UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA (UNED)

TRABAJO FIN DE MÁSTER

Muestro Aleatorio de Circuitos basado en BDDs

Autor:
Elena PINILLA SEDILES

Tutores:
David Fernández Amorós
Rubén Heradio Gil

Trabajo Fin de Máster presentado bajo el cumplimiento de los requerimientos del Máster Universitario en Investigación en Ingeniería de Software y Sistemas Informáticos para el curso 2021-2022.

28 de septiembre de 2022

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 28/09/2022

Quién suscribe:

Autor(a): Elena Pinilla Sediles
D.N.I/N.I.E/Pasaporte.: 73002813P

Hace constar que es la autor(a) del trabajo:

Título completo del trabajo.

Muestro Aleatorio de Circuitos basado en BDDs

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

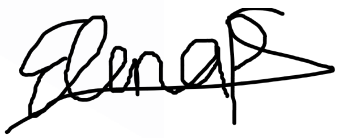
Fdo.

Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es



Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA (UNED)

Resumen

E.T.S. DE INGENIERÍA INFORMÁTICA

DEPT. DE INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS (ISSI)

Máster Universitario en Investigación en Ingeniería de Software y Sistemas Informáticos

Muestro Aleatorio de Circuitos basado en BDDs

de Elena PINILLA SEDILES

El muestreo aleatorio de circuitos integrados es una técnica establecida que consiste en la generación de muestras aleatorias aproximadamente uniformes, ya que todas las configuraciones tienen la misma posibilidad de aparecer en la muestra, a partir de una traducción del circuito a lógica booleana. El principal inconveniente de esta metodología es su escalabilidad, que se vuelve cada vez más apremiante con el aumento progresivo del tamaño y la complejidad de los circuitos en las aplicaciones actuales. Aquí se propone el uso de un conjunto de diagramas de decisión binarios (BDDs) para muestrear las traducciones de estos circuitos. La uniformidad está garantizada por el diseño, y la escalabilidad mejora considerablemente. Para evidenciarlo, este documento incluye una validación experimental de nuestra herramienta, comparándola con otras cinco herramientas de última generación. Los resultados han mostrado que este nuevo enfoque supera al resto de herramientas, tanto en términos de rendimiento, donde se han reducido los tiempos de generación en al menos un orden de magnitud, como en términos de escalabilidad, ya que el circuito más grande de 3402 variables se ha muestreado en 0.77s, mientras que el resto de herramientas han fallado o tomado varios segundos.

Agradecimientos

Me gustaría agradecer sobre todo a mis tutores David Fernández Amorós y Rubén Heradio Gil su constante apoyo, disponibilidad y ayuda durante los meses en los que he estado trabajando en este Trabajo Fin de Máster.

Por supuesto también quiero agradecer la paciencia y compresión de mi familia, amigos y pareja durante mis ausencias y momentos de inquietud. Gracias también por los inagotables ánimos que siempre me han transmitido.

Índice general

Índice de figuras

Índice de cuadros

Índice de Algoritmos

Lista de Abreviaturas

BDD	**B**inary **D**ecision **D**iagram - Diagramas de Decisión Binarios
CNF	**C**onjuctive **N**ormal **F**orm - Forma Conjuntiva Normal
d-DNNF	**d**-**D**eterministic **D**ecomposable **N**egation **N**ormal **F**orm - Forma Normal de Negación Descomponible Determinista
EDA	**E**lectronic **D**esign **A**utomation - Automatización de Diseño Electrónico
IC	**I**ntegrated **C**ircuit - Circuito Integrado
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers - Instituto de Ingenieros Eléctricos y Electrónicos
OBDD	**O**rdered **B**inary **D**ecision **D**iagram - Diagramas de Decisión Binarios Ordenados
SAT	**Sat**isfiability - Satisfactibilidad
SPUR	**S**atisfying **P**erfectly **U**niform **R**andom - Satisfactorio Perfectamente Uniforme Aleatorio
URS	**U**niform **R**andom **S**ampler - Muestreo Aleatorio Uniforme

Capítulo 1

Introducción

Este primer capítulo introduce la definición del problema planteado en este Trabajo Fin de Máster, la motivación para desarrollarlo, y los objetivos que se persiguen. Asimismo, al final del capítulo, se presenta la estructura del resto del documento.

1.1. Planteamiento del problema

La etapa que más tiempo y coste conlleva en el desarrollo y la generación de circuitos integrados (*Integrated Circuits*, IC) es la etapa de testeo, verificación, y validación de los mismos. Esto se debe principalmente a la complejidad y la escala de integración del IC común, que consta de millones de transistores. Sin embargo, al mismo tiempo esta etapa de pruebas es de gran relevancia, ya que permite detectar defectos y eliminar sus causas para asegurar la fiabilidad y calidad de los circuitos diseñados, así como el cumplimiento de sus especificaciones, a la vez que una detección temprana (especialmente antes de su fabricación) puede reducir muchos costes futuros .

Existen diferentes técnicas de testeo utilizadas para la verificación de circuitos electrónicos. La técnica que se aborda en este trabajo es el muestreo aleatorio difuso (también, o más, conocida como *fuzz testing* o *fuzzing*), que proporciona entradas aleatorias, o muestras, a los circuitos integrados para evaluar su fiabilidad. El muestreo aleatorio presenta numerosas ventajas, ya que puede automatizarse por completo fácilmente, es conceptualmente simple, no requiere ningún conocimiento previo sobre el comportamiento del sistema (prueba de caja negra) y no genera falsos positivos. Además, se trata de una técnica de testeo ampliamente utilizada de manera complementaria, gracias a que localiza errores que otras herramientas no llegan a detectar [1].

Se han desarrollado numerosas herramientas para realizar el muestreo aleatorio de circuitos electrónicos. Para ello, estos circuitos son previamente codificados en Forma Normal

Conjuntiva o CNF (*Conjuctive Normal Form*) [1][2, 3, 4, 5, 6]. En este sentido, cada una de estas herramientas presenta diferencias en torno a los siguientes factores característicos:

- Uniformidad: se dice que una herramienta de muestreo es uniforme si cada posible configuración del circuito se puede generar con la misma probabilidad (es decir, todos los valores tienen la misma probabilidad de aparecer en una muestra).
- Escalabilidad: anglicismo que designa una característica deseable por la cual una herramienta debería mantener un coste computacional directamente proporcional al tamaño del circuito o la entrada, sin perder su "calidad". En otras palabras, presenta una dependencia lineal entre el tamaño de la entrada y el tiempo de generación de las muestras.

Por consiguiente, toda herramienta pretende lograr el máximo grado de uniformidad y escalabilidad. En este contexto, algunas de ellas afirman ser perfectamente uniformes por diseño, aunque una evaluación independiente encontró que estas afirmaciones eran demasiado optimistas [7]. Además, la uniformidad es difícil de lograr debido al problema subyacente que presenta el cálculo del número de soluciones de un CNF, conocido como #SAT o recuento de modelos [2].

En cualquier caso, para la mayoría de las herramientas, el tiempo de cálculo aumenta rápidamente, y no de forma proporcional sino frecuentemente exponencial, con el tamaño de los circuitos. Esto se debe principalmente al mayor número de variables implicadas, lo que resulta en problemas de escalabilidad. Esta debilidad presente en numerosas de las herramientas de muestro aleatorio puede llegar a suponer un gran inconveniente para el desarrollo de circuitos electrónicos, los cuales evolucionan hacía tamaños más reducidos, pero conformando muchos más elementos, lo que se traduce en un mayor número de variables.

Dada la relevancia de estos dos aspectos, ellos constituyen la base de evaluación aplicada para la comparación y el análisis de las distintas herramientas presentadas a lo largo de este trabajo.

La propuesta que se recoge en este Trabajo Fin de Máster consiste en el uso de una base o conjunto de diagramas de decisión binarios (BDD) para generar valores de muestreo aleatorios

[1] Conjunto de cláusulas, donde una cláusula es una disyunción de literales, y un literal consiste en una variable o su negación.

