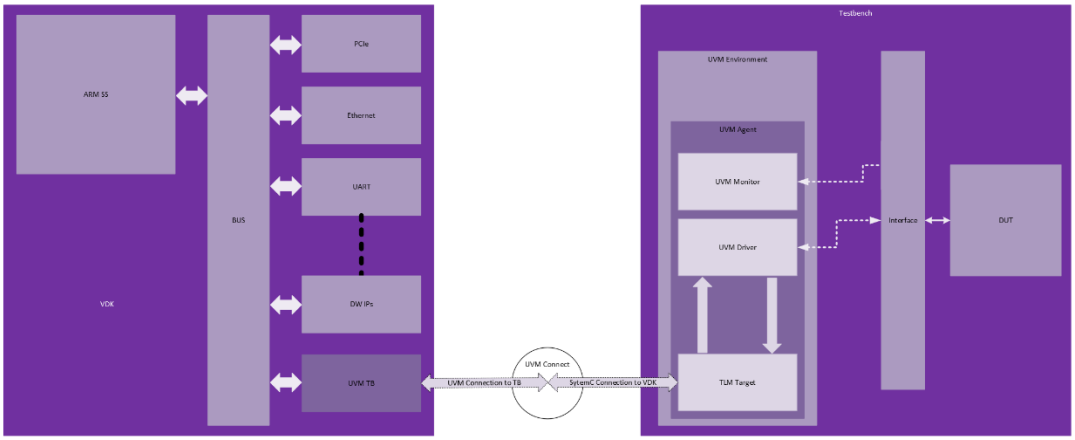

Fig 2-14 Diagrama a bloques de la integración completa entre un modelo en *Virtualizer Studio* y un ambiente de verificacion en *UVM*

Con esto finalizamos la integración con el modelo en el *VDK* y pasar a la parte de ejecución de pruebas.

2.3. Flujo de la simulación y ejecución

Antes de describir los pasos de ejecución de la simulación y obtención de resultados, es importante explicar el flujo de simulación cuando se genera esa interacción entre el modelo de *SystemC* en *Virtualizer Studio* y el ambiente de verificación de *UVM*.

En este caso tomemos el caso de querer acceder a un registro que se encuentra en un banco de registros en el ambiente de verificación desde un programa en C que se está ejecutando en el modelo en *Virtualizer Studio*. Este acceso se hace a través de una operación de lectura o de escritura a este registro.

Esta operación se da desde el flujo de programa en C, tomemos como referencia una operación de lectura-escritura. Primero, la operación de lectura se registra desde el programa de C, esta se propaga por los diferentes sistemas de buses para aplicarla en el correspondiente parte del mapa de memoria. Una vez que la transacción es aplicada al bus que corresponde al mapa de memoria donde está conectado el ambiente de verificación, esta será tomada por el *UVM* interceptor y reasignada a enviarse al ambiente mediante los conectores de *uvm connect*.

Esta transacción llega al ambiente de verificación por medio del *TLM Target*, el método de *b_transport* es ejecutado y se crea un objeto del tipo de *uvm_sequence_item* que el ambiente requiera, a este se le aplican todos los campos de dirección, datos, tipo de transacción, etc. El cual es enviado al *driver*, para que este pueda aplicar los estímulos al *RTL* con base a los contenidos de este y después obtener los datos deseados a leer.

Finalmente, el *driver* regresa al *TLM Target* los datos leídos en el mismo *uvm_sequence_item* para que este pueda reconstruir la transacción original e indicar que esta fue completada exitosamente, concluyendo la ejecución del método *b_transport*. Con la transacción terminada en el ambiente de verificación, el programa continúa con su ejecución dentro del modelo de *SystemC*.

Este mismo proceso vuelve a suceder con la operación de escritura, con la diferencia que el *driver* no espera por una respuesta de dato en el *RTL*, solo se aplican los datos y si no existió

error alguno se reporta la transacción como completada y el programa puede continuar con su ejecución.

Los siguientes diagramas de secuencia muestran el flujo de una transacción desde el modelo de *SystemC* hasta el ambiente de verificación en *UVM*.

