INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ESCOM

Trabajo Terminal

"Herramienta de Particionamiento de circuitos VLSI para Simulación Paralela HDL"

2014-A060

Presenta
Cuevas Sánchez Arturo

Directores
M. en C. Vega García Nayeli

México D.F. a 29 de junio del 2015

INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
SUBDIRECCIÓN ACADÉMICA

No. De TT: 2014-A060 Serie: Amarilla México D.F. a 29 de Junio del 2015

Documento Técnico

"Herramienta de Particionamiento de circuitos VLSI para Simulación Paralela HDL"

2014-A060

Presenta
Cuevas Sánchez Arturo[1]

Directores
M. en C. Vega García Nayeli

RESUMEN

En el presente documento se describe el Análisis, Diseño, Desarrollo, Implementación y pruebas del Trabajo Terminal 2014-A060 titulado "Herramienta de Particionamiento VLSI para Simulación Paralela HDL", cuyo objetivo es implementar un sistema de particionamiento de circuitos VLSI que permita dividir un circuito en componentes más pequeños, procurando siempre obtener una distribución uniforme de la carga de trabajo y un mínimo número de comunicaciones entre las entidades resultantes de forma que puedan ser simuladas mediante cómputo paralelo para reducir así la carga de trabajo en la etapa de verificación y pruebas del proceso de diseño de circuitos digitales.

Palabras Clave: Segmentación de Grafos, Problema NP-Hard, Verificación de Circuitos VLSI, Grafos, Lenguajes de descripción de hardware, Circuitos VLSI.

[1]arturo_cs.escom.ipn@outlook.es

INSTITUTO POLITÉCNICO NACIONAL
SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE FORMACIÓN INTEGRAL E INSTITUCIONAL

COMISION ACADÉMICA DE TRABAJO TERMINAL

México, D.F. a 6 de Julio de 2015

DR.FLAVIO ARTURO SÁNCHEZ GARFIAS
PRESIDENTE DE LA COMISIÓN ACADÉMICA
DE TRABAJO TERMINAL
P R E S E N T E

Por medio del presente, se informa que el alumno que integra el TRABAJO TERMINAL: 2014-A060, titulado "Herramienta de Particionamiento VLSI para Simulación Paralela HDL" concluyó satisfactoriamente su trabajo.

Los discos (DVDs) fueron revisados ampliamente por su servidora y corregidos, cubriendo el alcance y el objetivo planteados en el protocolo original y de acuerdo a los requisitos establecidos por la Comisión que Usted preside.

ATENTAMENTE

M. en C. Vega García Nayeli

Advertencia

"Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan."

La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen.

La subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n Teléfono: 57296000, extensión 52000.

Agradecimientos

A mis padres, que a pesar de las dificultades siempre estuvieron ahí, apoyándome en todo cuanto pudieron. Por su paciencia, su perseverancia, su fe, por la vida y por todo cuanto me han brindado desde el día de mi nacimiento hasta hoy que termino esta gran etapa de mi desarrollo profesional, porque hacía falta que fueran ustedes para poder yo ser quien soy ahora, y no podría estar más satisfecho con lo que he convertido y lo que he aprendido de ustedes, directa e indirectamente. Por esto y por muchas razones más, mi más sincera gratitud y respeto.

A Cristina, que estuvo a mi lado en todo momento desde que comencé con este proyecto, brindándome siempre su fe y apoyo incondicional. Por las tantas veces que me dio paz cuando mis obsesiones me tenían estancado y no encontraba la manera de continuar. Por todo lo que me brinda y todo cuanto hace por mí, aun cuando no lo sabe, y por mil razones más, mi eterna gratitud.

A todos, amigos, conocidos y docentes que directa e indirectamente han marcado mi paso por ESCOM, permitiéndome aprender de ellos, marcando mi vida y dejándome marcar las suyas de la mejor manera que pude, porque gracias a todos ustedes, familia y amigos, soy el que soy, gracias.

Cuevas Sánchez Arturo

Hoja de presentación

Título del trabajo terminal:

"Herramienta de Particionamiento de Circuitos VLSI para Simulación Paralela"

Número de registro:

Trabajo Terminal No.2014-A060

Resumen:

Este documento describe el desarrollo de una plataforma de particionamiento de circuitos VLSI para simulación en paralelo, que tiene como objetivo distribuir el proceso de simulación inherente en el diseño de circuitos VLSI descritos en Verilog particularmente, pero desarrollada de tal forma que sea útil para su implementación con diseños con características y naturaleza como las actuales (complejidad y tamaño), de tal manera que sea una herramienta útil para acelerar el tiempo de simulación y optimizar el flujo de verificación que se usa en la actualidad.

Su implementación se basa en conceptos de la teoría de grafos, haciendo uso de una estrategia de particionamiento con el fin de poder modularizar el circuito en cuestión teniendo como factores clave la distribución uniforme de la carga de trabajo y el mínimo posible de comunicaciones entre tales módulos, con lo cual podría considerarse como una herramienta potencial para el alivio del elevado poder de cómputo y la demanda de memoria que actualmente se requiere para la simulación de circuitos VLSI en el ámbito industrial.

Palabras clave:

Segmentación de grafos, Verificación de circuitos VLSI, Grafos, Lenguajes de descripción de hardware, Circuitos VLSI.

Participantes:

Cuevas Sánchez Arturo

Directores:

M. en C. Vega García Nayeli

Índice

Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1 Introducción

En la actualidad los circuitos son el producto industrial más importante, dirigido a un gran mercado, las empresas hacen grandes esfuerzos por producir el mejor chip con el menor costo en el menor tiempo posible [1]. Dado que la demanda de dicho mercado ha incrementado considerablemente, la competencia es cada vez más fuerte y por ello los circuitos son cada vez más grandes y complejos en cuanto a diseño, e integran en gran escala circuitos, combinando miles de transistores en un solo chip (*VLSI, ULSI*), esto por la necesidad de satisfacer más tareas y necesidades con un solo circuito, incrementando con ello el mercado de la industria.

Con el paso del tiempo y la evolución de las tecnologías, la problemática de diseñar un circuito de grandes capacidades que resuelva tareas complejas ha sido sustituida por la problemática consistente en la verificación de dichos diseños, pues probar el funcionamiento de un circuito se ha vuelto un cuello de botella muy costoso debido a que prolonga de manera considerable el *time-to-market*, generando con ello una pérdida de tiempo y dinero a quien se dedica a la fabricación de estos diseños. En los inicios de la globalización de éstos productos surgieron muchos enfoques y soluciones que resolvieron paliativamente el problema de la verificación, como los lenguajes *HDL*, tarjetas *FPGA* para programar y verificar el comportamiento del diseño; o herramientas *CAD-EDA* avanzadas que brindan toda una gama de recursos para analizar y estudiar un diseño. Desafortunadamente ninguna de ellos ha sido suficiente ya que estos circuitos incrementan de manera considerable sus características en poco tiempo, y con ello sus necesidades/demandas en cuanto a la verificación se refiere.

Por ello, en cuanto las computadoras comenzaron a tener las capacidades necesarias, se tomó un nuevo enfoque para la fase de pruebas del proceso de diseño, dicho enfoque es la simulación, con esto, una máquina se encarga de hacer todas las pruebas necesarias para considerar que cada sección de cada circuito funciona correctamente, sin embargo, los circuitos continúan creciendo. Tan solo para la simulación de un periodo de tiempo de 20ns. de un circuito relativamente pequeño, de alrededor de 20,000 transistores, una computadora de un solo procesador tardaría 10 hrs. en promedio [1].

Tomando en cuenta que los sistemas computacionales no han evolucionado a la par que el diseño de circuitos VLSI, y tomando en cuenta la necesidad de interacción entre sistemas de cómputo, surgieron aplicaciones para comunicaciones en red, que permiten la interacción entre diferentes procesos y diferentes sistemas de cómputo. Gracias a esto surge un nuevo enfoque para la tarea de probar funcionalmente *(testing)* el diseño de un circuito, éste es, la simulación paralela, que consiste en *particionar* un circuito de tal manera que diferentes máquinas puedan simular el comportamiento de una entidad menos compleja.

El *particionamiento* es una técnica para dividir un circuito/diseño o un sistema en una colección de partes más pequeñas (componentes). Debido a la naturaleza de los circuitos

actuales, resolver o disminuir la complejidad y dificultad de los problemas de optimización que implica la fase de *testing* en el proceso de diseño se ha convertido en una tarea central o crítica a ser considerada [2]. Cabe mencionar que el *particionamiento* tiene muchos enfoques, pero para nuestro ámbito, se consideran generalmente dos, el *particionamiento* para *FPGA´*s y el *particionamiento* en cuanto al grafo que describe el comportamiento del circuito (grafo característico), como se muestra en [2].

Tomando en cuenta la complejidad inherente en este tipo de diseños, *particionarlos* puede considerar diferentes enfoques y técnicas, pero los resultados no son alentadores, como se menciona en [3], ya que siguen existiendo muchos factores que alteran los resultados, por lo cual se toma el enfoque con base en el grafo característico, que es una forma de representar el comportamiento de un diseño de manera gráfica. De este modo se desechan los factores que alteraran los resultados y se trabaja con un esquema uniforme que puede ser aplicado a la mayoría de los circuitos de escalas actuales.

Aun teniendo un enfoque uniforme que pueda tratar la mayoría de diseños, siguen existiendo muchos factores que deben ser considerados para realizar el particionamiento de un circuito, como el tiempo de procesamiento que demanda la simulación, el menor número de comunicaciones entre componentes para reducir problemas de sincronización y la carga uniforme de trabajo, que consiste en dividir los componentes de forma que cada uno de ellos demande una cantidad proporcional de recursos computacionales y así lograr que la mayoría trabaje con la misma cantidad de recursos y el mismo tiempo. A causa de ello surgen otros factores importantes indirectos la granularidad de corte, que se refiere a qué tan fina será la división, ya que se puede dividir un diseño a nivel de compuerta, lo que resultaría ineficiente y demandaría más estaciones de trabajo.

1.1 Antecedentes

La extraordinaria complejidad que alcanzan hoy en día los circuitos digitales y los sistemas digitales complejos repercute fuertemente sobre la comprobación de su correcto funcionamiento, lo cual resulta sumamente difícil. Dentro del proceso de diseño de los sistemas digitales se incluyen varias etapas entre las cuales se encuentra la prueba del funcionamiento lógico del diseño.

Dicha etapa resulta ser la etapa crítica del proceso de diseño de sistemas digitales debido a los tiempos de producción y seguridad del producto final, si esto no es satisfactorio se pueden generar altos costos de verificación o corrección.

Algunas fallas históricamente significativas son:

- **Ariane 5.** Fue un cohete de un sólo uso, diseñado para colocar satélites en órbita geoestacionaria y para enviar cargas a órbitas bajas. Explotó en su primer vuelo a causa del reúso de algunas partes de un código desde su predecesor sin una verificación apropiada.
- **Therac-25**. Era una máquina de radioterapia producida por *AECL*. Debido a un error de software, seis personas murieron por sobredosis. Se estima que la radiación obtenida por los damnificados fue de al menos cien veces la esperada.
- **Pentium FDIV**. Error de diseño más famoso en la carrera de Intel con respecto a la unidad de punto flotante. Después de negar su responsabilidad y muchos sucesos más, la compañía se vio obligada a ofrecer reemplazos de todos los procesadores defectuosos.

La verificación es por naturaleza un proceso comparativo. Esta engloba un amplio espectro de técnicas para descubrir las fallas en el funcionamiento del dispositivo. La verificación funcional no revela la falla en sí misma, sino que revela la presencia de error [3].

Debido a esto y muchas otras razones, la tarea de verificación ha sido explorada por una gran variedad de equipos de trabajo utilizando múltiples enfoques con el fin de lograr una verificación eficiente y más completa asegurando así un producto de calidad y confiable, con los menores gastos posibles.

1.2 Planteamiento del Problema

Como se menciona en [4], la creciente cantidad de transistores integrables en un chip ha dado lugar a nuevas categorías de diseños como ULSI y VLSI en los cuales se añade la dificultad de diseñar arquitecturas correctas de tal manera que, a pesar de su escala de integración, sean completamente funcionales y se desempeñen de acuerdo a lo esperado.

La etapa de verificación en el proceso de diseño de circuitos digitales es la etapa más importante pues de ella depende el curso del producto y, por tanto, es la que demanda más tiempo ya que se requiere simular el dispositivo en todos los escenarios posibles de tal manera que se asegure su correcto funcionamiento. Como se ha mencionado, un descuido en este proceso puede provocar repercusiones significativas para la empresa y los usuarios del producto en cuestión.

Es por ello que se requieren nuevos enfoques y técnicas para hacer una ejecución eficiente de esta etapa. Una técnica de simulación distribuida permitiría reducir considerablemente los costos inherentes en dicha etapa del diseño por lo que se requiere realizar un particionamiento eficiente del mismo, es decir, asegurar que la distribución de la carga de trabajo y el costo de comunicación entre las particiones sea el mínimo posible y esté distribuido de manera uniforme, de tal manera que los resultados y los costos implicados sean los mejores posibles.

1.3 Solución Propuesta

Diseñar e implementar una plataforma capaz de recibir un diseño descrito en lenguaje *HDL* (*Verilog* y *System Verilog* tentativamente*)* que, mediante el proceso de pre-simulación y tomando como base algunos criterios, pueda obtener el comportamiento de dicho diseño. Así, la carga de trabajo de sus componentes y la cantidad comunicaciones entre ellos formarán su grafo característico para posteriormente particionarlo mediante algún algoritmo, de tal manera que, una vez particionado, cada módulo pueda ser simulado en un nodo diferente de un sistema distribuido disminuyendo el cuello de botella inherente en la etapa de verificación del proceso de diseño de circuitos VLSI, reduciendo de manera considerable el *time-to-market* y los gastos/costos implicados en el proceso.

1.4Objetivos

1.4.1 Objetivo General

Diseñar e implementar un sistema de particionamiento de circuitos *VLSI* que permita dividir un circuito en componentes más pequeños, tomando como base los parámetros determinados mediante la técnica de pre-simulación que serán la entrada para el algoritmo de particionamiento, procurando, así, obtener una distribución uniforme del trabajo y un mínimo de comunicaciones entre cada una de las entidades resultantes, tomando el nivel de módulo como granularidad máxima de corte.

1.4.2 Objetivos Específicos

- Elegir los parámetros indicadores del comportamiento del diseño utilizando la técnica de pre-simulación como herramienta para la generación del grafo característico.
- Modelar el diseño a través de un grafo dirigido con base en las características del diseño para dar pie al algoritmo de particionamiento.
- Elegir un algoritmo de particionamiento que garantice una adecuada distribución de la carga de trabajo y la minimización de comunicaciones entre las entidades.
- Programar y probar el algoritmo de particionamiento elegido.

1.5 Justificación

El mayor de los problemas en el proceso de diseño de un circuito *VLSI* es el cuello de botella que se genera en la fase de pruebas, ya que estos circuitos siguen creciendo en magnitud, complejidad, demanda de memoria, ciclos de cómputo, etc. El tiempo requerido para esta fase ha superado la capacidad de los equipos y mecanismos usados actualmente para tal efecto.

La simulación distribuida aprovecha al máximo las plataformas de cómputo paralelo con el fin de cubrir las demandas de memoria y cómputo, lo cual la convierte en la alternativa más viable. Ésta sólo funciona si se hace con base en el *particionamiento* eficiente del circuito, con lo cual se facilitan las implementaciones de los algoritmos de sincronización, eventos e intercambio de información, el balance de carga entre procesos, etc. con el fin de acondicionar modelos de cualquier rango para ser estudiados bajo este esquema [5].

Capítulo 2 Marco teórico y Estado del Arte

2.1 Ciclo de Diseño de Circuitos Digitales

Actualmente los dispositivos VLSI son circuitos integrados modificados con requisitos particulares para un uso concreto en lugar de su aplicación de uso general. La realización de estos productos tiene un proceso tan largo como el descrito en [6].

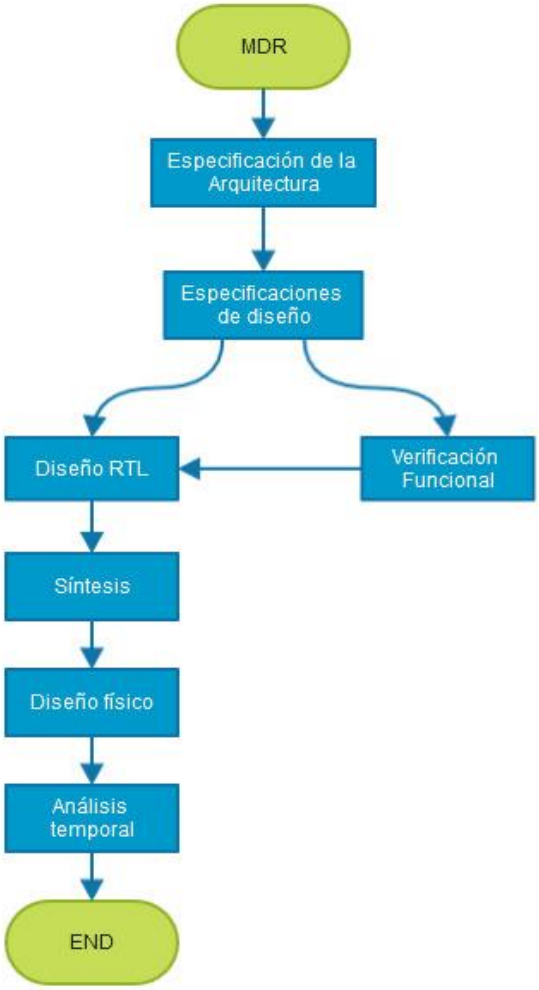

Figura 1 Proceso de diseño de VLSI's.

En la figura 1 [6] se muestra el proceso de diseño *VLSI*. Éste comienza con un *MDR* (Documento de Requisitos del Mercado) donde se describen los requisitos del producto en

cuestión, es decir, especificaciones importantes como las necesidades a cubrir, el valor propuesto, requerimientos, comportamiento, etc.

El siguiente paso del proceso de diseño es la especificación de la arquitectura, en el cual un equipo de diseñadores VLSI estructuran o crean la arquitectura que mejor se adapte a las especificaciones establecidas y funcione de manera eficiente. Para esto aterrizan las especificaciones en un diseño dando paso a las especificaciones del producto donde los arquitectos y diseñadores forman planes de trabajo mediante documentos detallados en los cuales describen estrategias, particiones del diseño, tipos de recursos a utilizar, etc. Esta arquitectura es implementada en algún lenguaje de descripción de hardware (HDL's) para conocer las dimensiones del proyecto y su alcance [7].

Una vez que se tiene la especificación de la arquitectura se da paso a la etapa crítica del proceso de diseño que es la verificación, para lo cual se requiere un plan de verificación que parte de toda la información disponible del producto para identificar de qué manera se verificará el correcto funcionamiento del dispositivo, es decir, comprobar que cumpla sus objetivos.

Después de esto se procede al diseño *RTL* donde el diseñador implementa el producto en algún *HDL* para dar paso a la verificación funcional que es la tarea más compleja del proceso de diseño de circuitos *VLSI* en donde, mediante un banco de pruebas, los encargados se dan a la tarea de verificar si el *DUT* trabaja de manera eficiente y si lo hace con base en las especificaciones establecidas anteriormente.

Si los resultados no son los esperados al terminar el proceso de verificación se repite lo anterior, de lo contrario se procede a la síntesis, es decir, el proceso de tomar el diseño para compilarlo y así obtener su especificación a nivel de compuertas, esto es, a un nivel más físico.

Por último se procede a desarrollar el diseño físico, es decir, la organización lógica, la planificación de suelo, el enrutamiento global, la compactación y el enrutamiento detallado con base a los resultados dando paso al análisis temporal para estudiar el dispositivo físico y conocer su rendimiento real, los tiempos de instalación, espera, recuperación, eliminación, latencias, etc.

2.2 Verificación de Circuitos Digitales

La etapa de verificación consume entre el 50% y el 70% del trabajo realizado en todo el ciclo de diseño y es un paso crítico en el flujo de diseño cuando los productos VLSI tienden a los varios millones de compuertas, por lo que la verificación es el principal cuello de botella en lo que respecta al proceso de diseño de circuitos. Este cuello de botella es consecuencia del aumento del nivel de abstracción del diseño. La mayoría de los diseños requieren un nuevo cambio de paradigma de implementación del diseño, ya que el 71% de los cambios de paradigma realizados se deben a fallos funcionales.

2.2.1 Verificación funcional

El objetivo de la verificación funcional es detectar fallas de manera que puedan identificarse y corregirse antes de entregar el producto. Si el diseñador RTL comete un error en el diseño o la codificación, esto se convierte en un error en el chip. Si no se corrigen los errores detectados, por mínimos que parezcan, pueden producir resultados no deseados, provocando, así, el fracaso del producto. No todos los errores ocasionan el fracaso del producto y, en ocasiones, un simple error puede dar lugar a una amplia gama de insuficiencias.

No todos los problemas son causados por errores de codificación, existe la posibilidad de que un error sea producto de la propia especificación, es decir, de la falta de comunicación entre los equipos de desarrollo puede dar paso a errores de diseño.

Una herramienta muy útil en el proceso de verificación, y con frecuencia la más usada, es el *TestBench*. Un *TestBench* es un banco de pruebas que imita el entorno del dispositivo en cuestión, es decir, simula una instanciación del dispositivo de manera que pueda ser manipulado simulando un producto terminado para comprobar si la aplicación RTL cumple con las especificaciones del diseño.

Este entorno crea condiciones válidas e inválidas, así como eventos esperados e inesperados para poner a prueba el diseño.

Generalmente los *TestBench* realizan las siguientes tareas:

- Crear una instancia del diseño bajo prueba (DUT).
- Estimular al dispositivo mediante vectores de prueba.
- Mostrar los resultados de salida mediante una forma de onda o mensajes en la terminal para su inspección visual. Para ello el encargado de la prueba verifica los resultados comparándolos con los esperados.

El *TestBench* simple es la técnica de verificación/simulación más antigua y requiere mayor mano de obra, por ende, la calidad de la verificación depende de la determinación y la dedicación de la persona encargada de la comprobación. Esto no presenta un modelo

práctico para la verificación cuando se trata de un diseño complejo, y cada vez que se realicen cambios o correcciones sobre el diseño se supondrá la misma cantidad de esfuerzo para cada verificación correspondiente.