[2] Para conseguir la probabilidad uniforme, la práctica habitual es contar explícita o implícitamente el número de soluciones posibles que tiene una fórmula, lo que se conoce como el problema #SAT. Sin embargo, al contrario de lo que pasa con el problema de satisfactibilidad, SAT, un problema #SAT puede presentar gran dificultad incluso para casos tan sencillos como en los que la fórmula sólo está formada por cláusulas de 2 literales. En teoría, la representación en BDD simplifica mucho este problema ya que sólo se requiere recorrer los nodos una vez. No obstante, construir el BDD puede implicar mayor complejidad.

para los circuitos integrados con el fin de ofrecer una propuesta ganadora como algoritmo de muestreo.

En este tipo de diagramas, cada puerta lógica que conforma el circuito está representada por un BDD, lo que ayuda a aliviar los posibles problemas derivados de construir grandes BDDs. A diferencia de generar un solo BDD para todo el problema, este enfoque presenta una mayor escalabilidad.

El conjunto de BDDs resultante, como representación del circuito, se usa posteriormente para generar muestras aleatorias de las variables de entrada del IC de una manera muy eficiente, como se pretende demostrar en este trabajo.

El uso de varios BDDs, en lugar de uno, generalmente implica una pérdida de uniformidad, lo que no sería un problema en las pruebas difusas, por ser las mismas enfoques de *soft computing*. Sin embargo, se mostrará que, en este caso, el algoritmo de muestreo es uniforme por diseño.

Por último, además de presentar en detalle este nuevo planteamiento aquí tratado, se comparará el mismo con diferentes enfoques usando como base de pruebas un conjunto de circuitos secuenciales como referencia, que ha necesitado traducirse previamente a CNF.

1.2. Motivación

La principal motivación para aplicar esta nueva técnica de generación de muestras aleatorias es su gran potencial en el análisis de circuitos, donde estos defectos de hardware son permanentes, al no poder repararse una vez fabricados, y sus consecuencias pueden ser determinantes. Es por ello, que el tiempo de verificación supera al de implementación en el proceso de diseño de circuitos electrónicos. Tanto de forma automática (con técnicas EDA), como de forma manual, se requiere de técnicas de verificación y validación para comprobar la operatividad del circuito y/o encontrar fallos de diseño.

Estas técnicas son de naturaleza diversa, desde el análisis matemático hasta las pruebas simuladas. Una de estas técnicas de verificación es el *fuzz testing* que aquí se trata. Sin embargo, todas ellas presentan los retos de uniformidad y escalabilidad, que se pretenden abordar en esta tesis, y por ello nuestro enfoque, ya de por sí rápido y eficiente, pretende presentar mejoras añadidas en estos aspectos con respecto a las técnicas usadas actualmente.

Por un lado, ofrecer una herramienta con mayor uniformidad está directamente relacionado con su precisión. Si puede asegurarse que las muestras resultantes se han obtenido teniendo

en cuenta todas las variables de entrada con la misma probabilidad, estas serán perfectamente representativas y derivarán en pruebas más precisas.

Por otro lado, una mayor escalabilidad tiene una relevancia trascendental y preponderante para la aplicación que nos ocupa. En esta línea de desarrollo, bien conocida por todos es la Ley de Moore, y su predicción de que el número de transistores en un circuito integrado se duplica cada dos años a medida que avanza el progreso tecnológico [8]. Actualmente, el modelo del iPhone 13 lleva un procesador Bionic A15 que posee 15000 millones de transistores [3]. Un chip de estas características posee innumerables puertas lógicas que derivarían en su correspondiente inmenso número de BDDs. Es por ello que la importancia de esta limitación sólo va en aumento.

1.3. Objetivos

Como se viene ya presentando a lo largo de esta introducción, el principal objetivo de esta memoria es poner a prueba en un nuevo campo de aplicación, como es el muestreo de circuitos, una herramienta de generación de muestras aleatorias basada en una base BDD que ha demostrado tener muy buenos resultados a nivel de varios aspectos relevantes, como son la velocidad de computación y generación de las muestras, su uniformidad, y la escalabilidad de la herramienta para su uso sobre entradas de gran tamaño.

Cabe destacar aquí que este algoritmo ha sido desarrollado por mis tutores. Sus exitosos estudios haciendo uso del mismo han motivado a investigar nuevas potencialidades, como se pretende llevar a cabo en este Trabajo Fin de Máster.

Además de ofrecer su potente velocidad y eficiencia, se pretende encontrar una herramienta que sea competitiva, e incluso superior a la competencia, en estos aspectos de suma trascendencia, sobre todo en este campo en particular (aunque no solamente) como son la escalabilidad y la uniformidad.

1.4. Estructura del trabajo

Este Trabajo Fin de Máster se estructura en 7 capítulos principales, a su vez divididos en numerosas secciones y subsecciones.

[3] https://www.cnet.com/tech/mobile/apples-a15-bionic-chip-powers-iphone-13-with-15-billion-transistors-new-graphics-and-ai/

En este Capítulo 1, se han presentado brevemente los conceptos de muestreo aleatorio, Diagrama de Decisión Binario (BDD), qué representa una base del mismo, y la definición de los términos escalabilidad y uniformidad, en torno a los cuales se fundamentará la argumentación, análisis y comparación de las herramientas a estudio. Además, las distintas subsecciones de este capítulo se han estructurado para presentar cuál es el problema planteado en este Trabajo Fin de Máster 1.1, cuál ha sido la motivación para tratar este problema en este trabajo 1.2, con qué finalidad 1.3, y cómo se llevará a cabo 1.4.

A continuación, en el Capítulo 2 se presenta el estado del arte. A lo largo de sus diferentes secciones, se entra en mayor detalle sobre los conceptos presentados en la introducción, usando de ayuda distintas figuras y ejemplos, así como las familias de algoritmos sobre las que se fundamentan las herramientas, que, a continuación, se exponen de forma individual, y con las cuales se comparará nuestra técnica.

En el Capítulo 3 se lleva a cabo la evaluación empírica de nuestra propuesta. Este capítulo se estructura en tres secciones principales:

- En la primera sección 3.1 se presentan la metodología y etapas seguidas a la hora de realizar este trabajo de investigación, y el enfoque en el que se han basado los experimentos.
- La segunda sección 3.2 se centra en el tratamiento y la gestión de los datos usados como referencia, cubriendo también su previa adaptación, requerida para llevar a cabo las pruebas posteriores. La estructura de las subsecciones de este capítulo sigue un orden cronológico.
- En la tercera y última sección 3.3 se desarrolla la comparación entre las distintas herramientas de muestreo aleatorio atendiendo a diferentes factores, como son el tiempo de generación, el tamaño ocupado, o los grafos en los que están basadas. Esta sección incluye tablas comparativas con los resultados de las pruebas realizadas.

En un posterior Capítulo 4 se analizarán en detalle los resultados arrojados por las comparaciones llevadas a cabo en el capítulo precedente, así como también se razonarán las conclusiones derivadas de los mismos.

Por último, en el Capítulo 5 se expondrán posibles aspectos de mejora y potencialidades que de este estudio podrían derivarse.

Capítulo 2

Estado del arte sobre la Verificación del diseño de hardware

Todos los modelos de desarrollo de software, como son el modelo de cascada, el método en V o el desarrollo ágil, entre otros, se componen de una o varias etapas de verificación antes de la implementación del programa. En el diseño de hardware, tanto a través de herramientas automáticas, como de forma manual, se siguen las buenas prácticas de forma análoga.

Antes de entrar en más detalle conviene diferenciar el significado de los términos validación y verificación, ya que, a veces, se usan indistintamente de forma errónea:

- **Validación**: proceso de evaluación con el cual se pretende asegurar que un producto cumple con los requisitos o requerimientos establecidos y/o solicitados por el cliente, inversor o usuario final.
- **Verificación**: método o proceso con el que se desea confirmar si la salida generada por el producto diseñado cumple con las especificaciones de sus entradas.

Así, en procesos de diseño de software y hardware se encuentran ambas etapas. De acuerdo con esta distinción, la fase del diseño de hardware que se analiza en este trabajo de investigación se corresponde con la etapa de verificación.