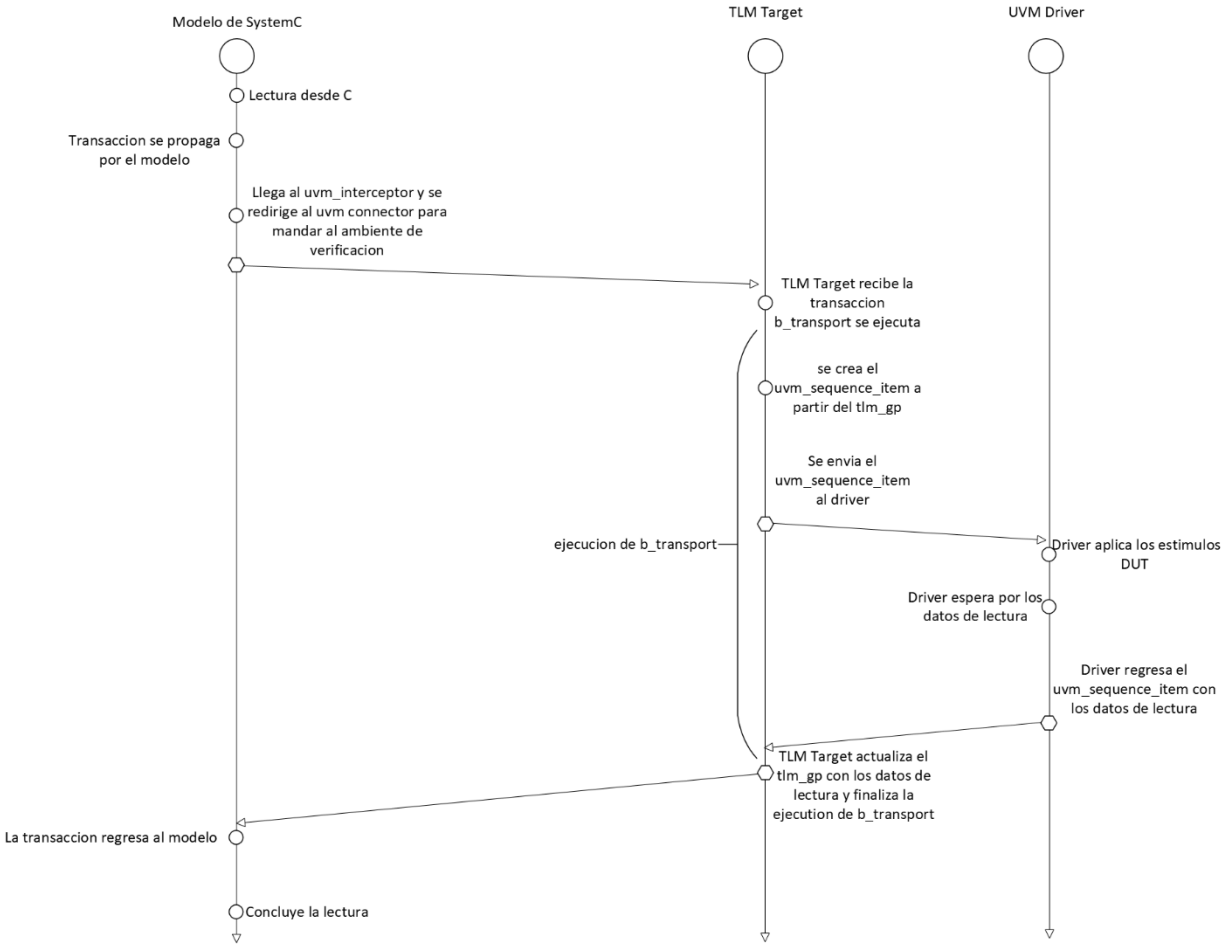

Fig 2-16 Diagrama de secuencia donde se ejemplifica el proceso de realizar una lectura desde un programa en C hasta su aplicación en el *DUT* del ambiente de verificacion

Con el flujo de transacciones explicado podemos pasar a explicar de manera breve los pasos para ejecutar la simulación de la integración de ambos ambientes. Para esto primero tenemos que hacer *build* del *VDK*, esto compilara y verificara la integración de nuestro modelo y todos los sub-bloques que lo componen. Así mismo esto generara una compilación de *VCS* de nuestro ambiente de verificacion.

Una vez que el *build* en *Virtualizer Studio* ha concluido y se obtenga un reporte libre de errores podemos pasar a ejecutar nuestra prueba dentro del mismo *Virtualizer Studio*. Es importante recordar que para este punto se asume que se tiene un flujo de compilación de C que entregue el binario necesario para nuestro modelo en *Virtualizer Studio*.