Otro tipo de *TestBench* son los llamados "de autocomprobación", los cuales comparan los resultados obtenidos en la simulación contra los resultados reales o esperados. Aunque este tipo demanda un mayor esfuerzo durante su fase inicial de creación, esta técnica puede reducir considerablemente la cantidad de esfuerzo requerido para volver a comprobar el *DUT* después de cada cambio. El tiempo de depuración se reduce considerablemente por el seguimiento de errores que puede integrarse en el mismo para conocer en qué punto está fallando el diseño.

Este tipo de *TestBench* tiene 2 componentes principales, los bloques de entrada y los de salida. El bloque de entrada consiste en el estímulo y el *Driver* que se encargará de dirigir dicho estímulo al *DUT*, mientras que el bloque de salida se compone de un monitor que recoge las salidas del *DUT* y las verifica. Estos enfoques requieren de alguien que escriba las pruebas explícitas que se harán a cada característica del diseño. El enfoque en el que cada función es descrita en un archivo de prueba separado es llamado verificación dirigida.

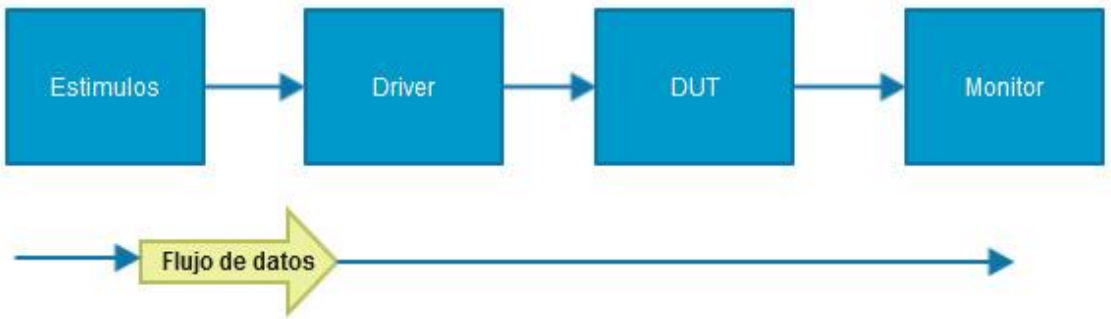

Figura 2 Componentes de un TestBench auto verificado.

En la figura 2 [6] se muestra la arquitectura de la verificación dirigida (una variante de la verificación común), donde el ambiente de verificación tiene un mecanismo para enviar el estímulo al DUT y recoger las respuestas para ser comprobadas. Los estímulos se generan como casos de pruebas, en éstos se puede utilizar una cantidad limitada de asignaciones al azar, a menudo, mediante la creación de valores aleatorios en lugar de simples asignaciones con valores predeterminados ayuda a una verificación más eficiente del diseño. Esto es un proceso tedioso cuando la complejidad del DUT aumenta, pues es más difícil crear patrones que existen el diseño y el mantenimiento de éstas unidades se vuelve pesado y lento.

En éste tipo de verificación, es el diseñador quien se encarga de escribir los escenarios, pero se puede dar el caso en el que no se contemplen todos los que se pueden presentar ni todos los posibles errores, por lo que hay una gran probabilidad de que los errores sucedan. Algunos de estos errores aparecen más tarde en el proceso de diseño o simplemente aparecen una vez que el producto ha sido fabricado.

Una solución a éste problema es la verificación aleatoria de restricciones, de ésta manera, los estímulos necesarios para verificar el DUT se generan automáticamente. El diseñador indica el conjunto de especificaciones y el *TestBench* crea automáticamente un espacio de soluciones y recoge los escenarios posibles de dicho espacio.

La verificación dirigida también reduce el esfuerzo manual y el código para las pruebas individuales, puesto que al generar los casos de prueba de manera automática se pueden generar todos o casi todos los escenarios posibles, con esto y tomando en cuenta que algunos casos son similares en cuanto a la lógica, los encargados pueden encontrar errores incluso antes de que el *TestBench* se haya completado.

Figura 3 Generación de verificación aleatoria.

Como se muestra en la figura 3 [6], generar completamente al azar las pruebas no tiene sentido puesto que se pueden generar escenarios repetidos desperdiciando el potencial del entorno y malgastando el tiempo de simulación. El diseñador debe especificar estructuras de datos para llevar un control de dichos escenarios y especificar las restricciones para orientar al generador aleatorio. La principal desventaja de recurrir a éste método es que no se sabe lo bien que se verifica el DUT, es decir, si el diseñador tuviese la manera de obtener información al respecto, podría generar escenarios específicos para poner a prueba la parte de la lógica o el diseño que no ha sido verificada.

Para todo esto surge el concepto de cobertura que es definida como el porcentaje de los objetivos de verificación que han sido cumplidos.

2.2.2 Cobertura

La cobertura de código se utiliza para medir la eficiencia de la ejecución de la verificación. Mediante una medida cuantitativa del espacio de pruebas, se describe el grado en que ha sido probado el código fuente del DUT, también se le conoce como cobertura estructural

Mediante la aplicación de técnicas de análisis de la cobertura de código para *HDL's*, la eficiencia de la verificación se ha mejorado al permitir que el diseñador pueda aislar áreas de código no probado. Mediante el informe de cobertura, el diseñador puede encontrar los valores bajos e identificar porqué ese código en particular no se ha probado completamente y con ello puede escribir más pruebas o dirigir la aleatoriedad para cubrir dichas áreas.

En la verificación a nivel de unidad, se verifica modulo por módulo en su propio entorno de prueba para demostrar que la lógica de control y de flujo de datos funciona correctamente. El objetivo de la verificación a nivel de módulo es comprobar que cada componente probado se ajusta a sus especificaciones y está listo para ser integrado con otros sub-componentes del producto.

Por todo esto, la cobertura de código se convierte en un criterio para el acabado de las pruebas ya que tiene que verificar todas las características de cada componente. En el nivel sub-sistema, el objetivo es asegurar que las interfaces entre las unidades son correctas y las unidades trabajan juntas para ejecutar la funcionalidad general, pero no es muy útil pues la verificación no va dirigida a todas las características de la unidad.

Algunos criterios de cobertura de código son:

- **Cobertura por declaración/línea:** Éste es un concepto básico y simple de cobertura, donde la métrica consiste en definir, del total de líneas de código que describen el DUT, cuántas han sido evaluadas. Para éste caso, se consideran solo las líneas ejecutables, independientemente de la sentencia, y las líneas no ejecutables como las que determinan el fin de alcance de una función o sentencia y los comentarios no se toman en cuenta puesto que no forman parte intrínseca del diseño.

- **Cobertura por bloque/segmento:** Es similar a la anterior, la diferencia radica en que considera como bloques a las declaraciones de operación ramificadas como *if*/*else*, las sentencias *case*, *wait*, *while* y ciclos *for*. Es una técnica que revela el código muerto de un diseño RTL.

- **Cobertura condicional:**También llamada cobertura por expresión, revela cómo se evalúan las variables o sub-expresiones en las sentencias condicionales, sólo se consideran las expresiones con operadores lógicos. Su desventaja radica en que no tiene en cuanta cómo se consigue el valor booleano a partir de las condiciones. Se

considera como la proporción del número de casos verificados del total de número de casos presentados.

- **Cobertura por saltos:** La cobertura por ramas solo se usa para saber si en las sentencias *if* se han analizado o comprobado todos los valores booleanos de todas las ramas.

- **Cobertura por cambios de estado:** Dado que las sentencias y condiciones alteran el flujo del estímulo a una ruta de acceso específica, se considera como la más completa de las métricas de cobertura ya que puede detectar los errores relacionados con la secuencia de operaciones. Se puede hacer en cada bloque de función, pero no alcanza a abarcar a más de un bloque puesto que los caminos son diferentes y pueden ser representados de manera iterativa, por lo que se hace bloque a bloque para realizar la cobertura de manera eficiente.

- **Cobertura de enlace:** Se utiliza en el caso de máquinas de estados para comprobar la inter-relación entre nodos y el direccionamiento del estímulo.

- **Cobertura FSM:** Es el tipo más complejo de cobertura ya que trabaja sobre el comportamiento del diseño, hace uso de la cobertura de máquinas de estados finitos para encontrar todos los errores posibles en éste tipo de máquinas de estados. En éste tipo de cobertura se busca cuantas veces se visita cada estado.

Ésta cobertura se compone su vez de 2 campos, la cobertura de estados que indica el número de estados visitados con respecto al número de estados presentes en el DUT y la cobertura de transición que indica el número de transiciones que se hacen de un estado a otro con respecto al número total de transiciones realizadas.

2.2.3 Cobertura funcional

Independientemente del protocolo de pruebas que se ejecute sobre un diseño, es cada vez más difícil saber cuándo parar dichas pruebas, debido a la naturaleza en cuanto a complejidad y tamaño de los diseños actuales. En el entorno de pruebas, se debe especificar un plan de análisis para cada punto de vulnerable o susceptible a ser analizado, de forma que al final todos los puntos son probados con las pruebas pertinentes declaradas en su respectivo plan y entonces se puede decir que se ha cubierto de manera satisfactoria la verificación del dispositivo. Cabe mencionar que esto es un proceso manual, es decir, el diseñador de las pruebas las codifica por separado con base a las especificaciones del dispositivo, puesto que es aquí donde se indica el comportamiento esperado del mismo y los valores aceptables que tiene cada variable.

Las principales cualidades de la cobertura funcional son las siguientes:

- Ayuda a determinar qué parte de la especificación he sido cubierta.
- Cualifica los TestBench, es decir, les añade características que los hace una herramienta de verificación más eficiente.
- Se considera como un criterio de paro en el nivel de unificación.
- Brinda información acerca de las características aún no probadas.
- Brinda información sobre las pruebas redundantes que consumen ciclos valiosos.

Con lo que se convierte en una guía para alcanzar los objetivos de la verificación basado en calificaciones o valores cuantitativos.

2.2.3.1 Plan de verificación

El plan de verificación es el punto focal para definir exactamente lo que tiene que ser probado así como los criterios de cobertura. El éxito de un proyecto de verificación depende en gran medida de la integridad y la aplicación precisa de un plan. Un plan de verificación puede ser definido de muchas maneras, tales como hojas de cálculo, documento o archivos de texto simple. Las plantillas son buenas si se utilizan continuamente en al ámbito industrial ya que lo convierte en una interfaz común para la información y el encargado de la revisión puede identificar fácilmente donde buscar qué valores.

Como bien sabemos, por eficiente que sea una herramienta al generar un TestBench, ésta solo creará un "cascarón" del mismo, y es responsabilidad del encardado de las pruebas programar el comportamiento que ha de ser simulado para comprobar el funcionamiento del diseño, por tanto, para una correcta verificación es necesario que quien entrega el diseño deba proporcionar además una especificación de dicho diseño, de tal manera que, si el encargado de las pruebas no conoce el diseño pueda saber qué pruebas son las que tiene que hacer para obtener el comportamiento especificado, es decir, de ésta manera, el encargado de las pruebas sabe qué entradas proporcionar el diseño y qué salidas se esperan respecto a dichas entradas.

Los componentes de un plan de verificación son:

- Revisión
- Recursos, presupuesto y calendario.
- Verificación del entorno.
- Comprobación del flujo.
- Extracción de características.
- Plan de generación de estímulos.
- Plan de análisis.
- Plan de cobertura.

- Detalles de componentes reutilizables.

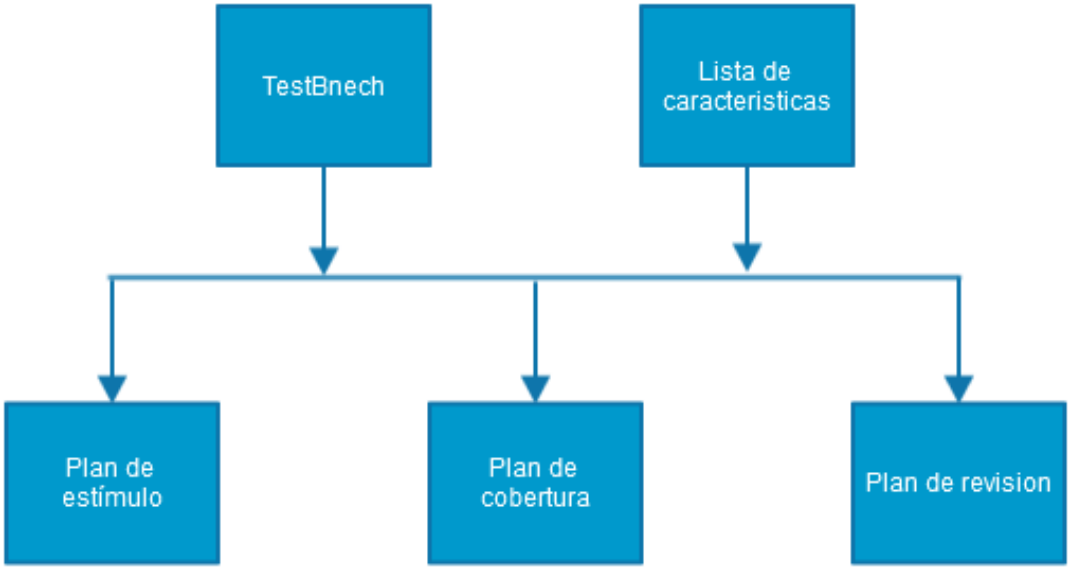

Figura4 Componentes del plan de verificación.

En la figura 4 se muestra la relación de los componentes de un plan de verificación.

2.2.4 Pre Simulación

La pre simulación se ha convertido en un método eficiente utilizado con el fin de realizar un análisis rápido del diseño, es decir, aun teniendo una especificación detallada del diseño, se puede recurrir a un pre-simulado para conocer el comportamiento "general" del diseño para poder crear así un plan de verificación, asegurando con esto un producto confiable.

Como se puede apreciar en [8] tomando en cuenta ciertos criterios, se puede considerar la pre-simulación como una herramienta eficiente para predecir y disminuir la carga de trabajo inherente en la simulación formal al obtener un análisis rápido del diseño con el fin de enfocar el plan de evaluación y no desperdiciar esfuerzo.

2.3 Lenguajes de descripción de Hardware

Los lenguajes de descripción de hardware o *HDL's* (*hardware Description languages*) fueron desarrollados para hacer frente a la creciente complejidad de los diseños, han sido

usados para modelar la arquitectura y comportamiento de sistemas electrónicos discretos en el proceso de diseño, en la actualidad existen herramientas software que permiten pasar de dichas descripciones a diseños a nivel de compuertas, a éste método de transición se le llama "síntesis".

Existen 3 maneras básicas de describir un diseño:

- Por comportamiento (*behavioral*).
- Por flujo de datos (*data-flow*).
- Estructural (*estructural*).

2.3.1 Ventajas

- La propia descripción en el lenguaje de alto nivel sirve como especificación del comportamiento del diseño.
- Los modelos descritos con estos lenguajes, pueden ser verificados fácilmente y de forma precisa por simuladores definidos en base a éstos HDL's.
- Cumple con el requerimiento más importante de toda especificación, no es ambiguo.
- Los TestBench pueden ser escritos en el mismo lenguaje con el que han sido modelados los diseños.
- El lenguaje soporta jerarquía, es decir, un sistema puede ser modelado como un conjunto de componentes interconectados.
- Es un estándar ANSI e IEEE por lo que los modelos descritos en estos lenguajes son totalmente portables.
- No hay limitaciones impuestas por el lenguaje para el tamaño del diseño.

2.3.2 Desventajas

- Supone una curva de aprendizaje ya que prácticamente se puede considerar como una nueva metodología.
- Se requiere la adquisición de nuevas herramientas, como simuladores y sintetizadores de HDL,
- El uso de estos lenguajes no controla el aspecto físico del diseño, dándole una mayor importancia a la funcionalidad, lo que puede complicarse si el espacio destinado al diseño es pequeño.

2.4 Particionamiento de Grafos

El particionamiento de grafos es un problema muy estudiado en el campo de la computación científica combinatoria. Su aplicación más importante es el mapeo de información y/o procesos en computación paralela (sistemas distribuidos), donde los objetivos son la distribución uniforme del trabajo y el mínimo de comunicaciones. [9]

Existen muchas variantes del particionamiento de grafos, pero todas siguen siendo problemas NP-hard/completos aun cuando existen adecuaciones heurísticas. El número de investigadores y trabajos dedicados a éste problema comenzó a crecer con la evolución de los sistemas digitales a finales de los años 60's y es una técnica utilizada para dividir representaciones gráficas de circuitos o sistemas en una colección de partes más pequeñas a las cuales se les llama componentes [2].

Con el paso del tiempo, y debido a la naturaleza de los diseños actuales, el particionamiento de grafos se volvió un enfoque recurrente en los algoritmos propuestos para mejorar el proceso de diseño de sistemas digitales debido a que un grafo dirigido es la manera más eficiente de homogeneizar la representación de cualquier circuito, independientemente del lenguaje o el enfoque con que se haya descrito el circuito en cuestión.

El presente trabajo puede brindar una idea clara de la cantidad de trabajo invertido a lo largo de los años en éste problema en particular, mostrando así un panorama de la magnitud del problema, sus alcances y el trabajo que queda por desarrollar al respecto.

A pesar de haber tantos enfoques para éste proceso, tomando en cuenta la información recabada en el proceso de investigación para este documento, el enfoque de utilizar una representación gráfica a manera de aislar la mayoría de diseños en los mismos terminos parece ser el más eficiente y prometedor. Al haber muchas metodologías de descripción y tan pocas limitantes sobre ello, trabajar en cuanto al grafo que describe el comportamiento del diseño en cuestión hace el análisis totalmente homogéneo y por tanto, implementar un algoritmo de particionamiento podría resultar aplicable para casi cualquier diseño.

Existen muchos métodos de particionamiento como los citados en [10] [11] [12], estos realizados con base en diferentes conceptos e ideas, también existen tendencias de particionamiento como las que se apoyan de heurística, es decir, que abandonan el concepto de buen tiempo de ejecución o buenas soluciones, y en ocasiones abandonan ambos conceptos, además de que, por la naturaleza del problema, no hay pruebas de que la solución no pueda ser arbitrariamente errónea o que no existe prueba de que siempre se ejecutará razonablemente rápido, como las citadas en [13]; también están mas propuestas con base en grafos, como las que se mencionan en [14] que ocupan alguno de los tantos algoritmos de particionamiento existentes (conos, replicación, en cadena, mínimo *cutzise*, recorrido, etc.) o bien, combinaciones y modificaciones de los mismos.

Cabe mencionar que el particionamiento de grafos puede ser aplicado a cualquier problema de optimización que pueda ser abstraído con un grafo, en la actualidad el ejemplo práctico y más recurrente es el particionamiento de redes de computadoras ya sea para subneteos específicos o simple administración, así como para la optimización de rutas.

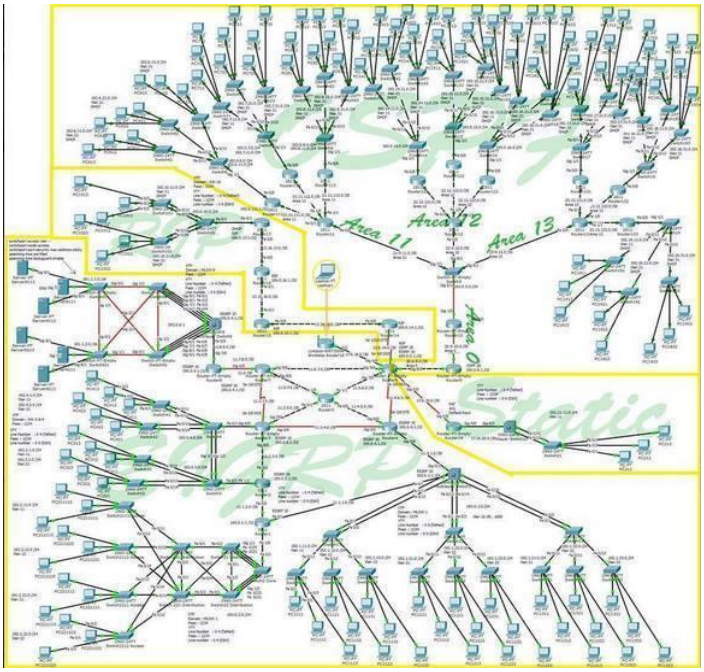

Figura 5 Redes de Computadoras.

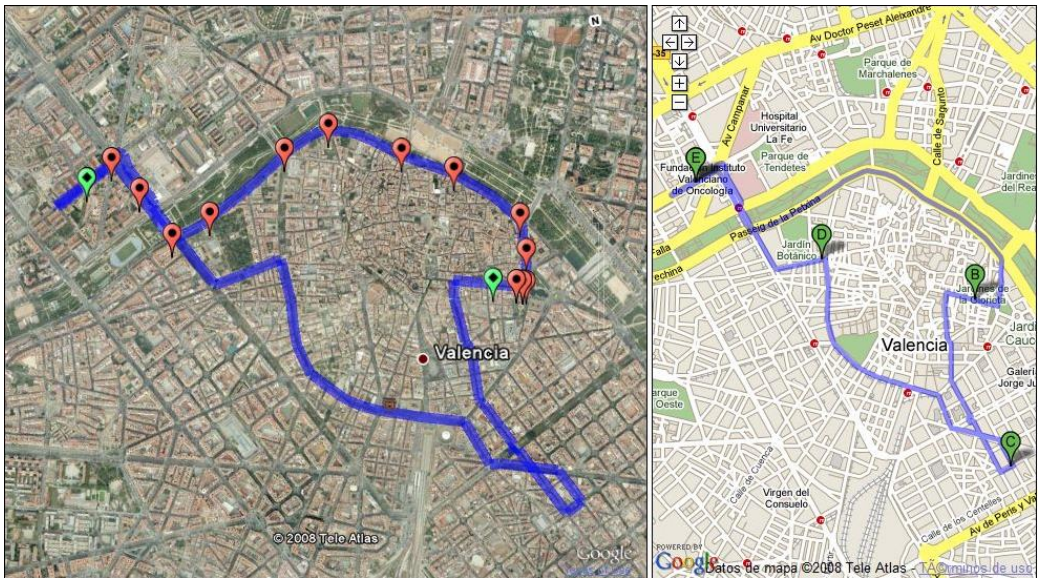

Figura 6 Optimización de rutas.

A continuación se mencionan algunos de los trabajos reportados en el estado del arte que utilizan alguna de las técnicas ya mencionadas de particionamiento de grafos.

2.4.1 Metodología de monitoreo para validación de circuitos VLSI

Este es un trabajo de tesis desarrollada en el CIC (Centro de Investigación en Computación) [4] en el que se implementó una herramienta de naturaleza similar, es decir, se propone desarrollar una herramienta capaz de recibir como entrada un diseño implementado en SystemC y mediante pre simulación identificar el comportamiento del diseño haciendo uso de técnicas rudimentarias para generar así su grafo característico y proceder al particionamiento del mismo.