Hoy en día, y no sólo en la industria de los semiconductores, el diseño de hardware se lleva a cabo mayoritariamente de forma automatizada a través de herramientas EDA (*Electronic Design Automation*). Estas son, actualmente, la principal técnica utilizada para diseñar sistemas electrónicos de alta complejidad, como circuitos integrados y placas de circuitos impresos, debido a su capacidad para analizar cientos de reglas de diseño físico y eléctrico para millones de transistores a alta velocidad sin necesidad de restricciones definidas por el equipo de desarrollo.

Las herramientas EDA también proporcionan distintas técnicas automatizadas de verificación, que normalmente suelen usarse de forma conjunta, para someter a pruebas a los diseños generados con el fin de ofrecer resultados fiables y de la calidad deseada.

En el desarrollo de este capítulo se describen así mismo algunas de las herramientas de muestreo más modernas, con las cuales se compara nuestra propuesta en la sección 3.3.

2.1. Técnicas de verificación

Las técnicas de verificación del diseño de hardware pueden clasificarse en dos tipos principales: análisis formales (estático) y verificación basada en simulación (dinámico). Los análisis de tipo estático se utilizan para verificar diseños sin aplicarles estímulos. Mientras que, por otro lado, la verificación de tipo dinámico tiene como objetivo verificar los diseños con la ayuda de herramientas de simulación. Un ejemplo de este último tipo de técnicas es el muestreo aleatorio [9].

2.2. Muestreo aleatorio

El muestreo aleatorio o *fuzz testing* fue desarrollado por primera vez en la Universidad de Wisconsin Madison en 1989 por el profesor Barton Miller y sus estudiantes, quienes aún continúan su trabajo en esta temática [1]. El sistema UNIX de Miller falló una noche de tormenta debido a interferencias en la señal, lo que le llevó a pensar en simular su experiencia bombardeando el sistema con señales de ruido aleatorio hasta que colapsase, encontrando así la causa específica de este problema [10]. Tras ello, el muestro aleatorio nació como una técnica de verificación de software, y sus buenos resultados llevaron a estudiar también su aplicación en sistemas de hardware.

Este método de verificación de sistemas consiste básicamente en encontrar errores o defectos en una implementación dada a través de la inyección de datos de entrada de valores aleatorios (siendo estos una combinación de las variables de entrada), y analizando los valores de salida resultantes.

Un programa de muestreo aleatorio se compone principalmente de dos elementos o procesos: la generación de datos y la identificación de vulnerabilidades. El primer proceso lo lleva a cabo un “generador”, y es la parte del método en la que se centra este trabajo, proponiendo un nuevo enfoque para producir las muestras, ya que su optimización tiene gran relevancia

[1] http://www.cs.wisc.edu/ bart/fuzz/

para, entre otros factores, la velocidad, calidad y adaptabilidad del método. Estos generadores pueden hacer uso de diferentes algoritmos, utilizar datos completamente aleatorios o combinar vectores definidos a partir de valores peligrosos, o erróneos, conocidos. La identificación de vulnerabilidades es realizada por las herramientas de depuración. También existe la posibilidad de hacer evolutivos estos procesos, retroalimentando al sistema con la información sobre su comportamiento resultante, haciendo las pruebas posteriores más efectivas.

Como punto de partida, puede disponerse de información sobre el código fuente de los diseños de hardware, detalles sobre las especificaciones de seguridad, información relacionada con la ejecución (simulación o emulación), el uso de la memoria, etc. que el generador de datos aleatorios tendrá en cuenta para generar las entradas.

En cambio, cuando esta información se desconoce, la generación debe ser completamente aleatoria, y para generar las muestras suelen crearse "mutaciones" intercambiando bits, modificando la longitud de las entradas o cambiando el signo, por ejemplo [11]. Sin embargo, este tipo de herramientas que generan muestras por *mutación de semillas*, como suele conocerse este proceso, logran muy baja cobertura después de un largo período de muestreo, ya que la mayoría de los resultados producidos por la mutación son instrucciones inválidas que pueden no tener sentido [9].

Es también común generar un conjunto inicial de datos de entrada, conocido a veces como "corpus inicial", que posteriormente se mutan.

2.2.1. Ventajas y desventajas

Llevar a cabo este tipo de pruebas para la verificación de circuitos integrados presenta una serie de ventajas con respecto a otras técnicas, que fomentan su uso, así como también puntos que pueden considerarse contraproducentes.

La principal **ventaja** de esta técnica de pruebas es su simplicidad, la cual favorece su fácil automatización, su velocidad, su uso como técnica de verificación complementaria, y su sencilla implementación. Una vez que el muestreador está en funcionamiento, no requiere de intervención humana. Además, no precisa premisas ni conocimientos previos sobre el comportamiento del sistema, convirtiéndola en una técnica de pruebas de caja negra, aunque también puede hacer uso de esta información si se conoce, y realizar pruebas de caja blanca. Este aspecto lo convierte en uno de los únicos medios para revisar la calidad de sistemas completamente cerrados, como lo es, por ejemplo, un teléfono SIP (*Session Initiation Protocol*).

Su enfoque aleatorio permite que este método encuentre errores que, a menudo, los ojos humanos habrían pasado por alto, y no genera falsos positivos entre sus resultados, lo que aumenta su fiabilidad [1]. El muestreo aleatorio proporciona, así, una buena imagen general de la calidad del sistema, su solidez y su nivel de seguridad.

En cambio, como **desventajas**, también presenta limitaciones derivadas de su simplicidad, ya que, generalmente, este tipo de verificación tiende solamente a encontrar errores simples. Otro problema a tener en cuenta es que cuando se llevan a cabo pruebas de caja negra, atacando a un sistema cerrado, aumenta la dificultad para evaluar la peligrosidad e impacto de las vulnerabilidades encontradas, no habiendo o reduciendo las posibilidades de depuración [12].

Una limitación específica del muestreo aleatorio aplicado a hardware es que requiere una definición sólida de las entradas para generar mutaciones significativas y llevar a cabo pruebas relevantes [11].

Estos factores se resumen en el siguiente cuadro:

Muestreo aleatorio	
Ventajas	**Desventajas**
Fácil automatización	Encuentra errores simples
No requiere conocer el funcionamiento del sistema (caja negra)	Difícil cálculo del impacto de las vulnerabilidades
Simplicidad	Requiere entradas detalladas
Uso como técnica complementaria	
Exclusivo para sistemas cerrados	
No genera falsos positivos	

CUADRO 2.1: Ventajas y desventajas del muestreo aleatorio

2.2.2. Clasificación

Las herramientas de muestreo aleatorio se pueden clasificar en dos familias atendiendo a la base de representación sobre la que se fundamentan: aquellas que utilizan **estructuras gráficas** como BDD y sus generalizaciones, como la forma normal de negación determinista descomponible (d-DNNF), y los que se basan en **árboles de búsqueda**, comúnmente mejoras sobre los algoritmos Davis-Putnam-Logemann-Loveland (DPLL) [13] tradicionalmente utilizados como contadores de modelos (problema #SAT).

Estas representaciones presentan grandes diferencias que se introducen a continuación, con ejemplos de cada una de ellas.

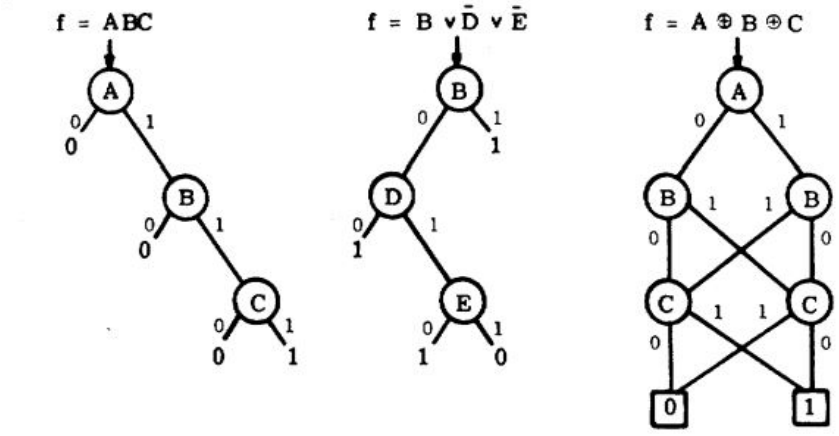
BDD

Los **Diagramas de Decisión Binarios** fueron introducidos inicialmente en 1958 por C. Y. Lee como Programas de Decisión Binaria, siendo estos una formulación de programas mucho más eficiente por su menor tamaño y mayor rapidez, que la representación booleana, entonces tomada como referencia al ser el fundamento de la teoría de conmutación [14]. Posteriormente, en 1978, Sheldon B. Akers introduciría el término de BDD, como un “método para definir, analizar, probar e implementar funciones digitales grandes” [15]. Desde entonces, los BDDs se han convertido rápidamente en el método elegido para representar y manipular funciones booleanas dentro de los ordenadores [16].