Para esto, se recomienda tener activado el *Breakpoint* de *Initial Crunch*, de esta manera podremos observar que el modelo de *SystemC* esta correctamente inicializado y los conectores de *UVMC* se asignaron correctamente a los componentes correspondientes en el ambiente de verificación, así como ver que la simulación en *VCS* ha iniciado correctamente y no existe ningún tipo de error en el ambiente o en el *RTL*.

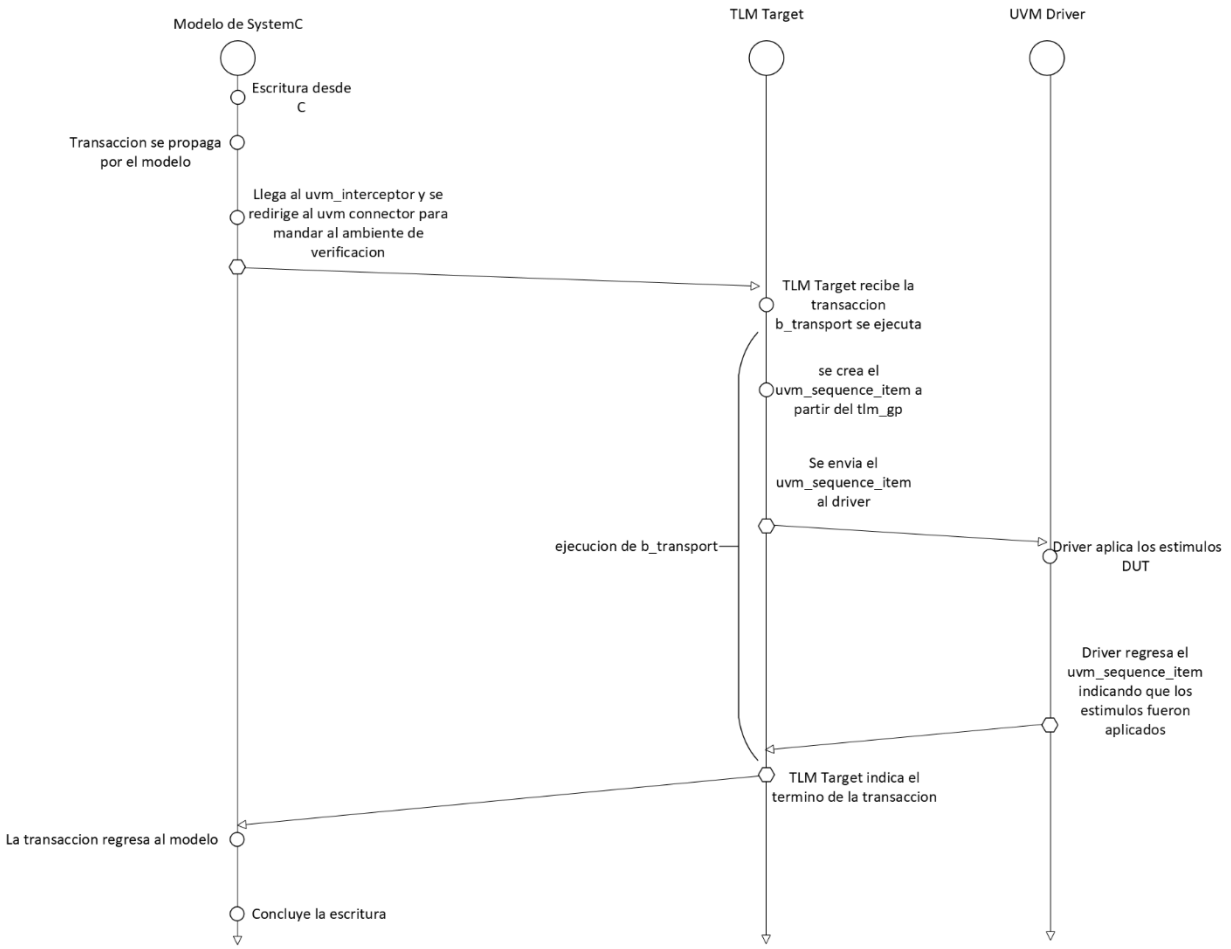

Fig 2-17 Diagrama de secuencia donde se muestra el flujo de una operación de escritura desde C y como es aplicado al *DUT* en el ambiente de verificacion

2. INTEGRACIÓN DE COMPONENTES Y FLUJO DE LA SIMULACIÓN

Al iniciar la simulación el aspecto de *Virtualizer Studio* cambiara, estaremos en la perspectiva de *VDK Debug*. Aquí podremos visualizar la estructura de nuestro *VDK*, mapa de memoria, *breakpoints* y las diferentes ventanas de visualización, ya sean ventanas de consolas seriales que tenga el modelo, gráficos, etc. Así como la ventana donde se podrá visualizar la consola donde *VCS* estará imprimiendo los mensajes de *UVM* sobre la simulación y el ambiente en general.