La principal limitante de ésta herramienta es que no fue desarrollada completamente y la investigación realizada quedó inconclusa, los datos obtenidos de la pre simulación son subjetivos puesto que el comportamiento fue extraído con base en un criterio personal y no general, que es lo que una herramienta de este tipo necesita. La herramienta solo fue capaz de particionar un sumador completo de 4 bits, obteniendo como componentes resultantes 4 sumadores completos de un bit (los componentes iniciales del diseño), haciendo uso de métricas no oficiales ni generales, lo que le impide ser una herramienta útil para el efecto de la verificación.

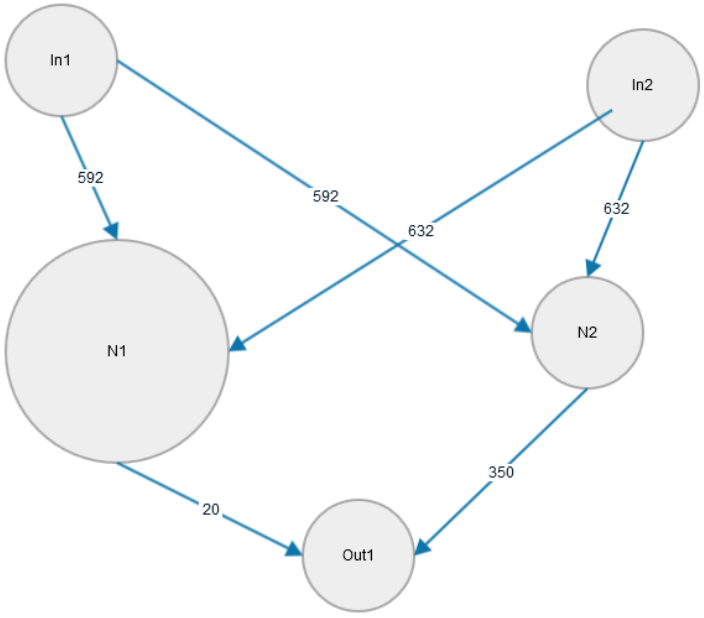

Figura 7 Grafo resultante.

Independientemente del criterio utilizado para conocer el comportamiento del circuito, el hecho de no ser general implica la implementación manual de los métodos para dicha verificación, esto implicaría conocer el diseño de alguna manera, por lo que la pre-

simulación podría parecer innecesario, además de que para un diseño de gran escala realizar ésta tarea podría implicar gastos aún mayores.

2.4.2 Parallel Logic Simulation of VLSI Systems

En este trabajo [5] se aborda la importancia de la simulación en el proceso de diseño de sistemas *VLSI*, los problemas que representa y las preocupaciones actuales al respecto. También describe lo que podría considerarse un parteaguas en lo que se refiere a los enfoques utilizados para atacar este problema.

El enfoque de este *paper* no es centrarse en las pruebas ni en el cómo se realizan si no realizarlas de una manera eficiente y a menor costo tiempo/procesamiento, es decir, utilizar la simulación en un entorno distribuido (simulación paralela) de manera que, al distribuir el trabajo se disminuyan costos de procesamiento y tiempo de simulación/pruebas. Éste enfoque ha tenido una gran aceptación dentro de la comunidad puesto que promete reducir costos y tiempos, además, es un enfoque nuevo y poco trabajado, por lo que, aunque no se pueden prometer soluciones concretas aún, en la teoría demuestra tener potencial para convertirse en uno de las principales herramientas para la reducción de esfuerzo en la etapa de simulación del proceso de diseño.

Para manejar dicho enfoque se deben tener en cuenta 5 puntos importantes:

1. **El algoritmo de sincronización:** es el que se encarga de coordinar la interacción entre los procesos dentro y fuera de cada integrante del *cluster*.

2. **Estructura del circuito:** comprende los componentes y comportamiento del circuito, así como sus respectivos vectores de prueba.

3. **Granularidad temporal:** Se refiere al tamaño de los intervalos de tiempo que el simulador tarda en tomar las muestras.

4. **Arquitectura objetivo:** La simulación paralela depende también del tipo de arquitectura que se va a simular, ya que existen diversas formas de particionar una misma arquitectura.

5. **Particionamiento y mapeo:** La forma en la que se divide la arquitectura en módulos más pequeños y son asignados a los diferentes procesadores.

Este trabajo es considerado de gran importancia ya que plantea que el particionamiento y mapeo del diseño es la parte de mayor relevancia en una simulación distribuida.

Otros aportes importantes del trabajo en cuestión son las comparaciones y datos relevantes sobre los algoritmos de sincronización, técnicas de particionamiento y mapeo, así como algunas implementaciones además de pruebas con resultados que permiten ver la

importancia de éstos últimos y que, la consideración de estos 5 puntos resulta ser efectiva para tratar el problema.

Si bien es información que ha sido muy ilustrativa en el proceso de investigación, sus aportes no van más allá de dar una idea de las consideraciones que se deben tomar, los problemas que podrían implicar y el trabajo que queda por realizar. Además de no ser una solución tangible que ayude a reducir el problema del cuello de botella en el proceso de diseño de sistemas digitales.

2.4.3 Transistor Level Simulation

En este trabajo [1] de investigación en el que se describe un enfoque que toma como premisa que lo importante, más que la distribución uniforme de trabajo es la reducción de conexiones entre las particiones, de tal forma que describe como fusionar, multiplicar y separar las particiones de tal manera que se cumpla dicha premisa, además describe algunos conceptos útiles para le representación y minimización de dichas señales tomando en cuenta ciertos criterios y su descripción matemática.

Otro de los puntos importantes de éste documento es que hace referencia a las herramientas TITAN, TIPART y COPART que están basadas en éste enfoque y éstas premisas para el particionamiento y simulación de los diseños de manera distribuida, por obvias razones, tomando en cuenta que son herramientas de uso privado por ciertas empresas no se dan detalles sobre la estructura/lógica de funcionamiento y solo se mencionan ciertos métodos, criterios y algoritmos utilizados bajo del mismo enfoque. Esto demuestra que, este tipo de herramientas son escasas o nulas a nivel comercial y dado que no todas las empresas cuentan con equipos de desarrollo para éstas etapas se requiere una herramienta que solucione éste problema.

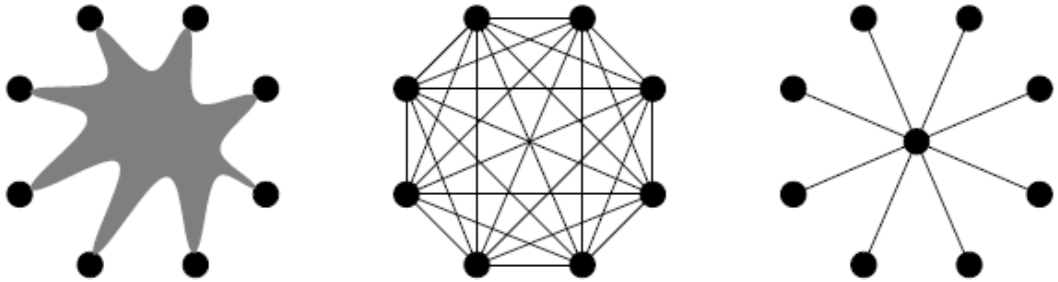

Figura 8 Representación de Señales.

La figura 6 demuestra la diversidad de representaciones y complejidades que puede haber sobre las conexiones e interacción entre los componentes de un diseño, lo que demuestra la importancia de encontrar una forma de homogeneizar la abstracción de los diseños.

La siguiente figura muestra uno de las maneras que existen para tratar con éste tipo de problemas que es la combinación de nodos, tomando en cuenta ciertos criterios, pero eso altera la interpretación del diseño y su comportamiento.

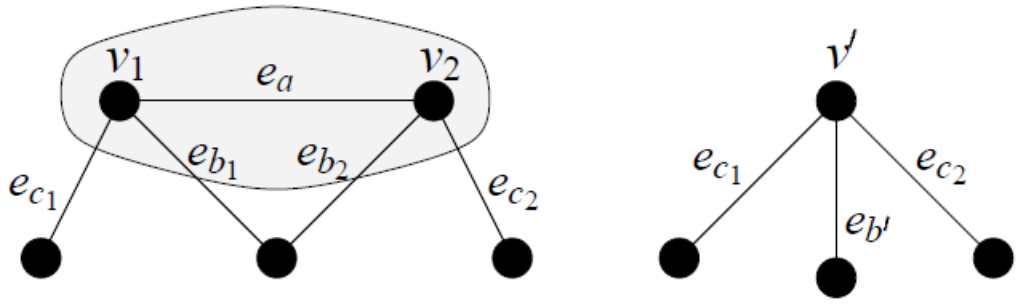

Figura 9 Mezclado de nodos.

Si bien existen otras maneras de reducir comunicaciones como la replicación (proceso inverso) cualquiera de éstas compromete la carga de trabajo y por ello el tiempo de procesamiento.

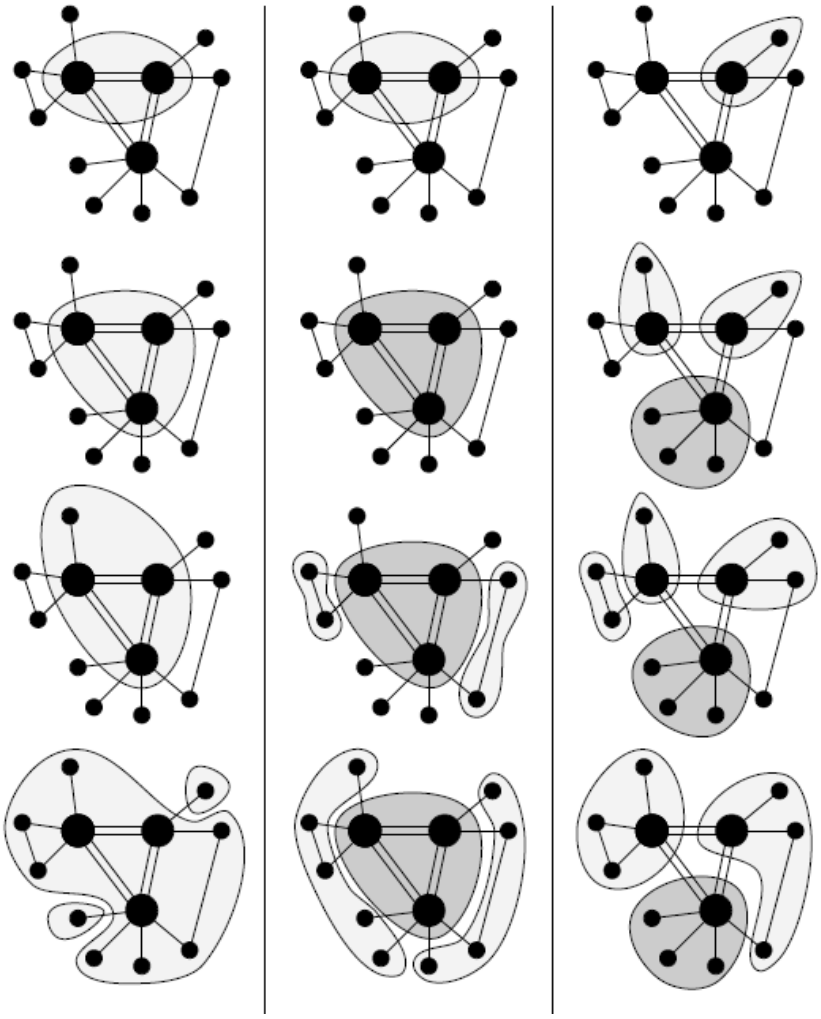

Figura 10 Agrupamiento aleatorio.

En la figura 7 se puede apreciar otra técnica muy recurrente en cuanto al enfoque en cuestión se refiere (preocuparse principalmente por el menor número de comunicaciones entre particiones), consiste en crear agrupaciones aleatorias y enlistarlas de tal manera que al final se elija la que exija menos comunicaciones. Al igual que los algoritmos anteriores, requiere tiempo y análisis realizar las comprobaciones, conteos y combinaciones. A pesar de que no es el mejor enfoque tomando en cuenta las dimensiones de los diseños actuales, es un gran referente sobre el trabajo realizado en el campo y la importancia del tema.

Dado que toma como principio uno de los criterios que nos interesan, se usará fuertemente como referencia para la creación del algoritmo de particionamiento que usará la herramienta.

2.4.4 Circuit Partitioning into Small Sets

Este trabajo [13] describe modelos matemáticos para el particionamiento de diseños *VLSI*, ventajas y desventajas de cada uno de ellos y conceptos que se deben tener en cuenta para la implementación de éste tipo de enfoques.

El principal aporte de éste documento son los múltiples enfoques de particionamiento de manera matemática, de tal forma que pueden ser aplicables (teóricamente) a cualquier tipo de diseño e incluso implementados en casi cualquier lenguaje de programación ya que todos están basados en heurísticas y el análisis comportamental del diseño.

La base o referente de los algoritmos descritos en éste documento es el algoritmo *k-sp*que de manera general consiste en un método heurístico para encontrar de manera rápida la mejor solución posible en cuanto al particionamiento de un circuito dado C=(V,E) con V elementos y E conexiones en grupos de elementos de tamaño fijo *k* para así dar paso a otros algoritmos que contemplan factores distintos para su implementación. Después se describe "*Thebottom-up (BU)heuristic*" y sus respectivas modalidades (*BU-Simple*, *BU-Matching y BU-IterativeImprovement*), sus cualidades y una breve descripción acerca del funcionamiento del algoritmo y sus referentes matemáticos para pasar a la comparación combinacional de los resultados de éstos algoritmos.

TD heuristic algorithm												
	2-SP				3-SP				4-SP			
	OPT	TD	dOPT	TIME	OPT	TD	oOPT	TIME	OPT	TD	dOPT	TIME
AT11	105	126	20%	3.01	76	96	29%	3.01	51	74	45%	2.53
AT12	29	145	63%	4.11	55	87	58%	4.00	53	49	49%	4.00
AT13	201	163	31%	22.04	183	239	31%	21.20	152	189	24%	20.00
T11	29	46	57%	2.00	20	29	45%	7.55	15	23	53%	5.28
T12	50	69	58%	31.30	43	58	35%	30.08	38	53	39%	28.22
T13	169	244	44%	70.53	115	169	47%	69.39	98	138	41%	66.45
T14	147	199	35%	12.1	121	165	36%	125.4	105	142	35%	33.50
Optimal Solution												
AT11	39	56	44%	2.51	36	56	56%	2.51	32	56	75%	2.40
AT12	44	63	43%	4.00	41	59	44%	3.21	39	56	44%	2.11
AT13	121	1.3	26%	19.11	99	123	24%	18.32	93	123	32%	18.32
T11	12	19	58%	5.44	11	19	73%	5.00	11	18	69%	5.00
T12	33	53	60%	28.22	30	52	73%	28.22	28	48	71%	20.56
T13	91	134	47%	66.00	85	126	48%	59.19	82	101	23%	49.19
T14	90	131	46%	100.5	77	92	19%	86.56	62	67	8%	56.00

Tabla 1 Comparación entre TDheuristic y Optimal Solution.

La tabla 1 muestra el tipo de comparaciones que se hacen entre los diferentes algoritmos, como ya se mencionó, el documento contiene las combinaciones de comparaciones entre cada modalidad del algoritmo, para grupos de elementos de tamaño *k* (5,6 y 7), las

soluciones base (soluciones entrada al algoritmo en cuestión) y los tipos de circuitos (RT11, RT12, etc.) que son circuitos utilizados de manera experimental que contienen un máximo de 300 elementos y 500 conexiones. Esta tabla muestra otros valores a comparar como:

- OPT número de conexiones en la partición óptima.
- TD número de conexiones en la partición mediante TDheuristic.
- dOPT la desviación de TD desde OPT ((TD-OPT)100)/OPT
- TIME el tiempo de realización de cada algoritmo en segundos.

Al ser aproximaciones heurísticas no nos aseguran que sea siempre la mejor solución, es decir, siguen existiendo factores que impiden volverlo un método confiable para la solución del problema al que estamos atacando.

2.4.5 Finding Best Cones from Random Clusters for FPGA Package Partitioning

Este trabajo describe el enfoque, que mejores resultados ha mostrado para el paticionamiento de grafos [15], obteniendo así una solución satisfactoria a los 2 aspectos que buscamos cubrir si lo combinamos con el algoritmo de agrupamiento aleatorio, es decir, la distribución uniforme del trabajo y el menor número de comunicaciones entre los componentes.

Debido a que será el algoritmo que tomaremos como base para la implementación de nuestra herramienta, describiremos el algoritmo de forma detallada. El algoritmo que nos interesa toma como principios el MBCM (*Modified best cone Algorithm*) y RC (*RandomClustering*) con algunas variaciones cada uno, por lo que describiremos cuales son éstos algoritmos como una introducción a el algoritmo principal.

MBCM (Modified best Cone Merge Algorithm)

Los conos son estructuras de mínimo corte ya que delimitan las regiones de bajo *fan-out*. Un fan-out alto puede resultar en la superposición de regiones entre conos.

1 Se crea una lista de los nodos raíz de cada cono (nodos entrada) de todos los nodos salida primarios a la cual llamaremos *CRoot*. Ésta lista tendrá como tamaño el número de conos en el circuito. Nótese que nunca puede haber más conos que elementos primarios secuenciales de entrada o salida, pero puede haber más elementos primarios secuenciales de entrada o salida que conos.

2 Escanear los nodos combinatorios de cada nodo en CRooten busca de entradas primarias o elementos secuenciales. Durante cada escaneo, agregar a cada nodo combinatorio una

etiqueta correspondiente al número de entrada del cono razón en la lista. Cada nodo combinatorio tiene un *label-list* (LL) de la manera:

$$LL(Nd) = \bigcup_{i=1}^{\#P(Nd)} LL(P(Nd)i)$$

Donde *#P(Nd)* es el número de padres *P(Nd)i* de nodos combinatorios *Nd*. Si no existe un padre con la misma etiqueta en la lista entonces se añade a la misma ya que representa una raíz de grupo con *fan-out* entre conos. Nótese que esos CRoot entradas tienen una etiqueta más grande que 1. El nuevo tamaño de la lista es el número de grupos en el circuito.

3Superponer secciones de los conos mediante agrupación de nodos combinatorios con etiquetas comunes en la lista. Cada grupo puesto tiene nodos únicos combinatorios definidos por:

$$Cluster_i = \{Nd | (\exists P(Nd)_i [LL(Nd) = LL(P(Nd)_i)]) \hat{} (LL(Nd) = LL(CRoot_i))\}$$

Donde*CRooti* es el nodo raíz del grupo *Clusteri* y *P(Nd)=CRooti* es posible solo si la siguiente condición se cumple:

$$\forall i \neq j (LL(CRoot_i) \cap LL(CRoot_j) = \emptyset)$$

Entonces #Clusters=#Cones sin conos superpuestos. Circuitos que tienen más grupos que conos indican alta superposición de conos y por tanto un alto *fan-out*.

4 Agregar elementos secuenciales a los grupos para reducir el tamaño de entradas-salidas del nodo. Esto significa agregar a los grupos con el número más grande de interconexiones en común.

5 Encontrar caminos combinatorios críticos con el algoritmo [16] y ordenar caminos por la disminución del retardo de trayectoria (i.e. el camino más crítico para ser procesado primero). Combinar grupos que tengan elementos combinatorios en la misma ruta crítica.

6Combinar grupos para encontrar los mejores conos. Esto se hace para encontrar la primera *ClusterRoot,* una lista de conos raíz (grupos) de la lista original *CRoot* creada anteriormente. A continuación, un par de grupos son elegidos para combinar. Cada par de grupos es hecho de un *Clusteri* de la lista *ClusterRoot*añadiendo también un pequeño grupo*Cd* de la lista.

$$ClusterChild(Cluster_i) = \{Cd | Cluster_i = PCluster(Cd) Cd \in ClusterCRoot\}$$

Donde *PCluster(Cd)* es el padre del grupo teniendo entradas de conexión desde el grupo *Cd*. El par de grupos que produce el mayor *costo de entradas-salidas* después de haber hecho la combinación es seleccionado si la combinación resultante cumple las restricciones

del paquete. De esta manera, los conos combinatorios crecen de manera que el mayor *costo de entradas-salidas*de los conos permanece fijo.

7 Combinar grupos (i,j) para llenar los paquetes siempre y cuando cumplan las restricciones del paquete en cuestión:

$$Size_M < MaxSize_{Pkg.},\ \ IO_M < MaxIO_{Pkg.}$$

Y así minimizar el número de paquetes, lo que significar maximizar la relación de costo del cluster resultante M.

$$Cost_M = \left[\frac{Size_M = Size_i + Size_j}{IO_M = IO_i + IO_j - (2 * IO_{common}}\right]$$

Donde el cuello de botella de pines *de entrada-salida* causa un máximo número de restricciones que rara vez se satisfacen. *IOcommon* es el número de conexiones comunes *de entrada salida*entre grupos. Combinar grupos con *CostM=1* produce grupos de *CostM>=1* donde el tamaño crece más que el número de entradas-salidas. El grupo con el mayor costo es combinado con el grupo que produzca un nuevo grupo con gran costo. Las particiones pueden por tanto, ser llenadas secuencialmente, lo que permite elegir entre paquetes. Dado que éste codicioso método de combinación produce primero los grupos con el mayor costo *de entradas-salidas*, el paquete con el mayor *tamaño o el máximo de entradas-salidas del paquete*. Seguramente será llenado primero.

Agrupamiento Aleatorio

El algoritmo *RC (Random clustering Algorithm)* ha demostrado una mejora importante en cuanto a la superposición de grupos debido a su enfoque, el algoritmo consiste en:

1 Asignar cada nodo combinatorio en el circuito como un grupo vértice de un grafo dirigido *G=(V,E)* donde no hay conexiones de grupos o la arista conecta un grupo consigo mismo, pero aquí puede haber múltiples conexiones entre grupos con la misma dirección.

2 Secuencialmente escanear desde cada grupo hacia afuera encontrando el costo resultante de los grupos de grupos. Para esto primero se inicializa la variable *S* con el número de grupos que no han sido escaneados.