Un BDD [17] es un grafo dirigido acíclico con una raíz, que se interpreta de arriba a abajo y representa valores lógicos como una alternativa a las fórmulas lógicas. Para ello, se ordenan las variables, donde cada una de las cuales tiene un nodo asociado a ella, el cual, a su vez, tiene asociados dos "hijos": el "hijo bajo", conectado con una línea discontinua de puntos, corresponde a poner la variable a falso, y el "hijo alto", conectado mediante una línea recta, corresponde a establecer la variable a un valor verdadero. El primer nodo se conoce como “raíz”, y cada una de sus “ramas” define un camino en el diagrama para cualquier valor booleano posible que pueden tomar sus variables, hasta un nodo “terminal”, que puede tomar los valores ’0’ (falso) o ’1’ (verdadero) [16]. La variable asociada al “padre” siempre precede a la de los "hijos".

En la Figura 2.1 se representan las funciones AND, OR y XOR, respectivamente, en forma de BDD.

FIGURA 2.1: Representación en BDD de las funciones AND, OR y XOR [15]



Los BDDs son similares a los árboles de decisión binarios, pero utilizan un grafo en lugar de un árbol para reducir el número de nodos, fusionando subgrafos isomorfos en uno solo. Además, el "padre" se omite cuando ambos hijos son iguales. Estos se conocen como BDDs reducidos. Aunque existen otros, los tres tipos de BDDs más comunes son:

- Reducido.
- Ordenado: todas las variables están ordenadas, y la variable padre siempre precede a la variable hijo.
- ROBDD: Reducido y ordenado.

La herramienta analizada en este trabajo se basa en la generación de ROBDDs, los cuales, además, son los más comunes.

La construcción de un BDD no requiere que la fórmula de entrada tenga un forma lógica concreta, como a menudo se precisa que esté expresada en CNF, lo cual es una ventaja sobre los métodos DPLL, que dictan este común uso de CNF mencionado. En un BDD, el número de nodos que lo conforman y su viabilidad son muy sensibles al orden que siguen las variables.

En el siguiente capítulo podrá verse otro ejemplo de BDD generado a partir de uno de los circuitos ejemplo en la Figura 3.3.

DPLL

Los algoritmos DPLL [13] se basan en construir y recorrer parcialmente un árbol de búsqueda binaria "hacia atrás" (*backtracking*) para comprobar la satisfactibilidad de fórmulas de lógica proposicional en una forma normal conjuntiva, lo que se conoce como solucionar el problema CNF-SAT [2]. De esta manera, estos algoritmos constituyen la base de muchos *SAT solvers* y demostradores de teoremas.

Los algoritmos de vuelta atrás o *backtracking* originales son usados principalmente como un método para encontrar soluciones a problemas de satisfacción de restricciones. Su funcionamiento consiste en la inicial elección de un literal de la fórmula al que se le asigna un valor de verdad, y se logra, así, simplificar la fórmula. A continuación, de forma recursiva, se comprueba si la fórmula resultante es satisfactible, concluyendo la satisfacibilidad de la fórmula inicial completa si lo es. En caso de que no lo sea, la verificación recursiva termina asumiendo el valor de verdad contrario. Con ello se logra dividir un problema en subproblemas más simples.

[2]El Problema de Satisfactibilidad CNF, o CNF-SAT, es una versión del Problema de Satisfactibilidad Booleana, donde, en su lugar, la fórmula booleana se expresa en CNF.

Inicialmente, en 1960, Martin Davis y Hilary Putnam desarrollaron un primer algoritmo de Davis-Putman como solución para el problema CNF-SAT. Este algoritmo se basaba en fórmulas expresadas en CNF, y, eligiendo en ellas variables de forma iterativa, se aplicaban las siguientes tres reglas de forma consecutiva para definir si las fórmulas resultantes de aplicarlas eran satisfactibles o no, pudiendo extrapolar el resultado a la fórmula original [18]:

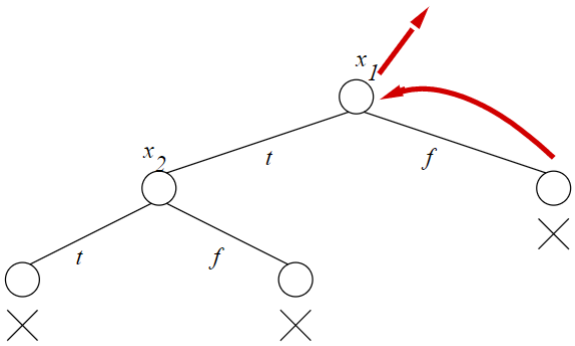
- *Regla para la eliminación de cláusulas con un único literal*: si una fórmula contiene una cláusula con un único literal p, y asimismo otra cláusula con un único literal $\overline{p}$, la fórmula es inválida, e igual a 0.
- *Regla de la eliminación pura literal*: una variable puede eliminarse de una fórmula, si todas sus apariciones son únicamente en forma afirmativa, o únicamente en forma negativa (o negada). Esta regla únicamente es válida para el problema SAT, no para #SAT.
- *Regla de la eliminación de fórmulas atómicas*: dada una variable en la fórmula, sus cláusulas pueden dividirse en aquellas en las que la variable aparece en forma afirmativa, aquellas en las que aparece negada, y aquellas en las que no aparece, permitiendo eliminar la variable tras agruparlas.

Posteriormente, en 1961, Martin Davis y Hilary Putnam, junto con George Logemann y Donald W. Loveland mejoraron el algoritmo anterior, basándose en la vuelta atrás, mediante la aplicación de la *regla de la unidad de propagación*, por la cual una cláusula unitaria solo puede ser satisfactible mediante la asignación del valor necesario para hacerla verdadera[13].

La representación gráfica del funcionamiento de este algoritmo puede verse en la Figura 2.2. En esta figura se representa el siguiente ejemplo [19]:

$$
\begin{gathered}
\{\neg x_1 \vee x_3 \vee x_4, \\
\neg x_2 \vee x_6 \vee x_4, \\
\neg x_2 \vee \neg x_6 \vee \neg x_3, \\
\neg x_4 \vee \neg x_2, \\
\mathrm{x}_2 \vee \neg x_3 \vee \neg x_1, \\
\mathrm{x}_2 \vee x_6 \vee x_3, \\
\mathrm{x}_2 \vee \neg x_6 \vee \neg x_4, \\
\mathrm{x}_1 \vee x_5, \\
\mathrm{x}_1 \vee x_6, \\
\neg x_6 \vee x_3 \vee \neg x_5, \\
\mathrm{x}_1 \vee \neg x_3 \vee \neg x_5\}
\end{gathered}
$$

FIGURA 2.2: Representación del algoritmo DPLL [19]



En una primera iteración se ha seleccionado la variable x_1, a la que se le asigna un valor verdadero, y, sustituyéndola en la fórmula total, la variable x_5 aparece siempre negada, por lo que se le asigna este valor y se elimina su cláusula (aplicando la mencionada *regla de la eliminación pura literal*), simplificando la fórmula. En un segundo paso, se elige la variable x_2 y se le asigna también un valor verdadero, lo que lleva a una contradicción. Por este motivo, se rectifica y se repite la operación con $x_2 = false$, lo cual vuelve a ser contradictorio, llevando a afirmar $x_1 = false$, pero esto conduce también a una contradicción. Por ello, como resultado final, se concluye que la fórmula es insatisfactible.