Una vez que la simulación haya llegado al *breakpoint* del *Initial Crunch* y comprobemos que todo haya sido inicializado correctamente podemos continuar con la simulación desde el botón *Resume Simulation* en la *GUI* de *Virtualizer Studio* y la simulación será ejecutada hasta que el programa de C haya concluido o la prueba del ambiente de *UVM* genere los mecanismos de terminación de prueba.

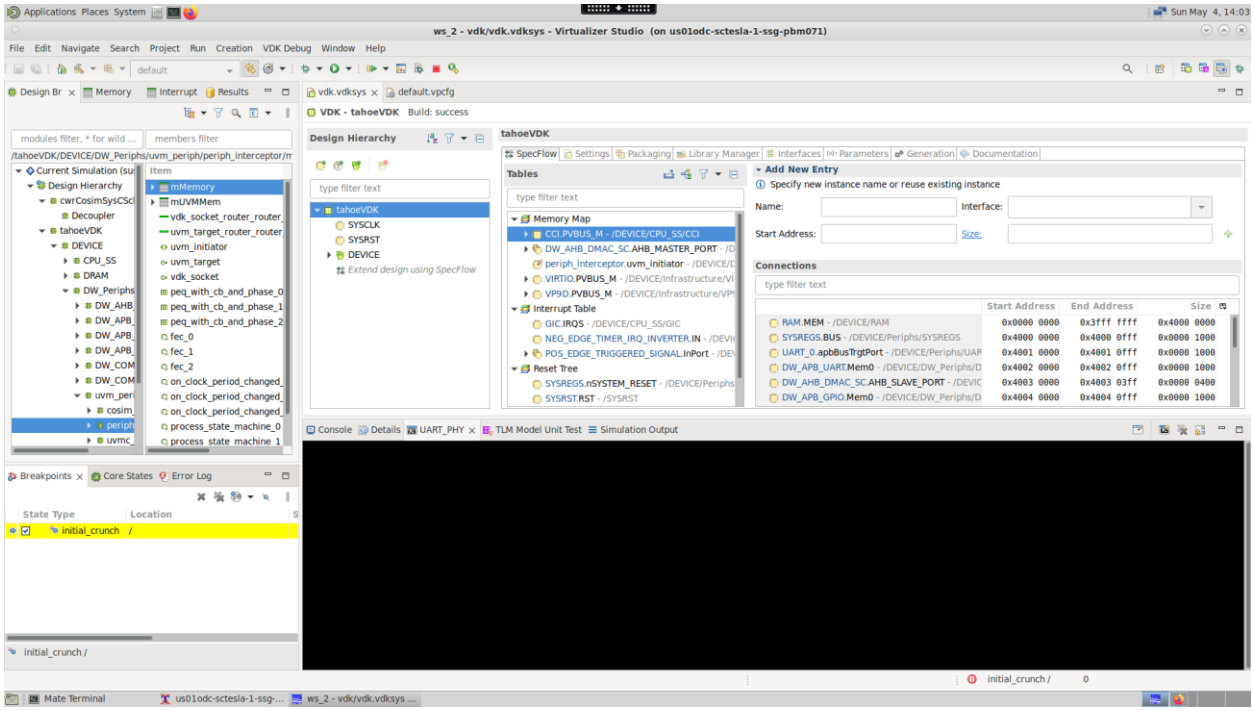

Fig 2-18 Perspectiva de *debug* al momento de correr una prueba en *Virtualizer Studio*, la prueba se mantiene detenida por el *breakpoint* de *initial_crunch*. Esta vista permite explorar y ver el estado inicial de todos los componentes del modelo