3 Recursivamente escanear los *NClusters* previamente especificados a lo profundo desde grupos de conexiones entrada-salida de los grupos que no han sido escaneados. En otras palabras, seleccionar un grupo con alguna conexión entrada-salida hacia *S* desde IO(S) donde:

$$IO(S) = \{v \in V - S | \{s, v\} \in E\ \ \hat{}\ s \in E\}$$

Y agregar el grupo seleccionado a *S*. Esto define M-grupos o grafos completos con M grupos donde M ≤ N. El patrón promedio es:

$$\exists(v \in IO(S) | Cost(S \cup v) \geq Cost(S))$$

Haciendo M igual a N teniendo usualmente el mayor costo pero éste no es siempre el caso. Definir la profundidad del grupo 1<N<15 dependiendo del máximo número de nodos en el circuito, aunque los valores de N mayores a 15 resultan con altos requerimientos de tiempo CPU que son por tanto, desalentadores. EL máximo grado de nodo fue elegido como el valor de N porque esto provoca una cota mayor en M-grupos en cuanto a tamaño que a menudo produce un máximo costo en ciclos. Encontrar el M-grupo que tenga el máximo costo de entrada-salida del circuito antes de combinar.

5 Ir al paso 2 si el escaneo recursivo no se ha realizado en cada grupo. En la práctica con grandes circuitos, el escaneo de todos los grupos es impráctico por los requerimientos de tiempo CPU. Escanear cada grupo también se ha encontrado innecesario.

4 Combinar los M grupos con el mayor costo M-cliques del circuito si el grupo resultante tiene un tamaño menor que el especificado como tamaño límite. El tamaño de estos grupos aleatorios es limitado a partir de que el tamaño crece más que las entradas-salidas. Grandes grupos tenderán a tener gran costo y mezclarse sin control. Esta cota grupal trabaja mejor para *SizeLimit=k*MaxSizePkg*. Donde k es el valor encontrado empíricamente y k=0.1 es usualmente el mejor. De esta manera, las soluciones óptimas locales con el mayor costo de entradas-salidas son encontradas y combinadas. Inicializar todos los grupos no escaneados e ir al paso 2 si los pares de grupos existen con un tamaño de combinación menor que tamaño límite.

3 Agregar elementos secuenciales a los grupos.

4 Combinar pares de grupos que tienen un camino crítico.

5 Combinar pares de grupos para maximizar el costo de entrada-salida y llenar los paquetes.

Encontrando los mejores conos para grupos aleatorios (MBCM-RC)

Como se mencionó anteriormente, el MBCM y RC pueden ser combinados para producir mejores resultados. RC comienza por dar soluciones óptimas localmente de grupos "naturales" que son grupos poco densos. MBCM termina con una solución global que combina lentamente en puntos seleccionados a través del circuito para un mayor costo de entrada-salida.

1 Asignar cada nodo combinatorio en el circuito como un grupo.

2 Definir localmente grupos óptimos aleatorios tanto que RC como las combinaciones produzcan como resultado*Cost>p*SizeM/IOM,* donde p es el valor encontrado de manera

empírica y se tiene que p=0.1 es el mejor valor. Ésta cota baja es necesaria para prevenir malos grupos de bajo costo por mezclar muy pronto, donde combinaciones realizadas no son desechas después para llenar los paquetes.

3 Agregar elementos secuenciales a los grupos.

4 Combinar pares de grupos con base a su contenido de rutas críticas.

5 Combinar pares de grupos como en MBCM. Continuar combinando tanto como las combinaciones produzcan $Cost > q * SizeM / IOM$, donde q=0.1.

6 Combinar pares de grupos para maximizar costo de entrada-salida y llenar paquetes.

Capítulo 3 Análisis y Diseño

3.1 Análisis

3.1.1 Estudio de factibilidad

El estudio de factibilidad es un análisis comprensivo que sirve para recopilar datos relevantes sobre el desarrollo de un proyecto y con base en ello tomar la mejor decisión para establecer si se procede su estudio, desarrollo o implementación, para el efecto del trabajo en cuestión. Los componentes de este estudio profundizan la investigación por medio de los análisis técnico/operativo. [17]

3.1.2 Factibilidad Técnica

Para el análisis de la factibilidad técnica se tomaron como base los requerimientos mínimos solicitados por las herramientas que se usarán en el desarrollo de la plataforma en cuestión y las capacidades del equipo donde se desarrollará el presente trabajo.

Las herramientas a utilizar son:

Quartus II 11.0 Web Edition

Es una herramienta de software producida por Altera para el análisis y síntesis de diseños realizados en algunos HDL como VHDL, Verilog y System Verilog entre otros. La versión web es una edición gratuita que puede ser descargada del sitio oficial o solicitada por correo, su inconveniente es que el número de dispositivos que se pueden compilar y programar son limitados, como FPGA's, que manejan una escala menor.

Requerimiento	Especificación
Memoria	20 GB disponibles (mínimo)
SO	Windows 7 (x32 & x64) Windows XP SP2(x32 & x64) Windows SP Pro (x64) Windows Server 2008 R2E(x64) Red Hat EL6(x64) Red Hat EL5(x32 & x64)
DSP Builder	MathWorks R2010b MathWorks R2011b MathWorks R2012a MathWorksR2012b

Tabla 2 Requerimientos Quartus II.

ModelSim

Es un entorno de depuración para diseños descritos en Verilog, VHDL y SystemC creado por *MentorGrpaphics®* basado en SKS, lo que la hace una potente herramienta para la verificación y manipulación de éste tipo de productos. Los recursos necesarios para ésta herramienta vienen contemplados en la tabla anterior ya que la versión Web "encapsula" ambos productos para brindar a los usuarios primerizos un entorno completo y funcional para desenvolverse en el ámbito del diseño de hardware.

Grafos

Es una herramienta gratuita desarrollada por Alejandro Rodríguez Villalobos http://personales.upv.es/arodrigu/ que pretende dibujar, modelar, resolver y analizar cualquier tipo de grafo o autómata [18]. Grafos resulta ser probablemente la única herramienta de licencia libre que permite la entrada de archivos graphML de manera fácil y rápida, además de su accesibilidad y la facilidad de manipulación de entradas/salidas de documentos. Si bien es una herramienta que aún está en desarrollo, la versión estable hasta ahora permite acceder a las operaciones que se necesitan para el efecto de mostrar de manera visual el grafo en cuestión.

Requerimiento	Especificación
SO	Windows XP Windows Vista Windows 7
Memoria RAM	Determinados por el SO
Disco Duro	5GB libres
Periféricos	Impresoras y tarjeta gráfica (solo si se requieren)
Conexón a internet	Actualización periódica y manuales de usuario.

Tabla 3 Requerimientos Grafos.

Visual Studio 2013

Es un entorno (IDE, por sus siglas en inglés) para sistemas operativos Windows. Soporta múltiples lenguajes de programación tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby,PHP; al igual que entornos de desarrollo web como ASP.NET MVC, Django, etc., a lo cual sumarle las nuevas capacidades online bajo Windows Azure en forma del editor Monaco.

Visual Studio permite a los desarrolladores crear aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET (a partir de la

versión .NET 2002). Así se pueden crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, dispositivos embebidos, consolas, etc.

Entre otras razones, dado que el lenguaje C# resulta ser bastante potente y amigable en ésta plataforma, debido a que se tiene cierta familididad con el mismo se ha elegido como base para el desarrollo del presente trabajo, además de que, ya que la plataforma permite recabar estadísticas con respecto al costo computacional y el tiempo de ejecución de cada proceso servirá para dejar un estimado de cada aspecto del proyecto en caso de que se desee continuar el proyecto después.

Requerimiento	Especificación
Procesador	1GHz o superior
RAM	1GB
Disco Duro	20 GB
Monitor	DirectX 9

Tabla 4 Rquerimientos VisualStudio 2013.

Equipo de trabajo

El equipo o estación de trabajo donde se desarrollará el presente trabajo terminal es una PC All-in-One que cuenta con las siguientes características:

Requerimiento	Especificación
SO	Windows 7 Professional
RAM	4GB (2.92 utilizable)
Disco Duro	1 TB
Procesador	3° generación Intel Core 3.30 GHz
Monitor	21,5" formato de imagen 16:9 FHD multitouch
Gráficos	AMD Radeon HD 1GB

Tabla 5 Equipo de Trabajo.

3.1.3 Determinación del análisis

Como resultado de éste estudio se determinó que en las condiciones actuales, se cuenta con la infraestructura necesaria para el desarrollo y puesta en funcionamiento del sistema propuesto ya que como mínimo, se cumplen con las especificaciones solicitadas por las herramientas a utilizar.

Tomando en cuenta que el propósito del presente trabajo es probar la funcionalidad y viabilidad del algoritmo de particionamiento propuesto, no se realiza ningún análisis de factibilidad económica, puesto que el producto no puede ser comercializado hasta no comprobar que el algoritmo es competente y que, el enfoque de trabajo de la herramienta

puede resultar útil para el efecto de la fase de verificación del proceso de diseño de sistemas digitales.

3.2 Diseño

3.2.1 Diagrama de Proceso

La herramienta a desarrollar con el fin de comprobar los resultados del enfoque propuesto se compone de 2 etapas principales con diferentes procesos cada una, los cuales serán descritos a continuación, la herramienta en general se describe mediante el siguiente esquema:

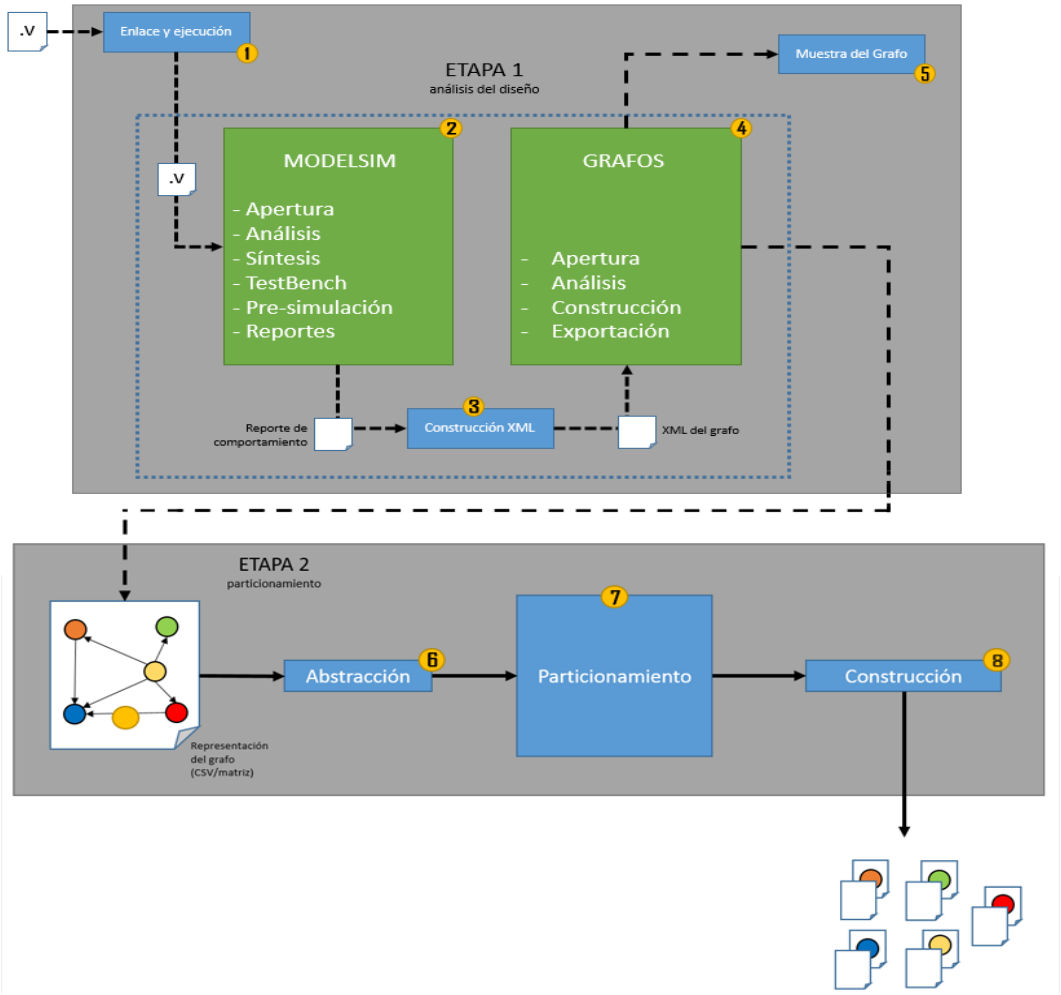

Figura 11 Diagrama de Proceso.

3.2.2 Etapas del proceso

La etapa 1 de la herramienta a desarrollar se compone de las siguientes tareas:

1º ENLACE Y EJECUCION: Se plante la parte de la herramienta que le permitirá al usuario indicar el archivo, una vez realizado esto, la herramienta se encargará, mediante un script de pasar dicho diseño a la herramienta ModelSim para pasar así a la siguiente tarea.

2º ModelSim: Se trata de una herramienta de simulación muy potente que para nuestros objetivos se encargará de abrir, sintetizar y realizar una pre-simulación del diseño a fin de entregar un reporte que describa el comportamiento general del circuito en cuestión (componentes, carga de trabajo y comunicación entre dichos componentes) para poder así, proceder a la siguiente tarea.

3º CONSTRUCCION XML: Se encargará de leer los reportes obtenidos y condensarlos en un XML a fin de que se pueda proceder a la tarea 4.

4º GRAFOS: Es una herramienta *opensource* diseñada para la creación, análisis y manipulación de grafos, ésta herramienta también soporta entrada XML lo que nos permite crear una versión visual del grafo característico del circuito en cuestión y poder así pasar a la siguiente tarea.

5º MUESTRA: Esta fase permitirá mostrar el grafo característico que describe el comportamiento del circuito en cuestión al usuario, si bien no es indispensable que el usuario tenga noción del grafo mencionado, se pretende de esta manera para que en un trabajo futuro el usuario pueda modificar a consideración la estructura del grafo y con ello el esquema de particionamiento.

La segunda etapa de la herramienta consta de la ejecución del algoritmo de particionamiento y sus respectivas implicaciones, esta etapa contempla de 3 tareas:

6º ABSTRACION es un pequeño modulo que se encargará de “abstraer” o bajar a una estructura de datos el grafo obtenido, a fin de poder ser manejado por un lenguaje y tratado en la siguiente tarea.

7º PARTICIONAMIENTO se encargará de realizar la tarea del particionamiento del grafo en cuestión con sus respectivas consideraciones, tomando como consideración principal obtener una carga uniforme de trabajo y el menor número de comunicaciones entre las entidades obtenidas.

8º CONSTRUCCION es el modulo final del proceso descrito, se encargara de generar el testBench de cada entidad/modulo resultante de la partición con el fin de entregar al usuario el diseño listo para la simulación en una estación distribuida.

3.2.3 Diseño de la interfaz

A continuación se muestra y describe la interfaz diseñada para la herramienta desarrollada en el presente documento:

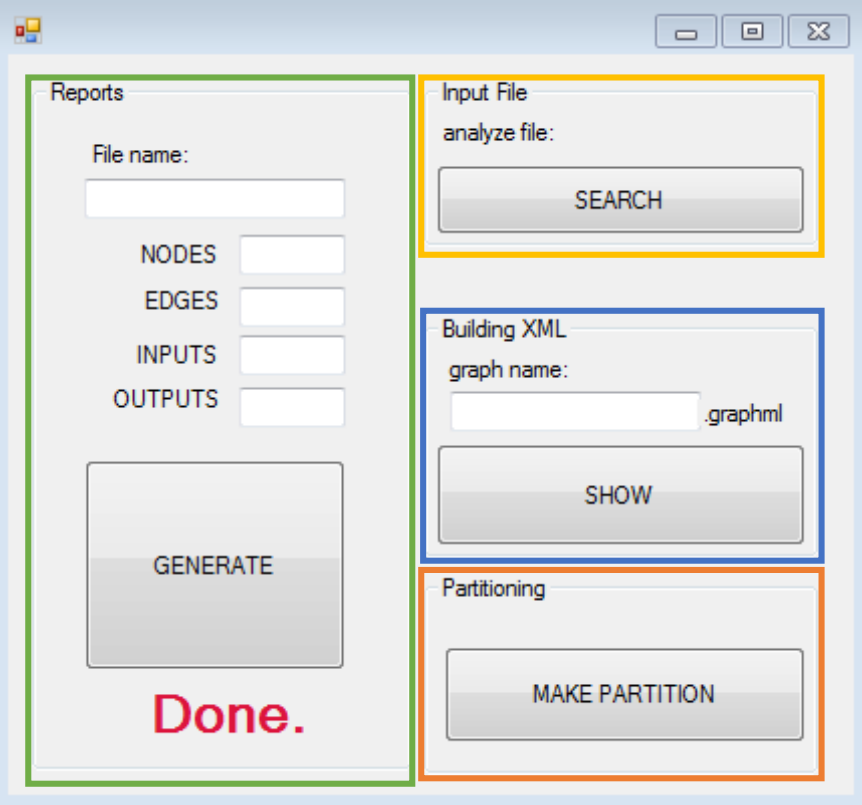

Figura 12 Interfaz de la Herramienta.

Generación de Reportes

El *módulo* "Reports" (**recuadro verde**) mostrado en la figura 10, es un módulo que se ha añadido a las especificaciones originales de la herramienta. Su implementación se debe a que, en la actualidad, no existe forma de automatizar la extracción de la información con respecto al comportamiento general del diseño en cuestión. Si bien la cobertura funcional promete ser un aproximado para tal efecto, es necesario que un analista se encargue de estudiar el diseño en cuestión para construir el TestBench específico. Es necesario mencionar que el reporte final e incluso la base de datos tienen una estructura específica, por lo que extraer la información útil de éstos implicaría un análisis extensivo que independientemente de su complejidad resulta ineficiente por lo mencionado en un principio. Dicha estructura puede ser verificada en [19].

Por todo esto, el módulo añadido tiene como propósito generar grafos de manera aleatoria que cumplan con los requerimientos o limitaciones propias de un diseño que se han identificado, tales como:

- Debe haber al menos un nodo de entrada para cada diseño.
- Debe haber al menos un nodo de salida para cada diseño.
- Los nodos entrada no pueden estar relacionados directamente con los nodos de salida.
- Cada nodo debe pertenecer a un nivel de profundidad (considerando 5 como máximo).
- Para cada nodo que no es de entrada ni de salida, debe haber al menos un arco de entrada y uno de salida. Esto debido a que un proceso que no manipula información y no recibe ni entrega ningún tipo de dato es un proceso que solo consume recursos, en el peor de los casos.

Para dar paso al algoritmo de generación aleatoria, se requiere que el diseñador especifique cierta información (figura 11 a) para poder cumplir con las especificaciones previamente mencionadas:

- File name: el nombre del grafo a generar, sin extensión.
- Nodes: la cantidad de nodos totales en el grafo.
- Edges: la cantidad de arcos totales en el grafo.
- Inputs: la cantidad de nodos entrada (se restan del total de nodos).
- Outputs: la cantidad de nodos de salida (se restan del total de nodos).

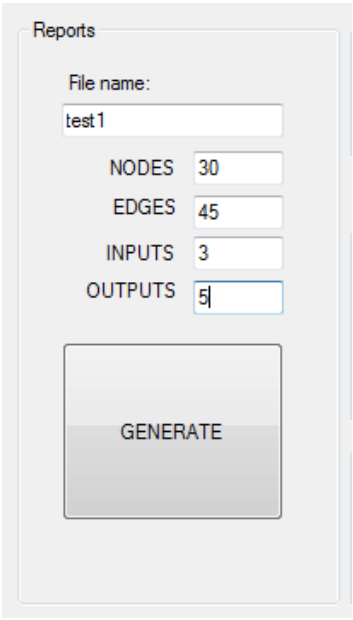

Figura 13 Generador de Grafos Aleatorios (a).

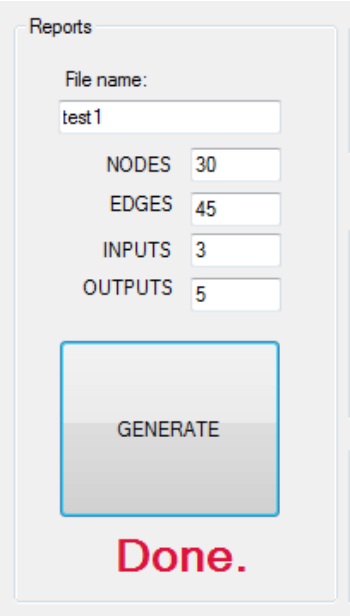

Figura 14 Generador de GrafosAleatorios (b).

Este módulo, una vez terminada su tarea (figura 12 b), entrega archivos de reporte descritos con base en el siguiente esquema:

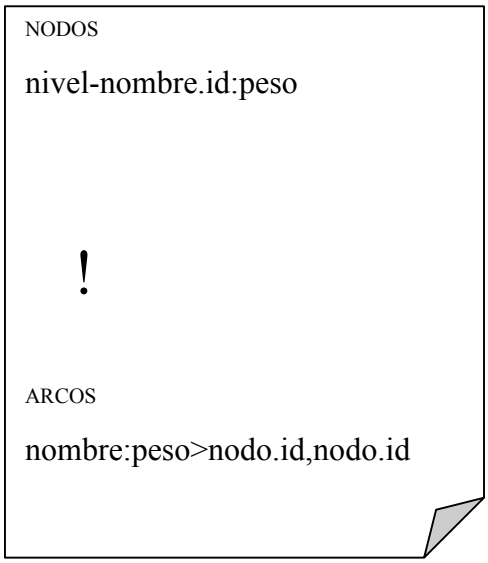

Figura 15Estructura de un Grafo Aleatorio.

De esta forma, la herramienta puede hacer uno de los símbolos especiales insertados entre cada componente para separar y abstraer la información, el final, al nombre del archivo se añade el número de nodos y número de arcos totales para poder trabajar con diseños previamente generados.

Cabe mencionar que, para que el grafo esté correctamente construido con base en el caso de estudio, no pueden existir nodos que no tengan conexión, y en el caso de los nodos de los niveles 2 al 4 deben tener por lo menos un arco de entrada (hacia un nivel inferior) y un arco de salida (hacia un nivel superior), de forma que, para que el algoritmo funcione correctamente el número de arcos especificado debe de ser de al menos el doble de la cantidad de nodos.

Abstracción del diseño

El siguiente módulo, "Input File" (**recuadro amarillo**) figura 10, corresponde a la abstracción del diseño en cuestión. Ésta es la parte en la que la herramienta se encarga de recibir un diseño como entrada (por el momento solo reportes bajo el esquema especificado anteriormente) para abstraer el contenido (nodos y arcos) en objetos de forma tal que se pueda proceder al algoritmo de particionamiento. Esto tomando como base las clases especificadas para tal efecto, las cuales serán descritas más adelante en el presente documento.