2.3. Algoritmos de muestreo aleatorio

Una vez presentadas las familias de algoritmos que conforman las bases de los principales algoritmos de muestreo, en esta sección se presentan en detalle los cinco seleccionados, por ser de mayor relevancia en la actualidad, para comparar nuestro algoritmo en estudio en este trabajo. Previamente, también se hará una pequeña introducción sobre los algoritmos de muestreo aleatorio uniforme (Uniform Random Sampler, URS).

El primer algoritmo URS para fórmulas lógicas fue presentado por Donald Knuth en [16]. Este algoritmo usa un BDD como entrada, aunque algunas propiedades del BDD, como anotarlo, se calculan antes de que tenga lugar la generación. Este paso se representa con pseudocódigo en el Algoritmo 1, donde los nodos en el BDD se recorren en orden topológico inverso, es decir, de abajo hacia arriba, y, para cada nodo, el número de soluciones derivadas válidas se calcula (en las líneas 8 y 9) utilizando la misma información de sus dos hijos (“low” y “high”). A continuación, esta información se usa para asignar a cada nodo la probabilidad (en la línea 11) de llegar al nodo 1 a través del hijo “alto”, lo cual se conoce como *anotar el BDD*.

Algoritmo 1 Algoritmo de Knuth para la anotación de nodos

1: **function** ANNOTATE(bdd)
2: **for** $t \in ReverseTopologicalOrder(bdd)$ **do**
3: **if** $t = 0$ **then**
4: $count(t) \leftarrow 0$
5: **else if** t = 1 **then**
6: $count(t) \leftarrow 1$
7: **else**
8: $thenCount \leftarrow count(t.high)2^{level(t.high)-level(t)}$
9: $elseCount \leftarrow count(t.low)2^{level(t.low)-level(t)}$
10: $count(t) \leftarrow thenCount + elseCount$
11: $Pr(t) \leftarrow thenCount/count(t)$
12: **end if**
13: **end for**
14: **return** bdd
15: **end function**

La Figura 3.3 del siguiente capítulo, mencionada anteriormente, muestra un BDD ya anotado con sus correspondientes probabilidades en cada nodo tras haberse seguido este algoritmo.

Una vez se anota el BDD, el paso de generación se realiza usando el Algoritmo 2. Aquí, el gráfico se recorre desde la raíz generando números aleatorios (línea 11) y comparándolos con la probabilidad del nodo (línea 15) para decidir si el hijo "bajo" o el hijo "alto" es el visitado a continuación, lo cual decide el valor de la variable correspondiente, hasta llegar al nodo 1. Así se genera un BDD de estados aleatorios.

2.3.1. UniGen

Los primeros algoritmos de *UniGen* [20, 21] nacieron en 2014 como resultado de iteraciones posteriores de dos herramientas: *UniWit*, presentada por Chakraborty et al. [22], y *PAWS*, de Ermon at al. [23]. Tanto *UniWit* como *PAWS* hacían uso de funciones *hash*.

Por un lado, *UniWit* empleaba funciones *hash* universales e independientes, seleccionadas aleatoriamente, para dividir el espacio de búsqueda en conjuntos de celdas aproximadamente equivalentes, bien "equilibradas" y más simples. Una vez que se decide un conjunto de celdas, se aplica el solucionador *CryptoMiniSAT*[3] [22].

[3]https://www.msoos.org/cryptominisat2

Algoritmo 2 Algoritmo de Knuth para el muestreo aleatorio

```
1:  function GENERATE(bdd)
2:      state ← secuencia del tamaño del nivel(1) inicializada a falso
3:      pos ← 0
4:      trav ← root(bdd)
5:      if trav = 0 then                                  ▷ bdd representa falso
6:          return state
7:      end if
8:      while trav ≠ 1 do
9:          while pos < level(trav) do                    ▷ Salto de nivel
10:             state[pos] ← randomBoolean()              ▷ Igualmente probable
11:             pos ← pos + 1
12:         end while
13:         if random() ≤ Pr(trav) then                   ▷ 0 ≤ random() ≤ 1
14:             trav ← trav.high
15:             state[pos] ← true
16:         else
17:             trav ← trav.low
18:         end if
19:         pos = pos + 1
20:     end while
21:     while pos < level(1) do                           ▷ Salto de nivel
22:         state[pos] ← randomBoolean()
23:         pos ← pos + 1
24:     end while
25:     return state
26: end function
```

Por otro lado, *PAWS* es un algoritmo de muestreo basado en la integración del conjunto en un espacio de mayor dimensión que luego se proyecta al azar usando funciones *hash* a un subespacio de menor dimensión, usando un modelo gráfico, y se explora usando métodos de búsqueda combinatoria. Así, se pueden aprovechar herramientas rápidas de optimización combinatoria [23].

Sin embargo, ambos algoritmos, aunque desmostraron en sus respectivas presentaciones ser considerablemente uniformes, sufren limitaciones inherentes (la computación de funciones *hash*, por ejemplo) que dificultan escalarlos a sistemas con a partir de varios miles de variables [20].

Cabe definirse en este punto el concepto de **soporte independiente** de una fórmula. Para ello, se parte de una fórmula booleana $\mathscr{F}$ expresada en CNF y el conjunto de variables que aparecen en ella, $\mathscr{X}$, al que se conoce como soporte. Tras esto, se define el término *asignación satisfactoria* o *testigo* de $\mathscr{F}$ como cada asignación de variables del soporte a un valor verdadero (*true*), tal que $\mathscr{F}$ también sea verdadera, es decir, se satisfaga. En cada una de estas posibles asignaciones satisfactorias, hay una parte de las variables verdaderas del soporte que determinan de forma única el resto de variables verdaderas. Así, al primer conjunto se conoce como soporte independiente, y, al segundo, como soporte dependiente. Asimismo, normalmente habrá más de un soporte independiente para una fórmula dada.

UniGen se desarrolló entonces como una nueva herramienta para resolver las deficiencias que presentaban *UniWit* y *PAWS*, primando obtener mayor uniformidad al aumentar el número de variables. Así, también basándose en funciones *hash* que dividen los conjuntos de forma aleatoria, *UniGen* empezó a trabajar con un soporte independiente de la fórmula CNF ya que este suele ser mucho más pequeño (hasta varios órdenes de magnitud), y determina por si sólo de forma única los valores de las variables en una asignación satisfactoria de la fórmula de partida, $\mathscr{F}$ [20].

En las versiones posteriores de *UniGen*, llamadas *UniGen2* y *UniGen3* [24], se implementaron mejoras adicionales en el tiempo de ejecución, como permitir la extracción parcial de soluciones y la reutilización inteligente de soluciones. Estos algoritmos siguen girando en torno al concepto de soporte independiente. El conjunto de soporte independiente corresponde a las entradas del circuito, lo que significa que primero sólo se deben muestrear las variables independientes y luego simplemente se deben calcular el resto de los valores. Se supone que estos algoritmos funcionan de una forma muy eficiente y casi completamente uniforme si se proporciona u obtiene un conjunto de soporte independiente, y todo lo contrario cuando no es así.

Encontrar un soporte independiente de una fórmula $\mathscr{I}$ pequeña se puede determinar a menudo fácilmente a partir del dominio de origen del que se deriva la fórmula en CNF. Sin embargo, el uso de la popular codificación de Tseitin, que se presentará en la sección 3.2.2, para convertir una fórmula a CNF, introduce nuevas variables que forman una apoyo dependiente de la fórmula.

2.3.2. SMARCH

Oh et al. [25] codificaron los modelos como BDDs para optimizar el conteo de configuraciones, generar muestras aleatorias y guiar la búsqueda de configuraciones casi óptimas, y con un mayor grado de aleatoriedad. A esta técnica la llamaron *Counting BDD* (CBDD). Sin embargo, el número de variables se consideró un factor limitante para construir los BDDs.