Con la simulación concluida podemos ver los archivos resultantes de la misma, logs de *VCS*, de *Virtualizer Studio*, archivos de cobertura, así como de formas de onda. Estos normalmente

se encuentran predeterminadamente en la siguiente ruta de archivos relativa al punto de origen del proyecto en *Virtualizer Studio*.

```
UVM_INFO /global/apps/vcs_2024.09-SP2-1/etc/uvm-1.2/base/uvm_report_server.svh(904) @ 500020000000: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   66
UVM_WARNING :    2
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/COMP/NAME]     2
[UVM/RELNOTES]     1
[UVM/REPORT/CATCHER]     1
[apb_cov]    16
[apb_mng_cfg]     1
[apb_mng_drv_vdk]    14
[apb_mng_mon]    14
[apb_sub_cfg]     1
[apb_sub_mon]     7
[tlm_trgt]     7
[uvm_test_top]     2

$finish called from file "/global/apps/vcs_2024.09-SP2-1/etc/uvm-1.2/base/uvm_root.svh", line 527.
$finish at simulation time         500020000000
SystemC: simulation stopped by user.
GDB stub port 12345 closed.
GDB stub port 12346 closed.
GDB stub port 12347 closed.
GDB stub port 12348 closed.
GDB stub port 12349 closed.
GDB stub port 12350 closed.
GDB stub port 12351 closed.
GDB stub port 12352 closed.
GDB stub port 12353 closed.
GDB stub port 12354 closed.
           V C S   S i m u l a t i o n   R e p o r t
Time: 500020000000 fs
CPU Time:      4.220 seconds;       Data structure size:   8.8Mb
Sun May  4 15:05:33 2025
(15:05:34) Simulation terminated with exit code 0
--------------------------------------------------------------------
```

Fig 2-19 Reporte de término de la simulación por parte de *VCS* y *UVM*, no existen reporte de mensajes de error o mensajes fatales que hayan comprometido la prueba

Conclusiones

Las nuevas tendencias en el desarrollo y necesidades que se buscan ahora de los *SoCs* y diferentes componentes de silicio, han hecho que las metodologías de verificación convencionales se hayan encontrado con ciertos retos que han sido difíciles de sobrellevar. El poder ejecutar cargas de software en etapas de *Pre-Si* permite evaluar tempranamente su rendimiento, hacer estimaciones de potencia e incluso descubrir defectos que solo se podrían presentar al estar corriendo dichos programas.

Esta metodología busca entrar en ese punto intermedio de poder reutilizar toda la infraestructura que se emplea para un ambiente de verificación y la facilidad de correr software en modelos virtuales utilizando *SystemC*. Esto nos da ventaja en no tener que recurrir siempre a los modelos de emulación y prototipado, que, si bien aportan información muy importante para el desarrollo y validación del *SoC*, sin tener que pasar por la complejidad de tener que adaptar el diseño a una plataforma donde los tiempos de simulación y depuración pueden ser elevados.

No obstante, esto implica que debemos tener listo un modelo en *SystemC* del *SoC* que aproxime el comportamiento y arquitectura que deseamos implementar, lo cual podría aumentar los requerimientos y complejidad en el desarrollo del este. Es aquí donde tenemos que hacer uso de las diferentes herramientas que nos permitan acelerar dicho desarrollo e integración.

Es importante recordar que todas las metodologías tienen sus beneficios y desventajas, si bien la metodología aquí presentada representa un gran beneficio para acelerar la verificación pre-silicio, cabe resaltar que una gran desventaja es que cualquier transacción que ocurra entre el modelo y el ambiente de verificación será ejecutada de manera bloqueante.

La otra es la escalabilidad que la metodología pueda ofrecer, si bien es posible, sería necesario comenzar a analizar el rendimiento que da la metodología e identificar los posibles escenarios donde se formen los potenciales cuellos de botella.

Mientras tanto, para un diseño pequeño a mediano, donde se desee poder comenzar a soportar el desarrollo de *software* y aportar a las pruebas de verificacion pre-silicio, esta metodología esta para aportar gran valor sin tener que recurrir al 100% a las etapas de emulación reduciendo en gran parte la complejidad y el esfuerzo requerido.

Apéndice

Bibliografía

[IEEE-23] "IEEE Standard for Standard SystemC® Language Reference Manual," in IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011) , vol., no., pp.1-618, 8 Sept. 2023, doi: 10.1109/IEEESTD.2023.10246125.

[Verification Academy-15] Verification Academy, Introduction to UVM Connect. 2015, https://verificationacademy.com/verification-methodology-reference/uvmc-2.3.0/docs/html/files/docs/OVERVIEW-txt.html

[Accelera-15] Accellera, “Universal Verification Methodology (UVM) 1.2 User’s Guide”, 8 October 2015

[Synopsys-25] Synopsys, Virtualizer: VDK Creation & Deployment Tools. 2025, https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html#virtualizer-studio

Índice