Construcción XML

El módulo "Building XML", (**recuadro azul**) figura 10, se encarga de traducir el reporte que se ha seleccionado en un XML, específicamente en el formato graphML, para lo cual escribe la cabecera y los campos descritos previamente en el documento para pasar así a escribir los componentes del grafo.

Debido a que se hace uso de una herramienta de licencia libre y el procesamiento de graphML resulta ser un proceso bastante complejo, se deben especificar los valores default pero no se pueden utilizar, por lo que se ha creado un algoritmo capaz de escribir éstos campos de manera completa para que, al estar completamente traducido bajo el estándar pueda ser visualizado por cualquier otra herramienta. Una vez terminado el proceso de "traducción" la plataforma se encarga de ejecutar la herramienta Grafos para que el usuario pueda abrir el grafo generado y observarlo.

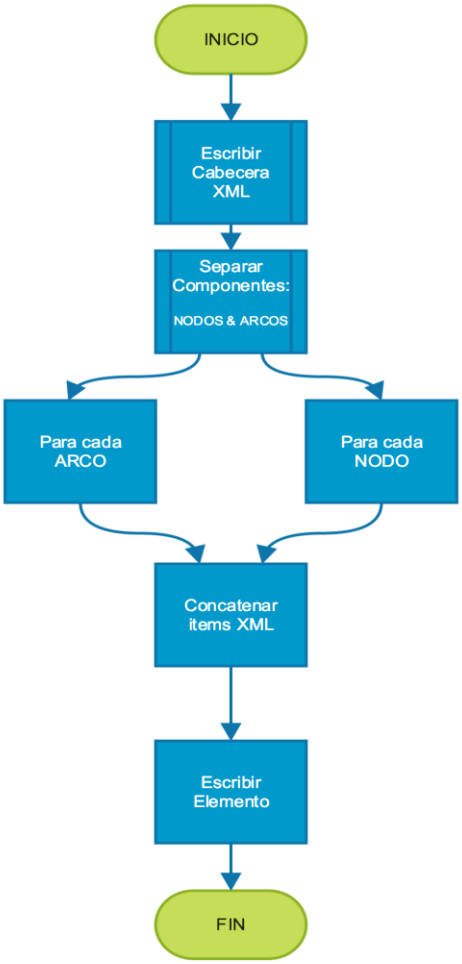

Figura 16 Proceso de Construcción XML.

Como se puede ver en la figura 17, se ha diseñado el módulo de tal manera que el árbol se despliegue de forma horizontal para que no afecte la estructura general conforme crece el árbol del diseño en cuestión y se ha dispuesto de un código de colores para diferenciar cada uno de los niveles del mismo (figura 18).

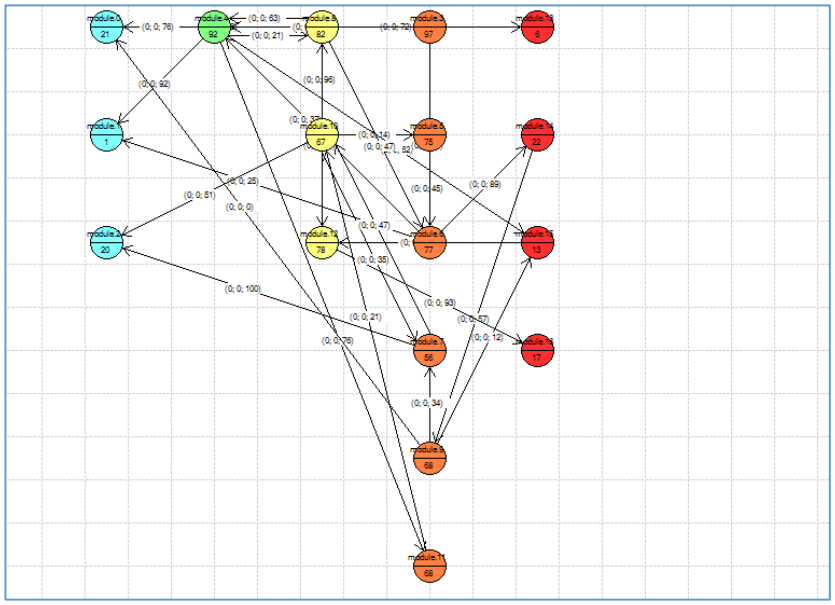

Figura 17 Vista de un Grafo.

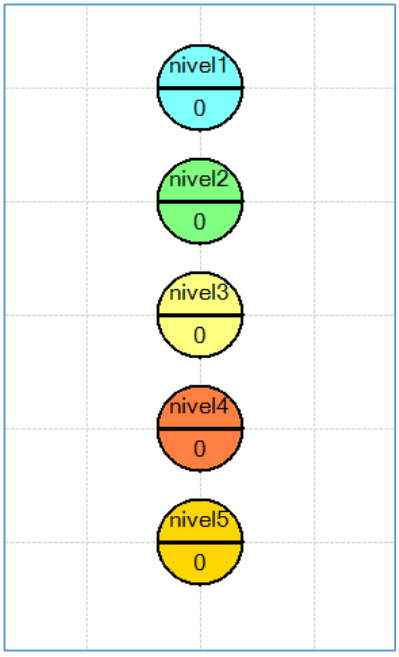

Figura 18 Especificaciones de color por Nivel.

Particionamiento

El modulo "*partitioning*" ,(**recuadro naranja**) figura 10, se encarga de realizar todo proceso pertinente al algoritmo propuesto, la generación de conos y el agrupamiento aleatorio, de forma que, al final se entregue un documento que sirva como esquema para indicar cuál sería la partición ideal del diseño. Cabe mencionar que, por las mismas razones que no se hace la automatización de los reportes, realizar la partición del diseño implica un análisis léxico/sintáctico del diseño de forma tal que se puede automatizar la reescritura del diseño, dicho proceso queda para trabajo a futuro en una posible maestría.

En términos computacionales, el algoritmo de generación de conos propuesto queda de la siguiente manera:

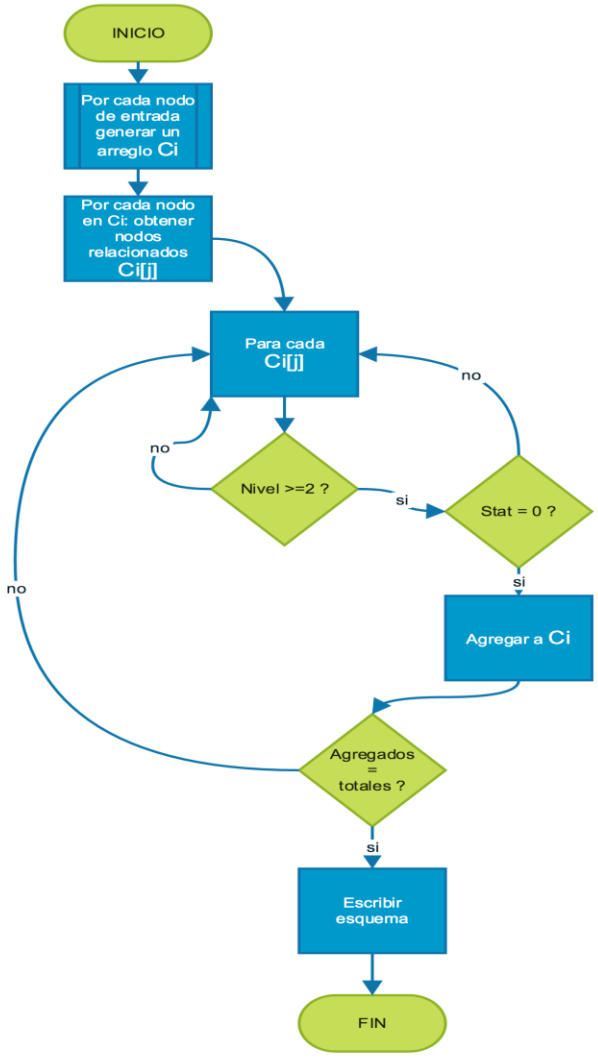

Figura 19 Algoritmo de generación de Conos.

3.3 Casos de uso

Para la herramienta en cuestión es sólo un usuario el que interactúa con el sistema, el diseñador, que al interactuar con ésta generará comportamientos de respuesta en función de la acción que realice, como se puede mostrar en el siguiente diagrama de casos de uso:

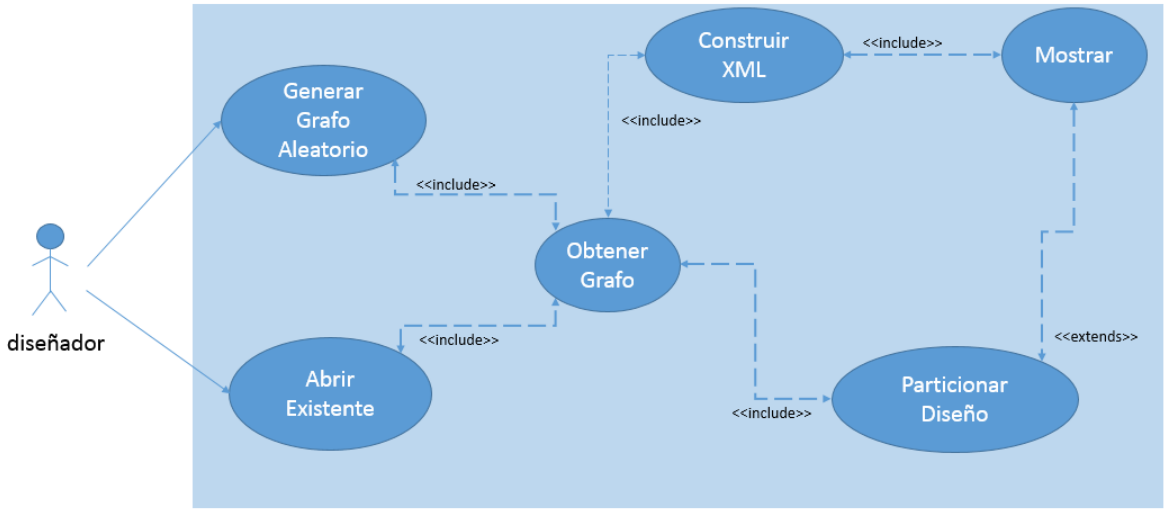

Figura 20 Diagrama de casos de uso

3.3.1 Descripción de los casos de uso

A continuación se describen cada uno de los casos de uso presentados en el diagrama general, figura 17.

ID:	**C1**	
Nombre:	Particionar Grafo.	
Referencias Cruzadas:	N/A.	
Actores:	Diseñador.	
Descripción:	El diseñador desea Particionar un diseño.	
Condición Inicial:	Grafo Seleccionado.	
Condición de salida:	Grafo particionado.	
Flujo de Actividades:	**Actor**	**Sistema**
	1. Presiona el botón "MAKE PARTITION".	2. Verifica la

		existencia de un nodo elegido. 3. Ejecuta el algoritmo. 4. Genera el esquema de particiones.
Condiciones de excepción:	Ninguna.	
Elaborado por:	Cuevas Sánchez Arturo	

Tabla 6 Caso de Uso No.1.

ID:	**C2**	
Nombre:	Generar Grafo Aleatorio.	
Referencias Cruzadas:	N/A.	
Actores:	Diseñador.	
Descripción:	El Diseñador decide generar un grafo aleatorio.	
Condición Inicial:	Especifica los parámetros/características del grafo.	
Condición de salida:	Se genera el archivo *.txt con el grafo característico.	
Flujo de Actividades:	**Actor**	**Sistema**
	1. Especifica los parámetros del grafo. 2. Presiona el botón "GENERATE"	3. Valida los datos de entrada. 4. Ejecuta el algoritmo de generación.
Condiciones de excepción:	Ninguna.	
Elaborado por:	Cuevas Sánchez Arturo	

Tabla 7 Caso de Uso No.2.

ID:	**C3**	
Nombre:	Elegir existente	
Referencias Cruzadas:	N/A.	
Actores:	Diseñador.	
Descripción:	El Diseñador decide cargar un grafo ya generado.	
Condición Inicial:	Archivos generados previamente en el equipo.	
Condición de salida:	Se abre el archivo y se abstrae el contenido.	
Flujo de Actividades:	**Actor**	**Sistema**
	1. Presiona el botón "SEARCH". 3. Elige el grafo.	2. Muestra la ventana de navegación. 4. Abre el archivo y abstrae el contenido.
Condiciones de excepción:	Ninguna.	
Elaborado por:	Cuevas Sánchez Arturo	

Tabla 8 Caso de Uso No.3.

ID:	**C4**	
Nombre:	Mostrar grafo.	
Referencias Cruzadas:	N/A.	
Actores:	Diseñador.	
Descripción:	El diseñador decide ver el formato visual del grafo en cuestión.	
Condición Inicial:	Obtiene un grafo.	
Condición de salida:	Genera el *.graphML y abre la herramienta para su visualización.	
Flujo de Actividades:	**Actor**	**Sistema**
	1. Presiona el botón "SHOW".	2. Verifica que se haya seleccionado un archivo previamente. 4. Ejecuta el algoritmo de construcción. 5. Abre la

	6. Selecciona en el panel de navegación el archivo generado para visualizarlo.	herramienta grafos.
Condiciones de excepción:	Ninguna.	
Elaborado por:	Cuevas Sánchez Arturo	

Tabla 9 Caso de Uso No.4.

3.4 Diagrama de clases

Para la abstracción del grafo en cuestión que de paso al algoritmo de particionamiento se hace uso de 2 clases únicas que contienen la información pertinente al algoritmo (nodos y arcos). Dado que las estructuras "*cluster"* y "*conos*" se pueden manejar como simples arreglos para agrupas contenidos, no se ha especificado una clase para cada uno de ellos, de ésta manera se minimiza el costo computacional, que si bien no es drástico si proporciona facilidades para su manejo por las bondades de la clase en cuestión.

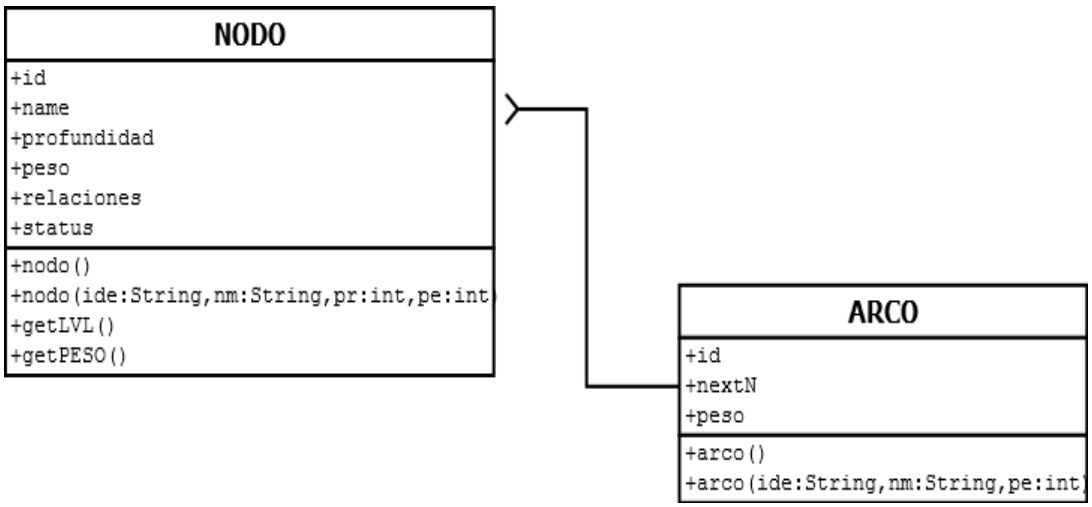

Figura 21 Diagrama de clases.

3.4.1 Descripción de Diagrama de clases

A continuación se describen los campos de cada clase y las significaciones de los mismos:

NODO

Id: es un campo de tipo *String* que identifica al nodo dentro del grafo en cuestión.

Name: es un campo de tipo *String* que ayudará a identificar el nombre del módulo dentro del diseño para posibles consultas que pudieran surgir.

Profundidad: es un campo de tipo *int* que indica en qué nivel del árbol se encuentra el nodo en cuestión, debido a que se desconoce la cantidad de niveles que puede poseer un diseño se ha dispuesto un máximo de 5.

Peso: es un campo de tipo *int* que representa el peso del nodo en cuestión, es decir, especifica la cantidad de trabajo del módulo al que dicho nodo está asociado.

Relaciones: es una estructura de tipo *ArrayList* propia de C# que ayudará a agrupar los arcos adyacentes al nodo en cuestión, de acuerdo a las consideraciones del algoritmo.

Status: es un campo de tipo entero (binario) que servirá como bandera para determinar si el nodo en cuestión ha sido visitado o no en el algoritmo de agrupamiento (conos).

ARCO

Id: identificará al arco en cuestión, es un campo de tipo *String*.

nextN: es un campo de tipo *String* que indicará el otro extremo del arco en cuestión, es decir, a qué nodo se conecta el nodo al que pertenece dicho arco.

Peso: es un campo de tipo entero que representa la latencia del arco, lo que en cuanto al algoritmo, representa el peso de la comunicación entre los nodos adyacentes.

Capítulo 4 Implementación y Pruebas

En esta sección, con base en la figura 11 del diagrama de proceso de la herramienta en cuestión, se describen la implementación y pruebas realizadas en cada una de las etapas y sus respectivas tareas de forma tal que quede constatado el trabajo realizado y el trabajo que queda por realizar en cada una de ellas.

4.1 Implementación y Pruebas de la Etapa 1

La etapa uno del diagrama de proceso de la herramienta descrita en el presente documento comprende las tareas de enlace y ejecución, pre-simulación, extracción de estadísticas y la muestra del grafo característico. Si bien se tienen elegidos los parámetros indicadores del comportamiento del diseño, el esquema, su extracción e interpretación, hacerlo de manera automática no es una tarea factible para un solo trabajo terminal dadas las características de tiempo y forma. Por todo esto, se ha tenido que implementar un módulo que cree algoritmos de manera aleatoria con las especificaciones previamente mencionadas

4.1.1 Generación de Grafos Aleatorios

El algoritmo creado para la generación de grafos aleatorios que cumpla con las restricciones deseadas, de forma tal que se obtenga un grafo lo más parecido a un diseño real trabaja con el esquema mostrado en la siguiente figura:

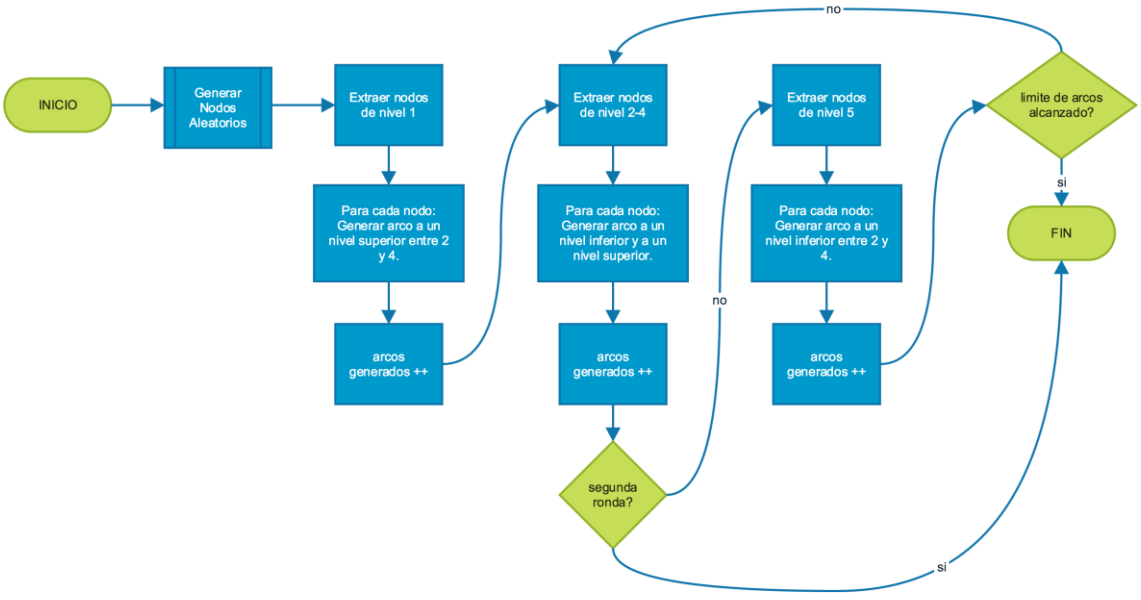

Figura 22 Generación de Grafos Aleatorios.

La primera sección del archivo descrito en el capítulo 3 que describe los grafos generados es generar los nodos, un proceso simple que añade un peso aleatorio entre 1 y 100 tomando como referencia al criterio de verificación de la cobertura funcional que trabaja con respecto al porcentaje del trabajo realizado en un momento dado. La segunda parte consiste en generar los arcos de la siguiente manera:

1. Obtener todos los nodos de nivel 1
2. Conectar todos los nodos de nivel 1 con todos los nodos de nivel 2
3. Conectar todos los nodos de nivel 1 con algún nodo de nivel 2-4.
4. Obtener todos los nodos de niveles entre 2 y 4.
5. Conectar cada uno de los nodos de niveles entre 2 y 4 con un nodo inferior y un nodo superior cada uno.
6. Obtener todos los nodos de nivel 5.
7. Conectar todos los nodos de nivel 5 con todos los nodos de nivel 4.
8. Conectar todos los nodos de nivel 5 con cualquier nodo de niveles entre 2 y 4.

Realizando la generación de arcos aleatorios de ésta manera se aseguran las restricciones especificadas en el capítulo anterior para la generación de grafos.

Para probar el algoritmo de generación de grafos aleatorios se hicieron diferentes corridas con diferentes parámetros, de forma tal que se pueda apreciar la construcción del grafo y el comportamiento de dicho algoritmo. Su comprobación queda implícita en la siguiente sección al mostrar de manera visual el grafo en cuestión.

4.1.2 Construcción XML del grafo característico

Tomando en cuenta el esquema de descripción de grafos del capítulo anterior, se hizo uso del módulo de construcción XML (tarea 3 de la etapa 1, figura 11) para el efecto de apreciar de manera "gráfica" el resultado obtenido del módulo previo. Cabe mencionar que el esquema de éste proceso ha quedado descrito en la figura 16 del capítulo 3, donde se describe el diseño de la interfaz de la herramienta aquí presentada.