Posteriormente, en [6], propusieron la herramienta llamada *SMARCH*, que está construida sobre el solucionador #SAT *sharpSAT*, en lugar de CBDD, con el objetivo de mejorar la escalabilidad [26]. Específicamente, con esta herramienta se cuenta el número de soluciones y luego se genera un número aleatorio para seleccionar una de esas soluciones. A continuación, se aplica una búsqueda binaria para obtener la solución en cuestión determinando sucesivamente el valor de cada variable con la ayuda de más llamadas al solucionador. El algoritmo es bastante sencillo, pero no incluye ninguna optimización, por lo que se espera que el rendimiento sea deficiente. La escalabilidad depende de lo que *sharpSAT* pueda manejar.

Cuando esta herramienta fue presentada, se comparó con *UniGen2*, a la cual superó con creces en términos de escalabilidad.

2.3.3. QuickSampler

QuickSampler es otra herramienta de muestreo concebida por Dutra et al. [2]. Funciona generando primero una solución candidata al azar y luego aplicando un solucionador *MAX-SAT*[4] para encontrar una solución similar a la candidata. A partir de ahí, se aplican una serie de mutaciones (es decir, se invierten los valores de las variables) para generar más candidatos en las siguientes llamadas al solucionador. En este caso, esta herramienta priorizó la velocidad de computación frente a la uniformidad o la escalabilidad.

QuickSampler fue diseñado para usarse en pruebas difusas, por lo que no importa si algunos candidatos no son soluciones reales a las restricciones, o son soluciones inválidas. Esta

[4]MAX-SAT trata de encontrar una solución que maximice el número de cláusulas satisfechas. Es decir, que aunque no haya una solución para la fórmula, proporciona una solución aproximada.

herramienta también se beneficia enormemente del uso de un conjunto de soporte independiente. Sin él, se espera que el rendimiento de la herramienta se degrade considerablemente.

Partiendo de una fórmula en CNF, representada en formato DIMACS (presentado en detalle en la sección 3.2.1), e incluyendo una lista de variables formando un soporte independiente, *QuickSampler* fue comparado en su presentación con *UniGen2*, al cual superó en velocidad en casi 5 órdenes de magnitud. Teniendo en cuenta también una posterior validación de las soluciones, *QuickSampler* seguía siendo tres veces más rápido. Sin embargo, no es tan uniforme como *UniGen2*.

2.3.4. SPUR

Achlioptas et al. [4] desarrollaron *SPUR*, siglas de Satisfying Perfectly Uniform Random (Satisfactorio Perfectamente Uniforme Aleatorio), modificando el solucionador *sharpSAT*. *SPUR* realiza una búsqueda de conteo basada en modelos DPLL como *sharpSAT*, capturando componentes para mayor eficiencia y rapidez. Su contribución es aprovechar la alta velocidad del contador de modelos *sharpSAT* en el contexto del muestreo. *sharpSAT* se comporta eligiendo una variable arbitraria y contando el número de modelos que resultan de hacer la variable igual a 0 y a 1, de ahi se establecen sus probabilidades. Como principal mejora sobre este solucionador, *SPUR* almacena soluciones parciales para producir soluciones combinadas de las obtenidas por *sharpSAT* al problema global, aumentando así el número de muestras generadas, una técnica llamada *muestreo de reservorios*.

Sus autores realizaron una comparación con el anteriormente mencionado *UniGen2* en [4], resultando *SPUR* más de 400 veces más rápido y 3 veces más probable de generar con éxito muestras en fórmulas con un gran número de variables (más de 10000).

2.3.5. KUS

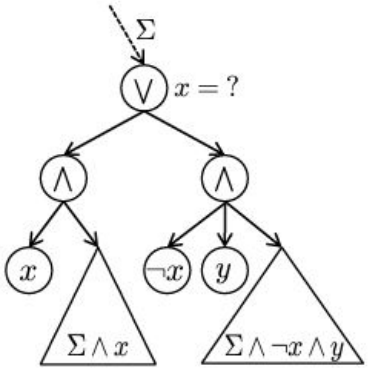
La última herramienta de muestreo de esta revisión, denominada *KUS*, fue presentada por Sharma et al. [3]. *KUS* usa una variante de los OBDDs, llamada *d-DNNF* o Deterministic Decomposable Negation Normal Form (Forma Normal de Negación Descomponible Determinista), que es un superconjunto estricto de BDD, para realizar URS.

Un d-DNNF es un grafo que consta de nodos AND y nodos OR. Los algoritmos de Knuth están generalizados para esta estructura. Los nodos OR representan disyunciones sobre variables disjuntas, por lo que las probabilidades de los hijos se pueden sumar después de algún

ajuste para obtener la probabilidad del padre. Por su lado los nodos AND representan con conjuntos de variables disjuntas, por lo que las probabilidades se pueden calcular multiplicando las probabilidades ajustadas de los hijos.

En la Figura 2.3 puede verse un ejemplo de estos grafos.

FIGURA 2.3: Ejemplo d-DNNF [27]



KUS se basa en d4, un compilador de d-DNNF que transforma la fórmula de CNF a d-DNNF. Tras esta transformación, se anota el d-DNNF, asignándole una tupla que consiste en el número de soluciones y el conjunto de variables en la subfórmula correspondiente al nodo. A continuación, se invoca una subrutina que muestrea y devuelve las muestras resultantes dibujadas de manera uniforme e independiente. Finalmente, KUS da una asignación aleatoria a las variables no asignadas para cada muestra teniendo en cuenta las variables no restringidas que no aparecen en el d-DNNF invocado.

La principal ventaja de este algoritmo *KUS* es que la muestra completa se genera con un único recorrido al grafo. La evaluación de *KUS* mostró que era más rápido que *SPUR* y *UniGen2*. Así, es un algoritmo muy rápido, pero siempre partiendo de que la construcción del d-DNNF sea viable.

Junto a estos algoritmos, en el siguiente capítulo se presentará la propuesta *Genrandom* sometida a análisis en este Trabajo Fin de Máster.

En el Cuadro 2.2 se recogen algunas de las características clave y distintivas de cada uno de los algoritmos de muestreo que se han presentado en el este capítulo, junto con nuestra propuesta, con la que se compararán más en detalle en la sección 3.3.

CUADRO 2.2: Características clave de las herramientas comparadas

Características	**Smarch**	**UniGen3**	**Quick Sampler**	**SPUR**	**KUS**	**Genrandom**
Basadas en grafos					•	•
Basadas en árboles de búsqueda	•	•	•	•		
Soporte independiente		•	•			
Tamaño de entradas limitado	•					
SAT solver usado	sharpSAT	CryptoMiniSAT	MAX-SAT	sharpSAT		

Capítulo 3

Evaluación empírica de Genrandom

Este capítulo constituye la parte central del trabajo, y en él se desarrolla la evaluación empírica de nuestra propuesta, el algoritmo de muestreo aleatorio *Genrandom*.

Como se mencionó en la Introducción 1.4, este capítulo está estructurado en tres secciones principales. En primer lugar, se presenta la metodología adoptada a la hora de llevar a cabo esta evaluación. A continuación, se expone el procesamiento al que han debido ser sometidos los datos de partida. Por último, se muestran en detalle los resultados de la evaluación, así como se analizan y comparan los mismos entre las distintas herramientas.

3.1. Metodología

Antes de entrar en la parte más práctica, en esta sección se presentan las distintas etapas que se han llevado a cabo en este trabajo de investigación a la hora de evaluar el algoritmo propuesto. Cada etapa se presenta en una subsección.

3.1.1. Datos de partida

Los circuitos de referencia originales generalmente no están disponibles, a menudo debido principalmente a problemas de propiedad intelectual. Por lo tanto, se suelen aplicar métodos de ingeniería inversa para intentar recuperar los archivos originales, aunque siempre hay alguna pérdida de información (sobre todo estructural) al codificar un circuito a CNF, que sería más evidente en una representación lógica proposicional completa, y podría resultar de utilidad en su resolución (por ejemplo, reduciendo su tiempo de ejecución). Debido a ello, se han desarrollado muchos métodos diferentes para llevar a cabo esta transformación.

Li [28] presenta un primer esfuerzo por extraer dinámicamente listas de literales equivalentes de tres literales como máximo usando un procedimiento DPLL. Sin embargo, este

enfoque resultaba costoso ya que realiza muchas pruebas sintácticas inútiles y sufre de restricciones, como la longitud de las listas detectadas. Posteriormente, tomando este primer intento como base, intentando extraer información estructural en forma de puertas lógicas de fórmulas ya expresadas en CNF, encontraron coincidencias simples de puertas AND y OR en [29]. Ostrowski et al. propusieron hacer uso de un concepto de gráfico parcial de cláusulas para limitar el número de pruebas sintácticas a realizar, pero sin limitar el número de literales en las puertas.

Roy et al. fueron los primeros que se centraron en identificar la estructura del circuito lógico en una instancia dada del SAT y ofrecen un algoritmo general para detectar todas las ocurrencias de una puerta lógica en esa instancia, utilizando un comparador de gráficos genérico [30].

Más tarde, Fu y Malik [31] no sólo extrajeron puertas lógicas, sino que también garantizaron la extracción del circuito acíclico más grande posible utilizando un solucionador SAT. Este enfoque se basa en una biblioteca de puertas que describe las puertas a extraer, lo que lo hace más flexible pero menos eficiente que la coincidencia de patrones. Basándose en este trabajo, Seltner [32] y Biere desarrollaron un programa llamado *cnf2aig*[1] que puede reconstruir circuitos a partir de CNF y generarlos como gráficos de inversores AND. Su programa consiste en algoritmos para detectar las puertas de hardware más comunes en CNF. También implementaron una solución para el problema parcial de MAX-SAT que garantiza que el circuito reconstruido es máximo con respecto a las puertas que detecta.

3.1.2. Evaluación de los resultados

En la sección 3.3, se llevará a cabo la comparación de las herramientas de muestreo aleatorio *UniGen3*, *SMARCH*, *QuickSampler*, *SPUR* y *KUS*, elegidas por ser consideradas referentes en el cálculo de muestras aleatorias, para evaluar su rapidez al generar las muestras, su escalabilidad y su uniformidad frente a nuestra herramienta propuesta, *Genrandom*. Por un lado, *SMARCH* es útil principalmente como base de la comparación. Por otro lado, *UniGen3* y *QuickSampler* funcionan muy bien cuando se dispone de un conjunto de soporte independiente como datos de partida. Asimismo, *SPUR* mostrará los límites de la búsqueda optimizada de DPLL y se pondrá a prueba la capacidad del compilador d4, del cual depende *KUS*, para obtener los gráficos d-DNNF.

En primer lugar se compararán los dos tipos de grafos usados sobre los modelos, BDD y d-DNNF, que, como se presentó en el Cuadro 2.2, corresponden a los algoritmos *Genrandom*

[1] http://fmv.jku.at/cnf2aig/

y *KUS*, respectivamente. Para ello, se tendrán en cuenta como factores el tamaño resultante de las transformaciones, y el tiempo invertido en generarlas.

En segundo lugar, se llevarán a cabo dos tipos de comparaciones con los seis algoritmos de muestreo a estudio en este Trabajo Fin de Máster. La primera comparación comparará el tiempo que tarda cada una de las herramientas en obtener un número determinado de muestras (mil). La segunda comparación determina un rango de tiempo común (dos horas) y se evaluará el número de muestras que es capaz de generar cada una de las herramientas en ese período de tiempo.

Estas comparaciones y sus resultados se presentarán y analizarán en la sección 3.3.

3.1.3. Discusión sobre los resultados

Tras obtener las tablas comparativas en la sección 3.3, se analizarán los resultados en términos de, principalmente, tiempo de computación y escalabilidad. También se tendrán en cuenta y estudiarán aquellos modelos que hayan podido presentar dificultades.

3.1.4. Conclusiones

Posteriormente en el Capítulo 4 se concluirán las afirmaciones como resultado de este trabajo de investigación.

3.2. Tratamiento de los datos

3.2.1. Conjunto de Datos seleccionado

El *benchmark* o conjunto de datos de referencia utilizados en este trabajo para probar los diferentes muestreadores aleatorios proviene del International Symposium on Circuits and Systems (ISCAS, Simposio Internacional de Circuitos y Sistemas) del IEEE en 1989, y está compuesto por una selección de modelos de la citada conferencia que se muestran, a continuación, en el Cuadro 3.1 con sus correspondientes características.

Estos modelos estaban inicialmente en formato *bench* y necesitaban ser traducidos a CNF. Este formato *bench* es un lenguaje de descripción jerárquica ampliamente utilizado en la descripción de circuitos de referencia, como los correspondientes a las conferencias ISCAS'85, ISCAS'89 e ISCAS'99. Este tipo de lenguaje proporciona un mecanismo para definir un sistema y reutilizar este sistema para construir otro. Los diferentes elementos que incorpora son

CUADRO 3.1: Circuitos de referencia

Modelo	#Variables	#Soportes	#Cláusulas
s344	184	9	429
s499	175	1	491
s635	320	2	762
s938	512	34	1,233
s967	439	16	1,157
s991	603	65	1,337
s1196	561	14	1,538
s1269	624	18	1,616
s1512	866	29	2,044
s3271	1,714	26	4,269
s3330	1,961	40	4,605
s3384	1,911	43	4,440
s4863	2,495	49	6,434
s6669	3,402	83	8,423

el nombre del sistema, los puertos (entradas, salidas o bidireccionales inouts), la función que se realiza sobre cada puerto, y una instanciación de un sistema en otro. Un ejemplo de su contenido se muestra en la Figura 3.1.

3.2.2. Traducción de los circuitos a lógica proposicional

La traducción de circuitos electrónicos a lógica proposicional es una técnica establecida, ya que la mayoría de los muestreadores aleatorios esperan que las entradas estén en forma normal conjuntiva (CNF). No obstante, esta no es una solución ideal para circuitos secuenciales (es decir, circuitos con memoria) porque CNF no presenta estados. La disponibilidad general de herramientas de análisis que utilizan CNF como entrada ha favorecido su adopción como estándar para el muestreo aleatorio de circuitos.

Esta traducción de las puertas lógicas a la lógica proposicional es relativamente sencilla. Sin embargo, se ha necesitado llevar a cabo una serie de etapas previas.

Consideraciones previas

Para el caso particular de este conjunto de circuitos de referencia, la traducción de los elementos flip-flops de tipo D no ha sido inmediata.

Los flip-flops tipo D son elementos electrónicos de circuitos que retrasan el cambio de estado de su señal de salida con respecto a la señal de entrada, hasta la recepción del siguiente flanco ascendente de una segunda señal de entrada de temporización de reloj. Su uso estriba

FIGURA 3.1: Circuito ejemplo en formato *bench*

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)
G15 = OR(G12, G8)
G16 = OR(G3, G8)
G9 = NAND(G16, G15)
G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

en el almacenamiento en circuitos secuenciales. Por ejemplo, si se toma $\mathscr{A} = \text{DFF}(\mathscr{B})$, $\mathscr{A}$ toma su valor de $\mathscr{B}$ *antes* de ser evaluado, lo que, en principio, podría tener cualquier valor. Por lo general, $\mathscr{B}$ tendrá una definición más adelante que corresponde al valor de *después* de la evaluación.

En este sentido, nuestra traducción de este elemento de circuito elegirá $\mathscr{A}$ como variable de entrada, y la definición de $\mathscr{B}$ se traducirá tal cual. De esa forma, el valor del flip flop D antes de la evaluación de $\mathscr{B}$ va a $\mathscr{A}$, y el valor después de la evaluación va a $\mathscr{B}$.

Transformación de Tseitin

La Transformación de Tseitin (aunque puramente llamada Transformación de Tseytin, el apellido de su autor) es la forma más utilizada para transformar circuitos de lógica combinatoria en fórmulas booleanas expresadas en codificación CNF Tseitin [33].

Antes de la aparición de esta transformación, el método generalmente seguido para llevar a cabo esta conversión se basaba en la propiedad distributiva del álgebra elemental ("el producto de la suma es igual a la suma de los productos"), y en las dos Leyes de De Morgan, que conforman una versión de las leyes de la lógica proposicional clásica[2] [34]. Sin embargo, esta práctica devolvía una fórmula en CNF con un tamaño que podía llegar a ser exponencialmente mayor que el de la fórmula original. En cambio, la transformación de Tseitin propuso por primera vez un crecimiento lineal. Además, este método previo mantenía la equivalencia lógica entre las fórmulas, mientras que Tseitin mantiene su satisfactibilidad.