Para ello, la herramienta hace uso del estándar GraphML, un formato comprensible y fácil de usar para especificar grafos. Consta de un núcleo de lenguaje capaz de describir las propiedades estructurales de un grafo, y un mecanismo flexible de extensiones para agregar información específica para cada aplicación. Las extensiones pueden ser libremente combinadas o ignoradas sin afectar a los datos del grafo en sí mismo, de forma tal que los grafos generados puedan ser interpretados y dibujados por cualquier herramienta que soporte dicho formato. Por tanto, los componentes del archivo XML de cada grafo son:

La cabecera

```
<?xml versión=”1.0” encoding=”UTF-8”?>
```

La primera línea es una instrucción de proceso que indica que el documento cumple con el estándar XML 1.0 y que la codificación del documento es UTF-9 que es la codificación estándar para documentos XML.

```
<graphmlxmlns=”http://graphdrawing.org/xmlns”♦xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”xsi:schemalocation=”http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd”>
```

La segunda línea indica el esquema, dominio de nombres y el espacio de trabajo, se pueden dejar los valores estándar u omitirlos pero dado que se usan para la validación del documento, es altamente recomendable usarlos. La línea terminaría donde está el símbolo: ♦, en caso de elegir la segunda opción.

El Grafo

```
<graph id=”G” edgedefault=”directed”>
        <node id=”n0”/>
        <node id=”n1”/>
        . . .
        <edge source=”n0” target =”n2”>
        <edge source=”n1” target=”n0”>
</graph>
```

Como se puede ver, la sentencia *graph* se usa para denotar el cuerpo de un nuevo grafo y anidado en éste cuerpo se encuentran las sentencias *node* y *edge* que denotan los nodos y aristas respectivamente.

La herramienta maneja grafos que no son fuertemente tipados, esto quiere decir que puede contener aristas dirigidas y no dirigidas de igual manera.

Usaremos grafos no dirigidos, porque para fines prácticos no nos interesa en qué dirección se realiza la comunicación si no la carga de trabajo o la frecuencia con la que ésta se utiliza, la dirección e implementación de dicha comunicación será tratada por la estación paralela al momento de simular.

Los nodos

Los nodos del grafo son declarados mediante el elemento *node* y cada uno de ellos tiene su propio identificador por lo que no puede haber 2 nodos con el mismo nombre en todo el documento.

Las aristas

Las aristas son declaradas por el elemento *edge*, cada uno contiene la definición de 2 puntos *source* y *target* que son los identificadores de los nodos origen y destino respectivamente. Si se requiere una arista en un *loop* basta con poner el mismo valor en ambos campos.

Opcionalmente se puede declarar la dirección o tipo de arista mediante el atributo *directed*, si el valor es true se maneja como arista dirigida y sin dirección en el caso contrario, si no se especifican ninguno de éstos valores se aplica el tipo definido por default declarado en el cuerpo del grafo.

```
...
<edge id="e1" directed="true" source="n0" target="n2"/>
```

Los atributos

Los atributos del grafo son definidos mediante el elemento *key* que especifica el identificador, el nombre, el tipo y el dominio del atributo. El nombre del atributo es definido por el campo *attr.name* y debe ser único en el documento, los tipos de datos de los atributos pueden tener los tiposde datos de Java.

El dominio del atributo indica para qué elementos del grafo es aplicable dicho atributo, es decir, para el grafo, el nodo, las aristas o todo.

```
<key id="d1" dor="edge" attr.name="weight" attr.type="doble"/>
```

También es posible declarar valores default para estos atributos.

A partir de ésta información y la experimentación que se ha hecho sobre la construcción de un grafo mediante el esquema XML, se ha automatizado la construcción del archivo. Esto se hace mediante expresiones regulares, por medio de las cuales, una vez obtenido el grafo de entrada, se crea dicho archivo con las especificaciones pertenecientes al grafo (los nodos, sus pesos y el peso de sus enlaces) para poder así, mostrarlo de manera gráfica al analista, como se muestra en la siguiente figura.

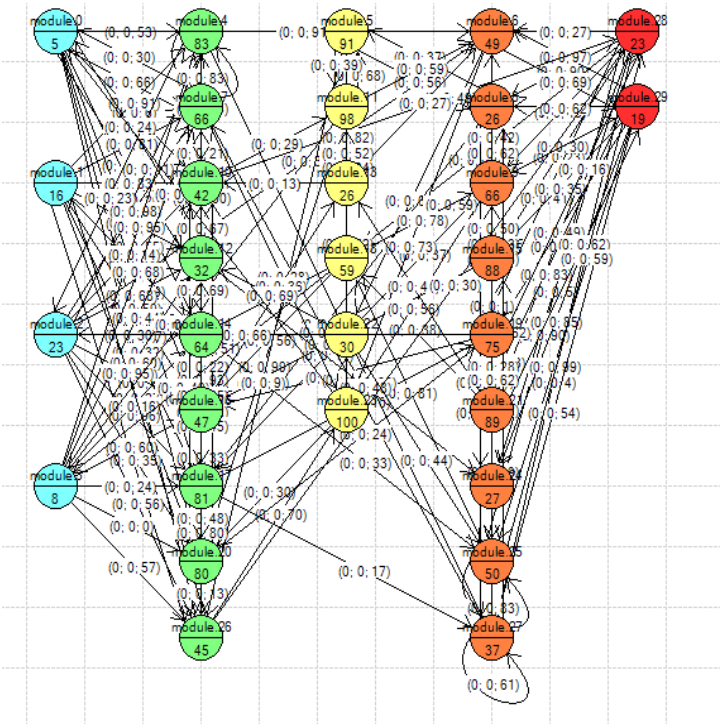

Figura 23 Corrida 1 (30 nodos,60 arcos).

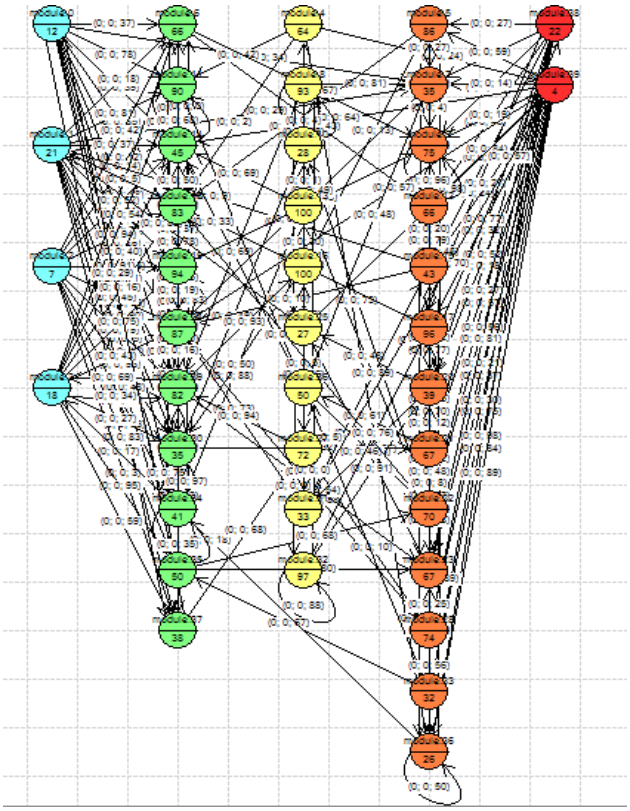

Figura 24 Corrida 2 (60 nodos, 120 arcos).

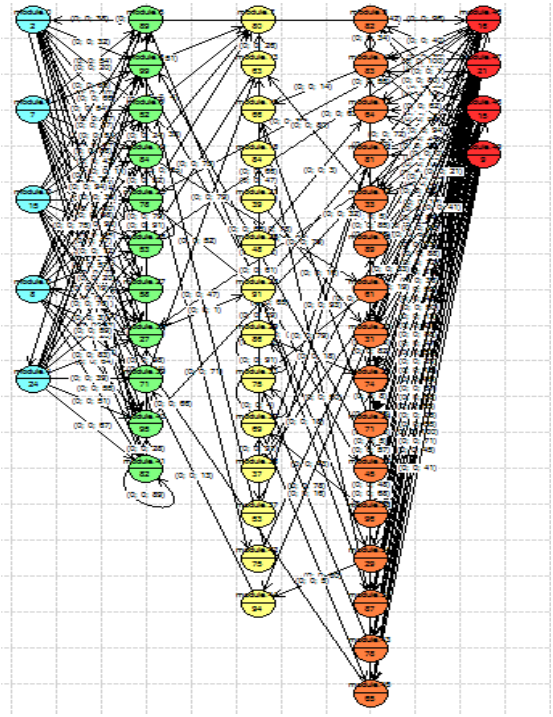

Figura 25 Corrida 3 (50 nodos, 100 arcos).

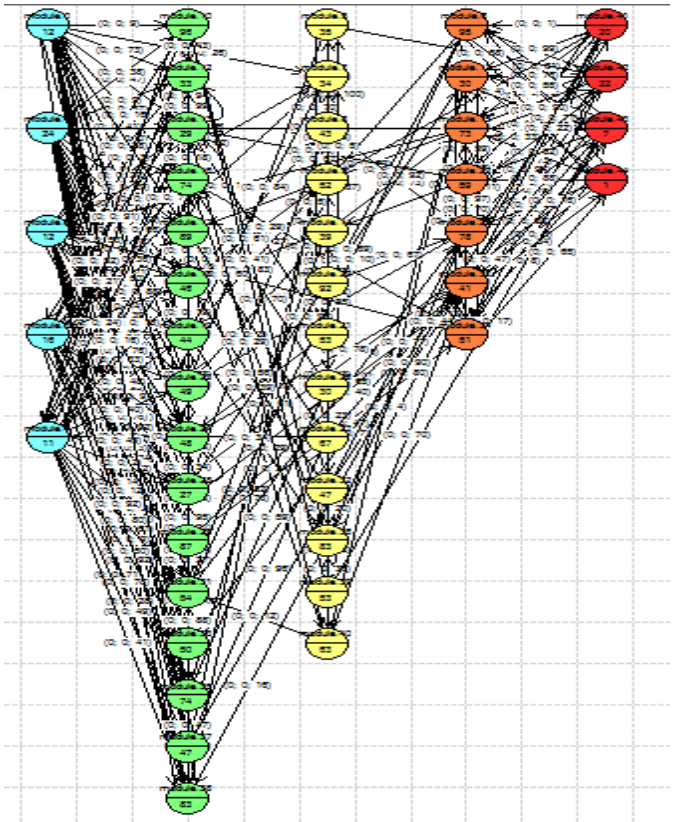

Figura 26 Corrida 4 (45 nodos, 90 arcos).

Como se puede apreciar en estas primeras corridas, el algoritmo de generación de grafos trabaja de manera coherente, puesto que no quedan nodos solos lo cual concuerda con una característica del caso de estudio y no se interpone con el algoritmo de particionamiento.

Un detalle importante descubierto en el proceso de prueba de el modulo en cuestión es que, al trabajar con números aleatorios para saber hacia qué nodos lanzar el arco el grafo se queda cargado hacia un lado, lo cual se puede apreciar ligeramente en las ultimas corridas donde los arcos comienzan a cargarse en la parte inferior de la imagen, es decir, el lado izquierdo del grafo (viéndolo de manera convencional), esto se debe a un problema particular propio del lenguaje C# lo que ocasiona que el número generado se estanque en un rango debido a la naturaleza de su algoritmo de aleatoriedad. Esto no representa un problema cuando se traba de nodos normales pero si se desea crear grafos de mayores magnitudes el problema puede resultar más significativo. Como se aprecia en la figura 26.

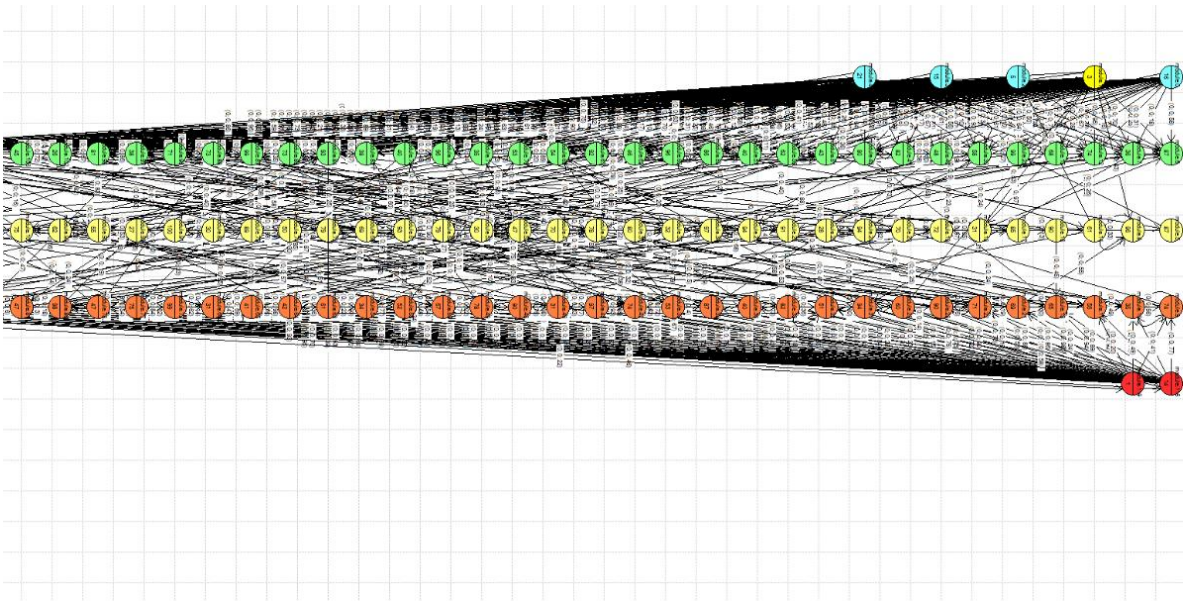

Figura 27 Corrida 5 (120 nodos, 240 arcos).

Tomando en cuenta las aplicaciones y la escalabilidad del problema tratado en éste documento, la generación de grafos también puede resultar ser una herramienta útil para la generación de los mismos y el estudio de su comportamiento según el caso de estudio para el que se requiera. Cabe mencionar que para ello serían necesarias ciertas adecuaciones puesto que los módulos desarrollados y explicados en éste trabajo son para un caso de estudio en particular que es la etapa de análisis y pruebas del proceso de diseño de sistemas/productos digitales.

La figura presentada a continuación muestra una vista previa del documento que describe a un grafo simple en formato GraphML, aunque para aportar contenido a la información visual que se presenta al usuario se ha hecho uso de atributos y métodos añadidos al documento que permiten atribuir otras características a los nodos y las aristas que componen el grafo en cuestión, tales como diámetro, grosor de línea, coordenadas para poder organizar la información y colores para diferenciar los niveles de profundidad del árbol o grafo generado.

Figura 28 Grafo Simple en XML.

Debido a que el formato GraphML es relativamente nuevo con respecto a otros esquemas basados en XML, la mayoría de las herramientas de edición capaces de soportar éstos formatos son de licencia con derechos, es decir, es necesario pagar por el derecho de su

uso. Para esto existe una herramienta llamada Grafos, creada por el profesor Dr. Alejandro Rodriguez Villalobos de licencia libre que permite la visualización de éste tipo de formatos

Al ser un formato complejo y pesado, la herramienta no es capaz de enlazar la herramienta y el formato como se suele hacer con la mayoría de archivos de cualquier formato, por lo que, la herramienta aquí presentada, para dar una solución paliativa y hacer menos problemático éste proceso, crea el archivo GraphML y ejecuta la aplicación, de forma tal que el usuario sólo tenga que ir al menú y abrir el archivo deseado.

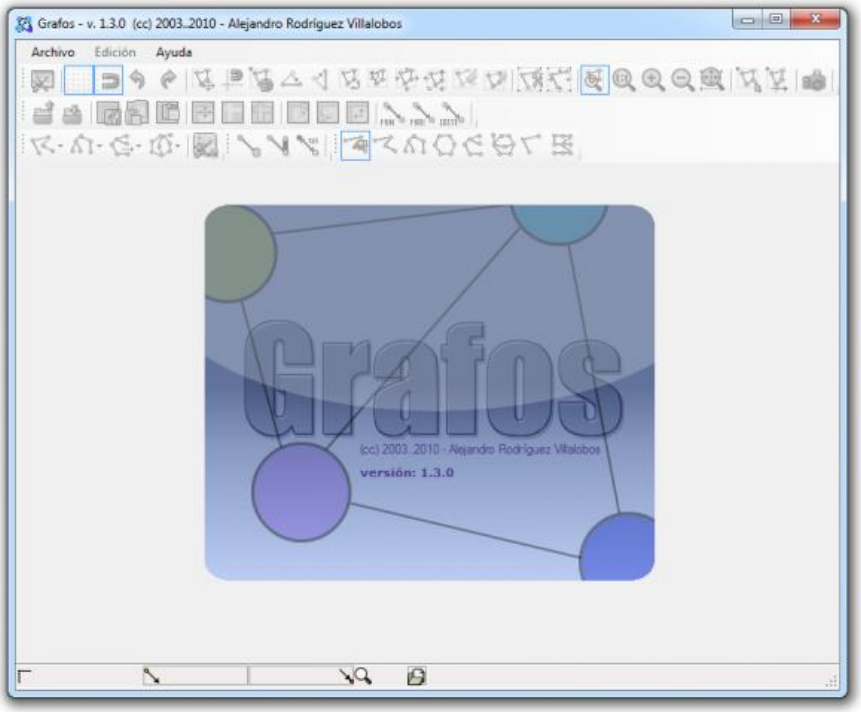

Figura 29 Herramienta Grafos.

4.2 Implementación y Pruebas de la Etapa 2

4.2.1 Abstracción

La tarea de abstracción se realiza en el momento en que se abre cualquier grafo que esté descrito con base en el esquema previamente especificado, de forma que, como muestra en diagrama de casos de uso de la figura 20, el usuario puede elegir que la herramienta solo muestre el grafo de manera visual o que realice el particionamiento sin pasar por la visualización, o bien, realizar ambas tareas en un orden indistinto.

La única forma de realizar pruebas a ésta tarea es mediante los *BreakPoint* que VisualStudio proporciona para la verificación de código en tiempo de ejecución, de forma que se pueda apreciar l que está haciendo el programa en segundo plano, las tareas de consola y el código que no tiene resultados visibles al exterior en caso de aplicaciones con interfaz de usuario. Para ello basta con pasar el cursor por encima de la variable que se desea analizar y el asistente de VS automáticamente muestra la información pertinente a la variable en cuestión como se muestra en las siguientes figuras, cabe mencionar que dicha propiedad también es aplicable a secciones de código específicas:

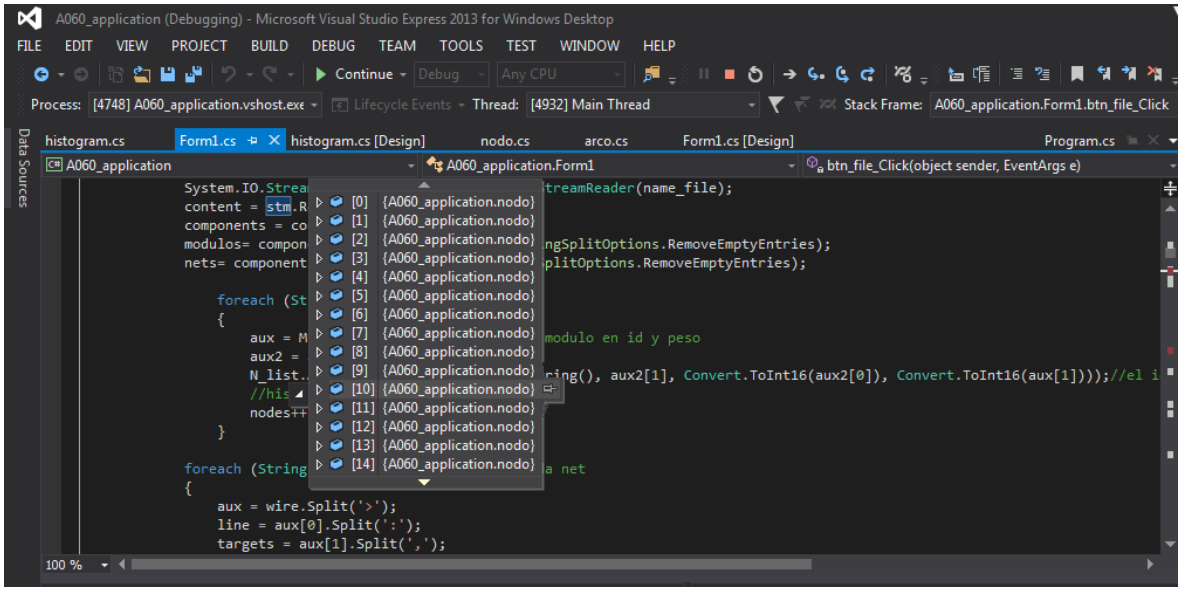

Figura 30 Componentes N_List.

En la figura 28 se aprecia el primer vistazo a la variable *N_ist* que es del tipo de dato *ArrayList* propio de C#, al que se agregan todos los nodos encontrados en el archivo leído, para comprobar que se está añadiendo de manera correcta la información de cada nodo, basta con dar clic en alguno de los elementos agregados para mostrar su contenido como se muestra en la siguiente figura.

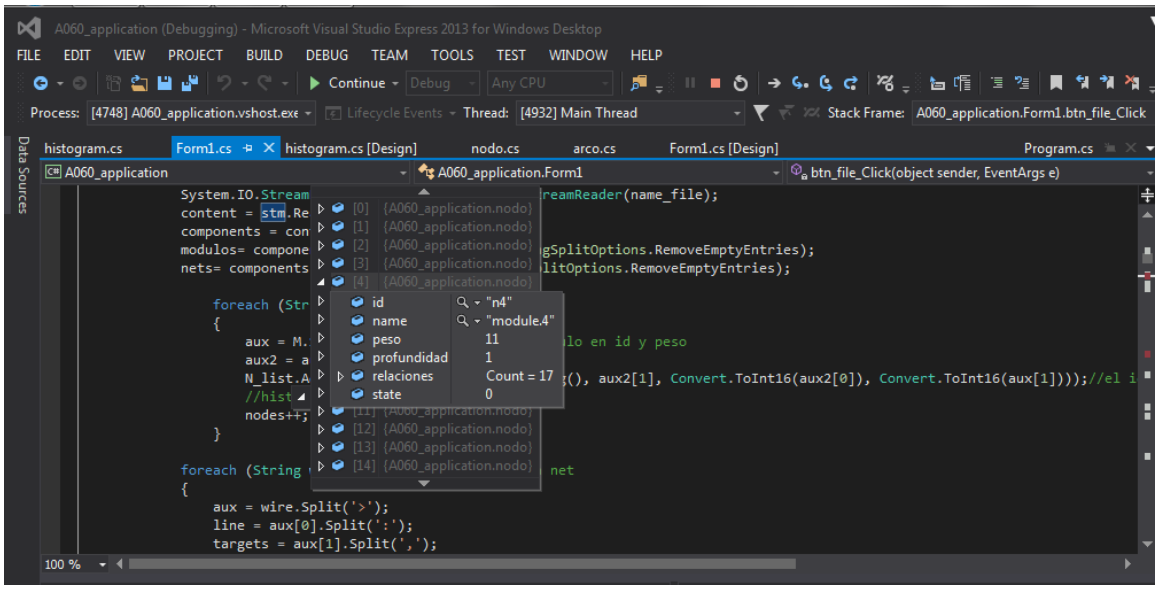

Figura 31 Información del nodo.

Como se puede apreciar, cada nodo agregado tiene su información especificada de manera correcta, es decir, cada dato es asignado a si respectivo campo de forma que se mantiene la congruencia. También se puede comprobar que las relaciones (arcos) se agregan correctamente al desplegar el contenido del campo relaciones, que como se mencionó anteriormente consiste en un campo de tipo *ArrayList* en el que se agrega un objeto de clase Arco por cada nodo al que se conecta el nodo en cuestión como se ve en la siguiente figura.

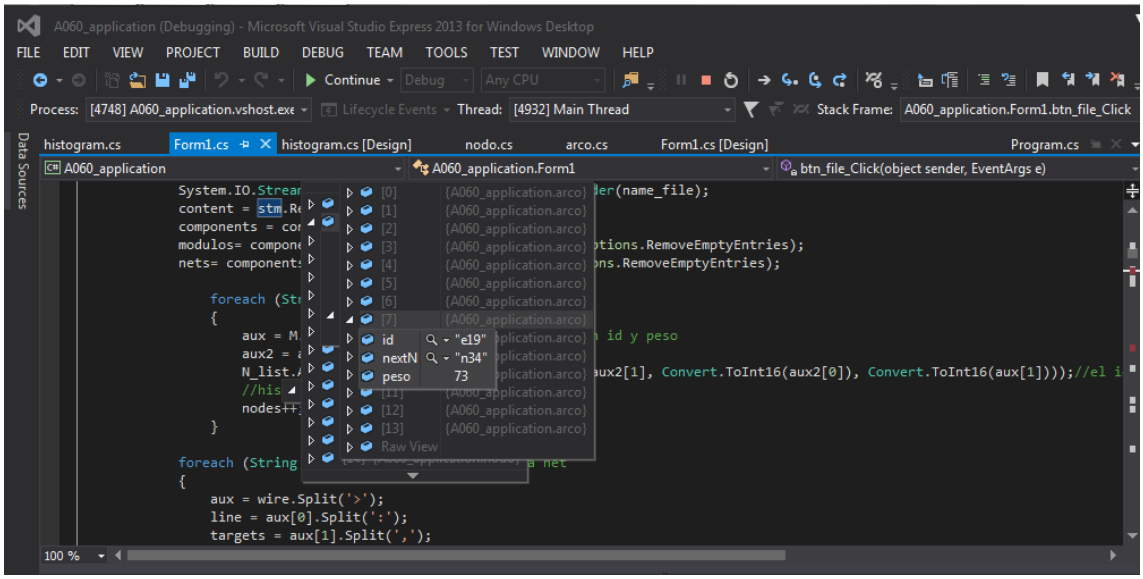

Figura 32 Campo Relaciones Comprobado.

Aquí se puede apreciar el contenido del campo relaciones, y al seleccionar cualquiera de los objetos Arco añadidos al arreglo se puede comprobar también el contenido, de forma tal que con esto queda constatada la correcta abstracción de los grafos de entrada.

4.2.2 Algoritmo de particionamiento

Generación de Conos

Como se describe en las secciones anteriores el algoritmo propuesta en éste trabajo parte de la idea de dos algoritmos ya existentes que resultan aportar criterios y consideraciones que ayudan al algoritmo propuesto a desempeñarse de mejor manera.

La primera etapa del particionamiento consiste en la generación de conos, el cual resulta ser un proceso que proporciona una solución local al equilibrio de comunicaciones entre los grupos generados, ya que, como se puede ver en la figura 27, los conos albergan en sí el peso de la mayoría de las comunicaciones, dejando así, las menos posibles para interactuar con los grupos vecinos. Las bases para esto se han tomado del algoritmo MBCM descrito en [15] que resulta ser un algoritmo al que se puede calificar como empírico pues que, si bien cuenta con bases sólidas matemáticas, jamás ha sido implementado en un lenguaje de programación, lo que implicaría, como suele pasar, aumento en complejidad debido a las limitaciones de éste tipo de lenguajes y las implicaciones inherentes en la abstracción de cualquier algoritmo a lenguaje máquina cuando éstos no han sido pensados en términos computacionales.

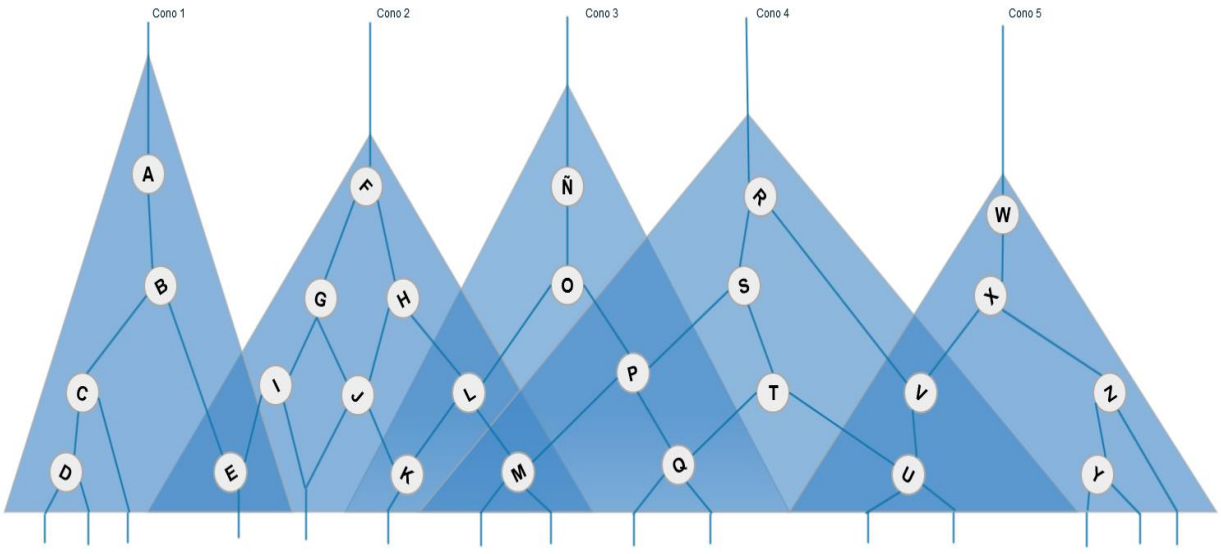

Figura 33 Regiones de superposición de grupos.

El algoritmo consiste en manejar el grafo como un árbol, de forma que el comportamiento y los métodos de desplazamiento sobre un árbol se apliquen al grafo en cuestión, comenzando con las raíces como los nodos *fan-in* y terminando con los nodos de los niveles más profundos como los nodos *fan-out*. Estableciendo un nivel de profundidad para cada fila de nodos como se muestra en la figura 28 de acuerdo con lo mencionado en secciones anteriores, es decir, debido a que no se tiene información al respecto para el caso de estudio en cuestión y los ejemplos parecen tener el mismo esquema, se ha establecido como máximo un nivel de profundidad 5. Esto ha sido programado de forma tal que, si con estudios posteriores resulta ser un número erróneo se puede modificar fácilmente y el algoritmo siga trabajando con mayores profundidades añadidas.

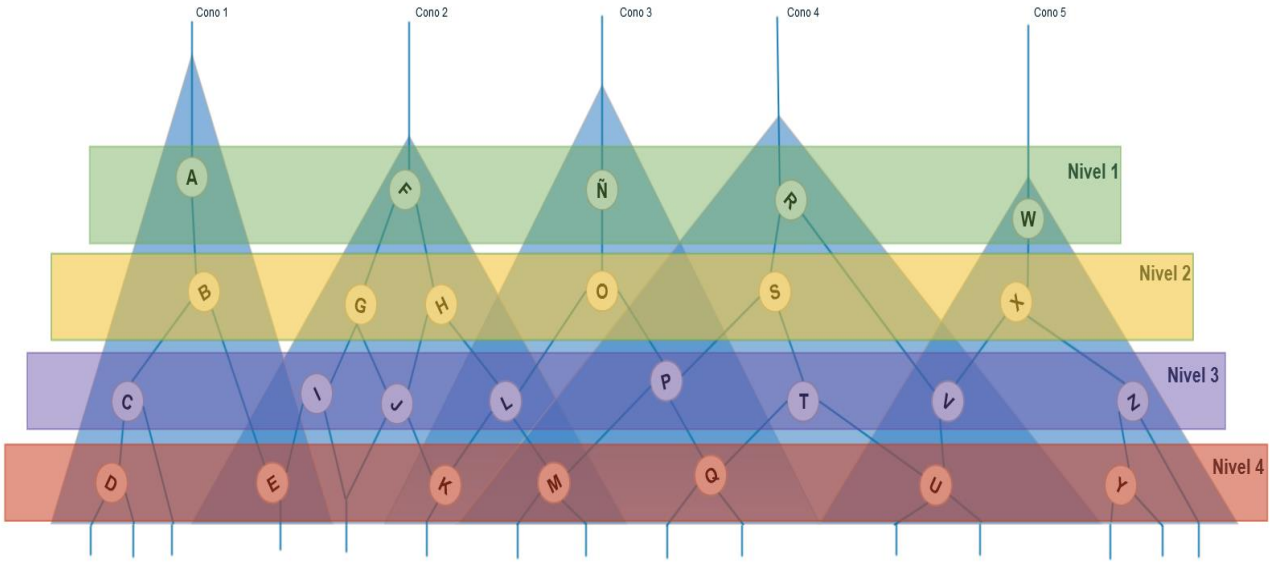

Figura 34 Niveles del árbol.

El siguiente paso es recorrer desde cada nodo de entrada el árbol siguiendo el sub-algoritmo descrito en la figura 29, de forma tal que se creen los conos y se pueda obtener así la primera parte de la solución al problema en cuestión que es el mínimo costo de comunicaciones entre los grupos generados.

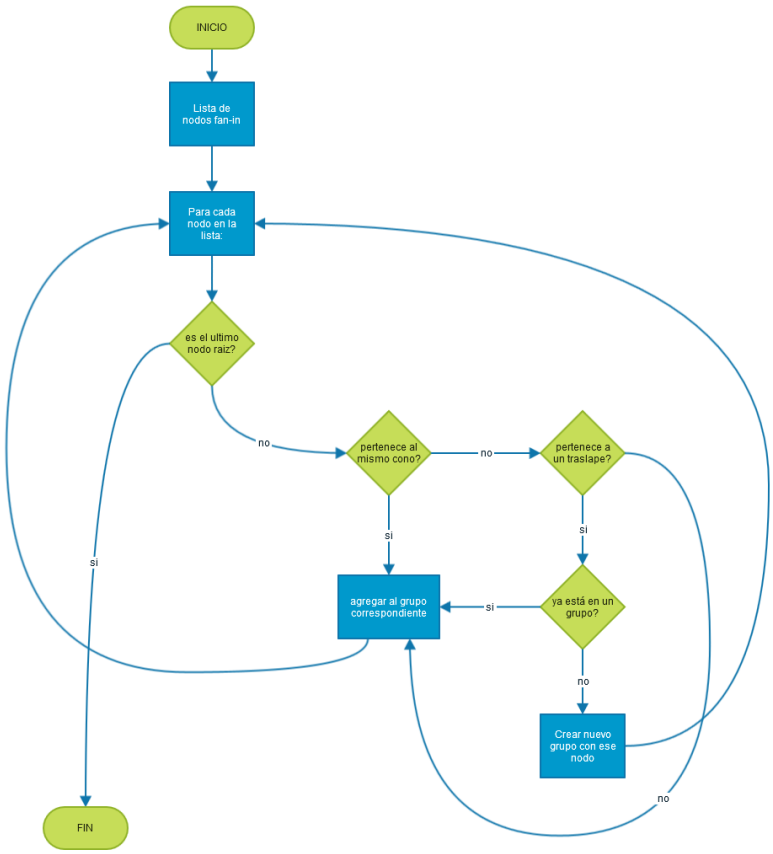

Figura 35 Proceso de separación de traslapes.

Como propuesta para primeros estudios del algoritmo que aquí se presenta, se decidió recorrer el árbol de manera horizontal, de izquierda a derecha primeramente y después en sentido contrario, ya que ésta primera versión del algoritmo consideraría la mejor corrida de las dos para mejorar aún más los resultados. Siguiendo con esta idea se programó el algoritmo para tal efecto con lo cual se encontraron las siguientes características:

- Dado que el recorrido del árbol es de forma horizontal, al realizar la búsqueda de los nodos relacionados al nodo en cuestión no se eliminan por completo los niveles de traslape puesto que, bajo éste esquema no se conoce el contenido del siguiente cono, de forma tal que no se puede saber si el nodo al que se conecta pertenece a otro cono o a un traslape. Esto se resuelve paliativamente haciendo que al momento de agregar nodos al cono se consideren sólo los nodos de niveles superiores, de ésta forma el cono en cuestión no absorbe por completo al cono siguiente.

- El primero cono siempre quedará más cargado en cuanto a nodos y arcos se refiere, esto es producto del mismo problema descrito en el punto anterior.
- El algoritmo se comporta de la misma forma al recorrer el árbol en cualquier sentido, es decir, ya sea de izquierda a derecha o de derecha a izquierda, el primer

cono siempre tendrá la mayor carga de trabajo pues pasa a absorber zonas de traslape.

- Cuando el árbol tiene demasiados nodos de entrada los últimos conos pueden quedar vacíos, es decir, el único nodo en dichos conos es el nodo raíz (el que da paso al algoritmo) debido a que sus enlaces quedan absorbidos en los conos anteriores, como se puede ver en la siguiente figura. NOTA: esto no pasa siempre que se corre la aplicación pero es un comportamiento importante que debe ser tomado en cuenta para su estudio.

```
CONO0::::: TW: 16121|TOE: 8842
CONO1::::: TW: 476|TOE: 576
CONO2::::: TW: 481|TOE: 499
CONO3::::: TW: 444|TOE: 525
CONO4::::: TW: 10|TOE: 0
CONO5::::: TW: 4|TOE: 0
CONO6::::: TW: 14|TOE: 0
CONO7::::: TW: 8|TOE: 0
CONO8::::: TW: 4|TOE: 0
CONO9::::: TW: 11|TOE: 0
CONO10::::: TW: 14|TOE: 0
CONO11::::: TW: 6|TOE: 0
CONO12::::: TW: 16|TOE: 0
CONO13::::: TW: 3|TOE: 0
CONO14::::: TW: 5|TOE: 0
CONO15::::: TW: 15|TOE: 0
CONO16::::: TW: 21|TOE: 0
```

Figura 36 Grafo con muchos nodos fan-in.

La figura 31 muestra el resultado o comprobación empírica de lo que se esperaría de como partición o creación de conos, en donde los niveles de traslape son separados del resto de los conos para generar así conos independientes. Para el caso del algoritmo y el recorrido implementados, los conos inferiores quedarían añadidos a los conos 1-5 y el cono 1 estaría considerando nodos del cono 2 y 7.

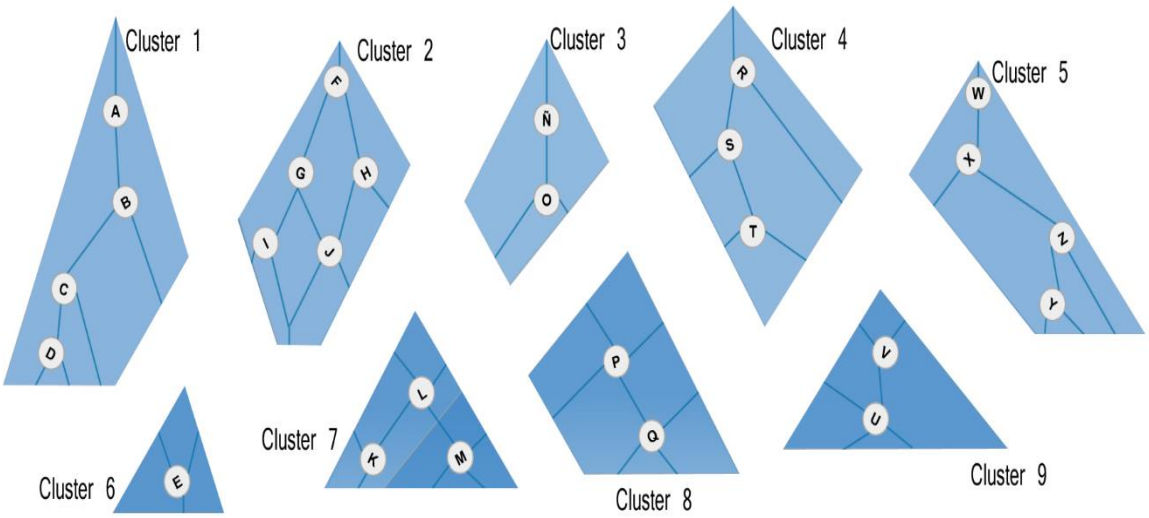

Figura 37 Separación de superposiciones.

Para solucionar esto, se requiere cambiar el enfoque de programación pasando a un esquema multi-hilo, de forma tal que se pueden recorrer todos los conos desde los nodos fan-in al mismo tiempo y ahí separar las zonas de traslape de acuerdo al algoritmo empírico.

Si bien, los resultados obtenidos son satisfactorios para ésta primera versión del algoritmo como se muestra en la tabla a continuación, independientemente de los conos creados, las cargas de comunicación quedan relativamente homogéneas que es precisamente lo que se espera con éste sub-algoritmo para la primera etapa del algoritmo de particionamiento de diseños VLSI propuesto.

Grafo	No. Nodos	No. Arcos	No. Entradas	Grupos generados		
testAG1	30	60	4	Cono	TW	TOE
				Cono 0	1117	232
				Cono 1	16	11
				Cono 2	187	28
				Cono 3	165	25
testAG2	60	120	4	Cono	TW	TOE
				Cono 1	3298	558
				Cono 2	60	33
				Cono 3	116	47
				Cono 4	8	25
testAG3	90	180	4	Cono	TW	TOE
				Cono 0	4985	871
				Cono 1	11	35
				Cono 2	154	59
				Cono 3	63	43
testAG4	120	240	5	Cono	TW	TOE
				Cono 0	16121	8842
				Cono 1	476	576

				Cono 2	481	499
				Cono 3	444	525
				Cono 4	10	0
				Cono 5	4	0

Tabla 10 Generación de Conos. TW (Total Work). TOE (Total Output Edges).

Como se puede apreciar en la tabla 10, se ha dispuesto un promedio de 4 entradas para cada grafo creado debido a que, como se mostró en la figura 30, independientemente de la cantidad de entradas que el grafo contenga el algoritmo puede llegar a crear sólo 4 conos y dejará vacíos los demás. Como se puede apreciar en la tabla 10, el ultimo grafo contiene 16 conos, de los cuales a partir del cono 5 comienzan a quedar vacíos, característica que debe ser estudiada con otros métodos de barrido para saber sus implicaciones.

Como se puede verificar en la siguiente figura, esto no siempre pasa, es decir, el algoritmo trabaja generalmente de forma eficiente, pero es importante documentar éste comportamiento para futuros estudios del algoritmo en cuestión.

```
CONO0::::: TW: 1569|TOE: 336
CONO1::::: TW: 47|TOE: 20
CONO2::::: TW: 22|TOE: 14
CONO3::::: TW: 181|TOE: 33
CONO4::::: TW: 421|TOE: 55
CONO5::::: TW: 98|TOE: 37
```

Figura 38 Prueba d eparticionamiento con 6 entradas.

Otro punto importante de la tabla 10 es que, si bien, el primer cono siempre queda cargado de nodos y aristas salientes (*Total Output Edges*), los demás nodos, independientemente de la cantidad de trabajo resultan cumplir con el cometido de esta primera sección puesto que quedan equilibradas de manera satisfactoria como se demuestra a continuación:

Grafo	Peso de los Componentes	Promedio
testAG1	11	21.33
	28	
	25	
Ya cetestAG2	35	35.66
	47	
	25	
testAG3	35	45.66

	59	
	43	
testAG4	576	533.33
	499	
	525	

Tabla 11 TOE's equilibrados mediante algoritmo de Conos.

Dejando de lado el primer cono y sumando el total de aristas salientes cada cono, se puede obtener un promedio, el cual comprueba que la cantidad de aristas salientes no están demasiado alejadas del promedio, por lo que se puede considerar una solución local satisfactoria tomando en cuenta la naturaleza NP de éste tipo de problemas. Logro que resulta ser bastante significativo para una primera versión del algoritmo propuesto.

Es importante recalcar que, el criterio más importante de éste algoritmo es el equilibrio de la carga de comunicaciones puesto que en un sistema distribuido, los algoritmos de sincronización y atención de peticiones resultan ser los más pesados computacional y temporalmente, de aquí que se haya atacado éste problema primero, quedando así en segundo plano la distribución uniforme de la carga de trabajo.

Agrupamiento aleatorio

Ésta tarea tiene como propósito generar una solución global al problema del particionamiento de grafos considerando únicamente los 2 factores importantes requeridos para el caso de estudio del trabajo aquí presentado, es decir, distribuir de manera uniforme la carga de trabajo y poseer el mínimo de comunicaciones en entre cada componente generado, de forma tal que, al estar estos dos factores lo mejor homogeneizados posibles, la plataforma de computación en paralelo que se utilice realizará menos esfuerzo en cuanto a la sincronización de procesos y cada equipo en dicha estación trabajará casi el mismo tiempo que las demás. De esta forma se asegura que la plataforma esté siendo utilizada de manera coherente y por supuesto, el tiempo de simulación se vea mejorado.

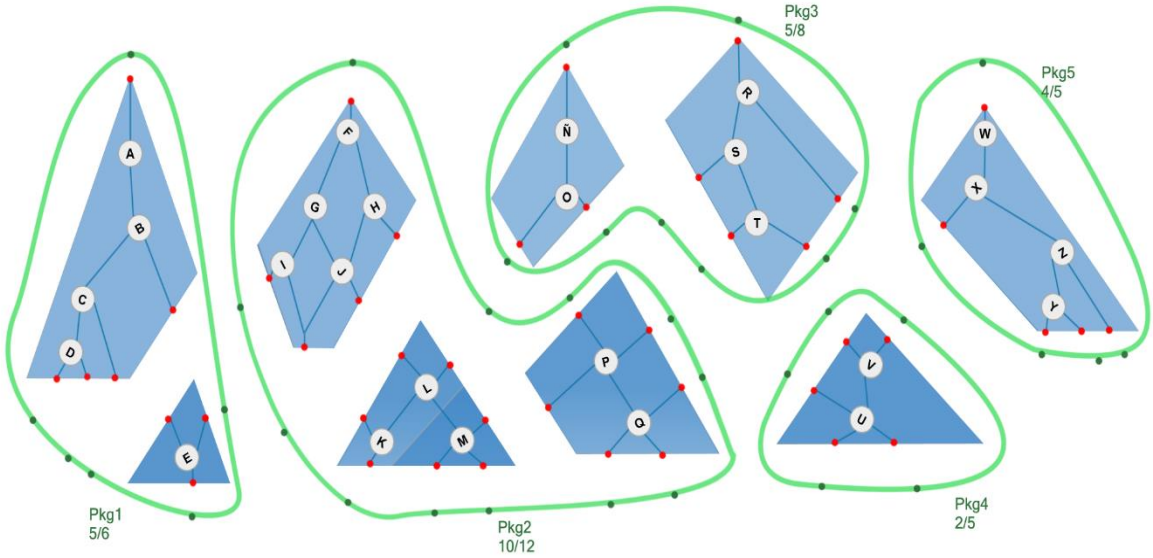

Figura 39 Ejemplo de agrupamiento aleatorio.

Si bien, existen muchas maneras de aleatorizar éste proceso, el utilizado para éste trabajo se basa en índices, debido a que computacionalmente cada cono es un *ArrayList* de objetos de la clase nodo, basta con seleccionarlos de acuerdo a su índice e irlos agregando a nuevos conos bajo criterios específicos y al final elegir la mejor opción de éstas. Para ilustrar tal efecto, la siguiente figura describe el algoritmo implementado específicamente para el presente trabajo y los criterios utilizados.

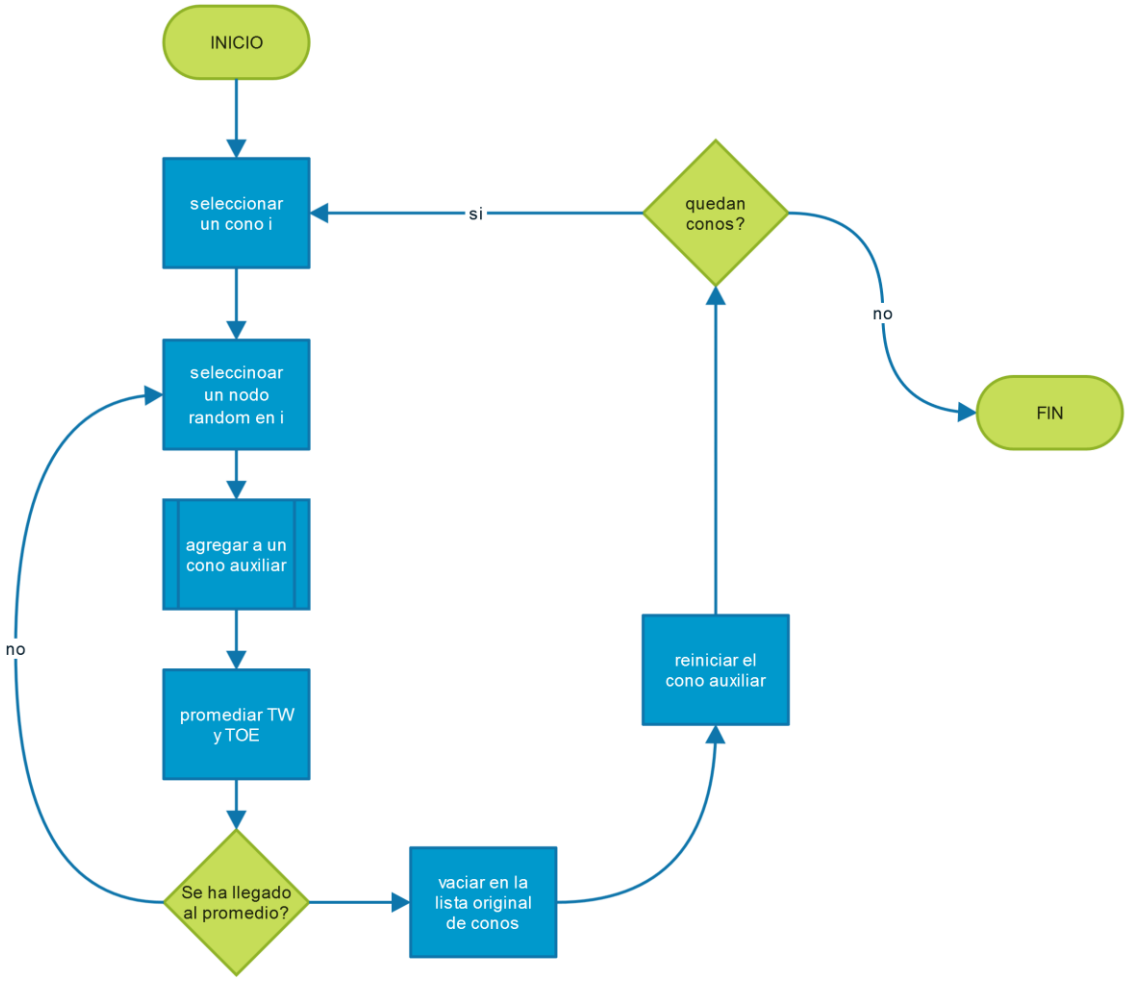

Figura 40 Algoritmo de agrupamiento.

Tomando en cuenta el comportamiento encontrado en la generación de conos descrita en la sección anterior, se ha modificado el algoritmo de forma que si obtenemos el promedio de TOE y TW para tener una referencia y en base a ello generamos el agrupamiento aleatorio, esto hará que, gracias a la desviación estándar se tiene un margen con respecto al valor que se va generando en estos dos factores para los nuevos conos. De ésta forma se asegura que el conjunto de grupos generados y los conos construidos a partir de ello son la mejor opción posible. Esto además de que al estar ya aleatorizados los índices de los nodos en cada cono, seleccionarlos incluso de manera secuencial ya implica aleatoriedad.

Por tanto, en cuanto se sobrepase el valor del promedio se crea un nuevo cono, de forma que, como se mostrará más adelante, el costo de arcos salientes y el peso total del cono resulta estar equilibrado y el número de arcos salientes.

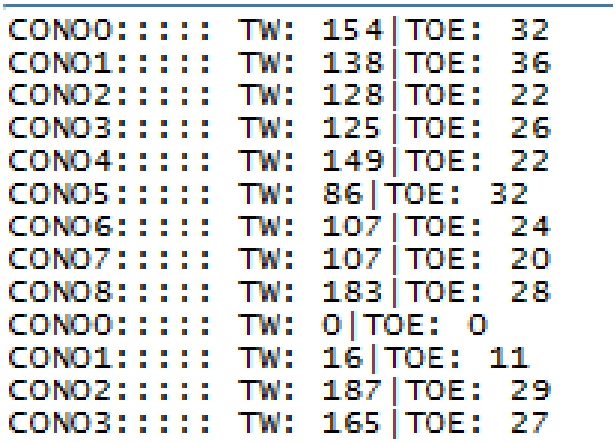

Figura 41 Ejemplo de particion.

Un comportamiento interesante descubierto en éste proceso, es que al final, pueden resultar conos vacíos debido a la naturaleza computacional del algoritmo, es decir, debido a que la premisa del algoritmo es que cada nodo puede ser en si un cono único genera un conjunto de conos del tamaño de la cantidad de nodos habidos en el cono 1 (cono sobrecargado), pero al agrupar nodos teniendo las consideraciones previamente descritas, llegará un punto en que se hayan agregado todos los nodos a los conos iniciales y queden conos “ficticios” vacíos.

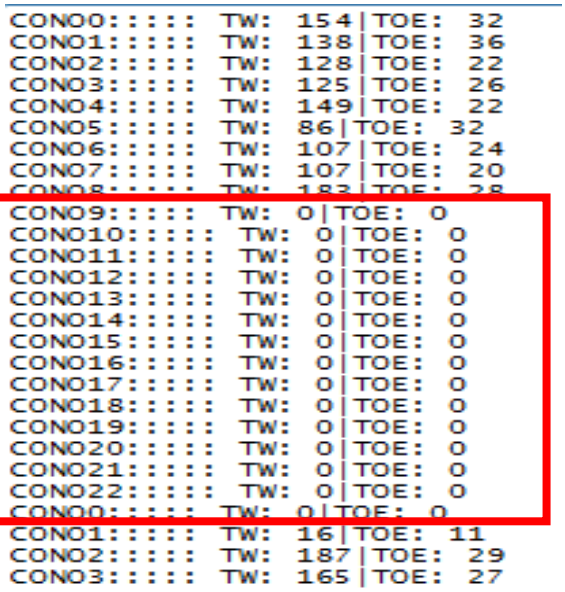

Figura 42 Conos ficticios.

Capítulo 5 Conclusiones y trabajo a futuro

5.1 Conclusiones

- Después de un minucioso estudio al lenguaje HDL elegido particularmente para el presente trabajo (*SystemVerilog*) debido a su disposición para el análisis de diseños, se eligieron como parámetros indicadores los puntos de cobertura y, debido a que la granularidad de corte mínima se ha designado como el nivel módulo, las coberturas especiales previamente mencionadas son las que ayudaran a tal efecto. Cabe mencionar que los puntos de cobertura no son específicamente para conocer el comportamiento del diseño, puesto que solo indican un porcentaje con respecto al total de pruebas realizadas al diseño en cuestión y por tanto, son un aproximado del comportamiento del diseño estudiado.

  La pre-simulación no fue usada como entrada del sistema debido a que, con apoyo del profesor Alfonso Martinez se llegó a la conclusión de que seguir invirtiendo tiempo en la comprensión de los componentes y su elaboración comprometía el resto del proyecto debido a las implicaciones de automatizar dicha tarea, ya que en la actualidad no hay manera de hacerlo. Esto se debe a que primero se requeriría un analizaros léxico/sintáctico que entienda de HDL, de forma que pueda construir a partir de ello un testBench específico para el diseño en cuestión, y un vector universal de forma que se asegure que se están probando todas las operaciones posibles en una sola entrada y así extraer el comportamiento del diseño como se esperaba.

- La representación del diseño a partir de su grafo característico resulto un éxito, independientemente de dónde provenga la entrada (estadísticas del diseño) si está construido con base en el esquema especificado, el sistema de abstracción y construcción XML implementados en ésta herramienta permitirán crear su representación gráfica de manera exitosa, bajo las limitaciones de la herramienta de apoyo para su interpretación (Grafos) como el área útil visible y la falta de interpretación y aplicación de valores default por la deficiencia de su motor interprete. No obstante, la visualización es satisfactoria como se muestra en el capítulo anterior.

- El algoritmo elegido para el particionamiento del diseño fue el algoritmo de conos, en una combinación con el enfoque del agrupamiento aleatorio, debido a las características NP-hard del problema en cuestión. Es importante mencionar que, al ser un algoritmo empírico en términos computacionales, es decir, que fue ideado en términos matemáticos y no pensando en las limitaciones implicadas en el ámbito computacional, el algoritmo de conos sofrió ciertas adecuaciones bajo el criterio de quien desarrolló el trabajo aquí presentado, por lo que pasa a ser una propuesta de algoritmo, es decir, un nuevo algoritmo que toma como base las aportaciones y consideraciones empíricas del algoritmo mencionado. A pesar de esto, como se

puede apreciar en la sección de pruebas, el algoritmo muestra ser útil al proporcionar un esquema del posible particionamiento del diseño.

- En cuanto a la programación se refiere, se ha logrado crear en términos computacionales el algoritmo descrito previamente y las pruebas han demostrado ser satisfactorias para ésta primera versión del mismo, lo que proporciona grandes expectativas para las versiones mejoradas en las que se cubran las deficiencias que no se lograron resolver en éste trabajo y las consideraciones encontradas que, añadidas pueden proporcionar mejores resultados.

- El conocimiento generado en términos computacionales y empíricos en cuanto a la creación de grafos aleatorios bajo criterios estrictos, el particionamiento de grafos, la construcción/interpretación del estándar GraphML resultan ser importantes por su escalabilidad y su campo de aplicación, ya que son algoritmos creados específicamente para ello (cosa que no existe) y por tanto pueden ser de mucha ayuda para generaciones futuras que quieran estudiar este campo.

5.2 Trabajo a futuro

Continuación se describen las consideraciones, modificaciones y añadiduras que podrían hacerse al trabajo presentado en éste trabajo de forma tal que, ya sea para darle continuación o utilizarlo como referencia o base para trabajos futuros, quien lo tenga información suficiente para trabajar con ello y hacer lo que crea conveniente al respecto:

- Debido a la naturaleza computacional y sus cambios, queda por estudiar la complejidad algorítmica del algoritmo aquí implementado, para referencias y mejoras al respecto.
- Queda por analizar su implementación en otros lenguajes y comprar la eficiencia.
- En cuanto a la construcción del grafo y debido a que tanto como la plataforma de desarrollo aquí usada y la herramienta auxiliar para la visualización de los mismos están en constante mejora, queda para la ésta herramienta particularmente, exportar el grafo como imagen para presentarlo al analista de forma más simple o utilizar las futuras actualizaciones para manejo de GraphML a ser incluidas en la plataforma VS.
- Queda por estudiar otros factores que puedan mejorar el algoritmo y sus resultados, como la posibilidad de fusionar arcos o replicar elementos que no representen un peligro de forma que el trabajo y las conexiones queden mejor distribuidas.
- Queda por encontrar diseños reales que sirvan como entrada al algoritmo, que si bien los grafos creados cumplen con las características necesarias, lo primero haría de ésta una herramienta mucho más completa.
- Queda cambiar el paradigma computacional de forma que, usando la computación multi-hilos se pueda recorrer el árbol de manera vertical y paralela, permitiendo que ciertos criterios sean añadidos y los niveles de traslape queden mejor separados.

- El algoritmo de aleatoriedad de la plataforma VS suele tener ciertas complicaciones descritas en la sección pasada, por lo que queda buscar mejores opciones para tal efecto.

Referencias

[1] N. F. a. V. G. a. J. Fleischamann, A new partition method for parallel simulation of VLSI circuits, París, Francia, 2000.

[2] F. M. Johannes, Partitioning of VLSI Circuits and Systems, Las Vegas,NV,USA, 1996.

[3] M. A. M. Cruz, Cobertura funcional en sistemas digitales, México D.F., 2012.

[4] J. I. R. Martinez, Metodología de monitoreo para validación de circuitos VLSI, México D.F., 2009.

[5] M. L. B. a. J. V. B. a. R. D. Chamberlain, Parallel logic simulation of VLSI Systems, North Carolina and Arizona and Washington, 1994.

[6] Gopikrishna, Verification concepts, Sathyabama Chennai, 2007.

[7] G. C. A. d. l. Á. V. C. A. Flores Lomeli Laura Lorena, Tipos de Diodos, Tijuana, 2012.

[8] C. D. H. Roger D. Chamberlain, Evaluating the us of pre-simulation in VLSI Ciscuit partitioning, Washington.

[9] S. R. a. E. G. Boman, Parallel partitioning with Zoltan, 1991.

[10] K. S. P. S. K. P. Subbaraj, An effective memetic algorithm for VLSI partitioning problem, Tamil Nadu, India, 2007.

[11] R. S. J. F. Norber Fröhlich, A new approach for partitioning VLSI circuits on transistor level, Munich, Alemania, 1997.

[12] J. L. a. L. Behjat, A connectivity based clustering algorithm whit application to VLSI circuit partitioning, Canadá: S.Tsukiyama, 2006.

[13] S. T. a. R. a. F. MAkedon, Circuit partitioning into small sets: a tool to support testing with furtherr applications, Dallas Texas, 1988.

[14] H. H. a. christopher J. Augeri, New graph-based algorithm for partitioning VLSI Circuits, Estados Unidos, 2004.

[15] J. H. G. S. D. Brasen, Finding Best Cones for Random Clusters for FPGA Package Partitioning, Francia: Intel Library, 2009.

[16] A. S.-V. K.J. Singh, A heuristic Algorithm for th fanout problem, 1990.

[17] K. A. E., Estudio de Factibilidad de un proyecto, Universidad del Atlántico, 2010.

[18] A. R. Villalobos, Grafos - software para la contrucción, edición y anpalisis de grafos, España, Pza. Ferrándiz, 2003.

[19] M. G. Corporation., «Coverage Control,» de *Verifying the Quality of Your TestBench with Code Coverage*, 2004.

[20] W. D. Shantanu Dutt, Cluster-Aware Iterative Improvement Techniques for Partitioning Large CLSI Circuits, Ilinois Chicago: ACM, 2002.

[21] C.-I. H. Chen, Graph Partitioning for Concurrent test scheduling in VLSI circuit, Dyton OH, 1991.

[22] C. D. P. Srinivas Patil Prithviraj Banerjee, Efficient circuit parttioning algorithms for parallel logic simulation, Ilinois, 19889.

[23] M. P. Hirendu Vaishnav, Delay optimal partitioning tagetinglow power VLSI circuits, Los Angeles.

[24] Grupal, Desarrolladores con HDLs y FPGAs, Yahoo Groups, 2006.

[25] M. K. a. K. Li, Fast graph partitioning algorithms, Canada: University of Victoria, 1995.

[26] J. L. a. L. Benhjat, A connectivity based clustering algorithm with application to VLSI circuit partitioning, Calgary, 2006.

[27] J. L. a. L. Behjat, A connectivity based clustering algorithm with application to VLSI circuit partitioning, Calgary, 2006.

[28] N. F. a. R. S. a. J. Fleischman, A new approach for partitioning VLSI circuits on trnasistor level, Munich, 1997.

[29] P. S. a. K. S. a. P. S. Kumar, An effective memetic algorithm for VLSI partitioning problem, UK, 2007.

[30] E. A. a. G. G. a. S. a. G. Darlington, An investigation of parallel memetic algorithms for VLISI circuit partitioning on multi-core computers, Ontaio, 2010.

[31] T. L. a. F. M. a. S. Tragoudas, Approximation algorithms for VLSI partitioning problems, Cambridge, 1990.

[32] D. J. a. P. N. K. a. L. S. a. R. Dheeraj, Computer aided partitioning for design of parallel testable VLSI systems., India, 2013.

[33] C. j. A. a. H. h. Ali, New graph-based algorithms for partitioning VLSI circuits, United States, 2004.

[34] F. M. Johannes, Partitioning of VLSI Circuits ans systems, Las Vegas, 1996.

[35] S. D. a. W. Deng, VLSI circuit partitioning by cluster-removal using iterative improvement techniques, Milpitas, 1996.

[36] C. S. a. G. Tumbush, System Verilog for Verification, New York.

[37] F. M. Johannes, Partitioning of VLSI circuits and systems, Munich, 1996.

[38] C. S. a. H. Bauer, Corrolla Partitionig for distributted logic Simulation of VLSI-Circuits, Munich.

[39] S. D. a. W. Deng, Cluster-Aware iterative improvement techniques for partitioning large VLSI circuits, Chicago, 2002.

[40] C.-I. H. Chen, Graph partitioning for concurrent test scheduling in VLSI circuit, Dayton, 1991.

[41] S. P. P. B. a. C. D. Polychronopoulus, Efficient circuit partitioning algorithms for parallel logic simulation, Illinois, 1989.

[42] H. V. a. M. Pedram, Delay optimal partitioning testing low power VLSI circuits, California.

[43] R. D. C. a. C. D. Henderson, Evaluating the use of pre-simulation in VLSI circuit poartitioning, Illinois.

[44] H. S. a. P. S. a. N. Dhavlikar, Partitioning of large HDL ASIC designs into multiple FPFA devices for prototyping and verification, Poland.

[45] F. d. S. J. a. J. L. e. Silva, Researching and partial analysis of overhead of a partition model for a partially reconfigurable hardware in a data-driven machine chicflow, Brazil, 1990.

[46] S. Q. a. Z. Qiu, Constructing Hardware/Software interface using protocol converters, Beijin, 2001.

[47] I. standard, Verilog HDL Reference manual, IEE press, 2001.

[48] T. L. a. Y. G. a. S. L. a. F. Ao, Parallel logic simulation of VLSI systems, ACM computing Surveys, 1994.

[49] H. A. a. C. Tropper, Scalable clustered time warp and logic simulation, VLSI Design, 1998.

Glosario

Driver En el ámbito de la verificación de circuitos, el driver es el encargado de administrar los estímulos y proporcionarlos al DUT.

DUT Por sus siglas en inglés, *Dispositive Under Testing*. Es el diseño/circuito en cuestión, es decir, el producto que está siendo probado para su comprobación funcional.

HDL Por sus siglas en inglés, *Hardware Description Languaje*. Son lenguajes específicos utilizados actualmente en la industria por su potencia y los beneficios/herramientas que brinda para el desarrollo de estos productos.

TestBench También conocido como *Testingworkbench*, es un entorno usado para verificar el correcto funcionamiento de un diseño, ya sea de hardware o software, aunque no es tan común para éstos últimos.

Cobertura Se refiere al porcentaje del diseño que ha sido probado, independientemente de los resultados.

FPGA Por sus siglas en inglés, *Field Programmable GateArray*. Es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada 'in situ' mediante un lenguaje de descripción especializado. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinatorio hasta complejos sistemas en un chip.

VLSI Por sus siglas en inglés, Very Large Scale Integration. Es el acrónimo para describir una escala de integración muy grande en ciertos dispositivos.

ULSI Por sus siglas en inglés, Ultra Large Scale Integration. Es el acrónimo para describir una escala de integración muy grande en ciertos dispositivos.

CUTSIZE Hace referencia al tamaño del corte, es decir, qué tan grandes serán las secciones una vez separadas, o qué tanto serán divididas las entidades, también se le conoce como granularidad de corte.

Time-to-market Es el tiempo que tarda un producto desde que es concebido hasta que sale al mercado.

Fan-Out En sistemas digitales se refiere a la latencia del dispositivo, es decir, la cantidad de señales de salida que éste tiene.