El funcionamiento de la Transformación de Tseitin consiste en agregar una variable auxiliar (conocida como variable Tseitin) para cada subfórmula que no sea un literal, representando la salida de esa puerta. Asimismo, se añade una cláusula final con un solo literal: la variable de salida de la puerta final. Tras ello, la fórmula resultante puede transformarse a CNF.

Sin embargo, incluso cuando la transformación de Tseitin agrega una nueva variable artificial para cada conector en la fórmula y añade una serie de restricciones adicionales, se pierde cierta información sobre la estructura del circuito que podría ser útil y reducir considerablemente el tiempo de ejecución de los solucionadores. Por esta razón, es relevante obtener los datos del circuito original, como se indicó en el Capítulo 2.

Gramática

Las acciones gramaticales y semánticas empleadas para la traducción del formato *bench* a CNF, se pueden ver en el Cuadro 3.2. Se compila una secuencia de variables de entrada, una secuencia con el resto de variables y otra secuencia de expresiones lógicas booleanas. Esta información puede almacenarse en un archivo o traducirse posteriormente a CNF. También se ordenan las variables de tal manera que las variables utilizadas en una definición de puerta preceden a la variable definida.

Las fórmulas en CNF resultantes de esta traducción son expresadas en formato DIMACS, un estándar de texto para fórmulas CNF. Estos archivos pueden empezar con comentarios, siendo estos líneas que comienzan con el carácter *c*. A continuación, aparece una línea expresando *p cnf nbvar nbclauses*, que indica que la instancia está en formato CNF, siendo *nbvar* el número exacto de variables que aparecen en el archivo, y *nbclauses* el número exacto de cláusulas contenidas en el archivo. Luego siguen las cláusulas, donde cada cláusula es una secuencia de números no nulos distintos entre *-nbvar* y *nbvar*, terminando todas las líneas en

[2]Las dos Leyes de De Morgan expresan que:
La negación de la conjunción es la disyunción de las negaciones: $\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$
La negación de la disyunción es la conjunción de las negaciones: $\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$

CUADRO 3.2: Sintaxis del circuito y acciones semánticas asociadas para su traducción

non-terminal	**producción**	**acción semántica**
file	ε	
file	line	
file	file line	
line	INPUT (NAME)	addInput(NAME)
line	OUTPUT (NAME)	addOutput(NAME)
line	output = type (inputs)	addVar(output) addExp(Comment(output) switch(type) case or : addExp(NAME → OR(inputs)) addExp(OR(inputs) → NAME) break; case nor : addExp(NAME → NOT(OR(inputs))) addExp(NOT(OR(inputs)) → NAME) break; case not : addExp(NAME → NOT(inputs)) addExp(NOT(inputs) → NAME) break; case and : addExp(NAME → AND(inputs)) addExp(AND(inputs) → NAME) break; case nand : addExp(NAME → NOT(AND(inputs))) addExp(NOT(AND(inputs))→ NAME) break; case nor : addExp(NAME → NOT(OR(inputs))) addExp(NOT(OR(inputs))→ NAME) break; case dff : addInput(inputs)
inputs	NAME	return NAME)
$inputs_1$	$inputs_2$, NAME	return concat($inputs_2$, NAME)

0. Como condición, una claúsula no puede contener los literales opuestos *i* y *-i* simultáneamente. En dichas cláusulas, los números positivos denotan las variables correspondientes, y los negativos sus negaciones.

Continuando con el mismo circuito que se ha usado como ejemplo en la figura anterior, su conversión a DIMACS CNF resultante a partir del formato *bench* original puede verse en la Figura 3.2 al final de este capítulo. El programa usado para traducir estos circuitos incluye como comentarios al comienzo del archivo una enumeración de todas las variables.

Los archivos en CNF resultantes de esta transformación nos permiten ya usarlos como entradas para los algoritmos de muestreo que se han presentado en el Capítulo 2. sin embargo, para aplicar *Genrandom* esta transformación no es necesaria, aunque sí se debe traducir el formato *bench* a un formato diferente donde se agrupan y expresan las cláusulas en la forma requerida para generar sus BDDs. Este proceso se presenta en la siguiente sección.

3.2.3. Muestreo de circuitos

En la sección 2.3, se ha presentado el Algoritmo de Knuth para anotar los nodos de un BDD 1, el cual se ha aplicado para anotar el BDD generado a partir del circuito ejemplo. Este BDD anotado puede verse en la Figura 3.3, donde el nodo 0 y las rutas que conducen a él han sido omitidas para lograr mayor claridad. Debido a que la representación en BDDs es muy sensible al orden de las variables, en este caso, el reordenamiento heurístico de las variables ha reducido el número de nodos de 145 a 49.

En esa misma sección 2.3 se ha introducido también el algoritmo de muestreo de Knuth para BDDs 2, el cual se aplica sobre el BDD anotado para generar el BDD de estados aleatorios de forma realmente rápida. Sin embargo, el tamaño de un BDD puede crecer exponencialmente con el número de variables, empeorando el rendimiento. Lamentablemente, este escenario, que corresponde al peor de los casos, es comúnmente la realidad.

Para ponerle solución a este problema, se propone utilizar un conjunto o base de BDDs, que es básicamente una secuencia de varios BDDs, con la particularidad de que los nodos pueden ser compartidos entre varios BDDs. La base consiste en crear un BDD para la traducción de cada puerta, al ser los BDD individuales mucho más fáciles de calcular porque generalmente hay unas pocas variables involucradas.

La traducción obtenida en la sección 3.2.2 se utiliza para construir una base BDD. Partiendo de estas traducciones, todas las variables, a excepción de las variables de entrada, tienen una definición, que va precedida de un comentario con el nombre de la variable que se define

con el objeto de marcar el comienzo de la traslación de una puerta. Se construye un BDD para cada definición de variable. Las variables se ordenan de tal manera que las variables de una definición son variables de entrada o ya se han definido. Para ello, se ha desarrollado un programa que transforma el circuito ejemplo en el formato que puede verse, a continuación, en la Figura 3.4.

La Figura 3.5 muestra una base BDD de este mismo circuito de ejemplo anotado con sus probabilidades. Las raíces están coloreadas en naranja y los otros nodos no terminales en cían. Se han omitido las probabilidades en los nodos de sólo lectura, el nodo 0, y las rutas que conducen a él.

Nuestro enfoque de muestreo se basa en el algoritmo URS de Knuth para BDD [16]. Este consiste en una estrategia de dos pasos: anotar los nodos y generar estados aleatorios.

El primer paso es anotar los nodos usando Algoritmo 1. La diferencia es que esta vez se aplica a una base BDD en lugar de a un solo BDD.

En el segundo paso, se generan tantos estados aleatorios como sea necesario. Para eso se usa el Algoritmo 3. Este algoritmo es muy similar al algoritmo GENERATE 2 de Knuth excepto que se realiza de forma secuencial, es decir, una vez para cada BDD. En primer lugar, se asignan valores aleatorios a las entradas (sus BDDs apuntan a 1). A continuación, a cada variable que no es de entrada se le asigna un valor recorriendo su BDD desde la raíz: las variables que aparecen en la definición ya tienen un valor, por lo que los próximos nodos a recorrer se seleccionan de acuerdo a lo dictado por ese valor. Finalmente, a la variable que se está definiendo se le asigna un valor de acuerdo con la probabilidad del nodo. Después de esto, se debe haber alcanzado el nodo 1, otra diferencia con el algoritmo de Knuth, que puede seguir asignando variables.

Siguiendo este Algoritmo 3 y la Figura 3.5, se puede observar que para obtener un valor aleatorio para G14 se asigna el valor opuesto a G0. Después de eso, se puede obtener un valor para G8: si G6 es falso, también lo es G8; de lo contrario, el valor para G8 es el mismo que el valor para G14, y así sucesivamente.

Además, es importante destacar sobre el ejemplo de esta Figura 3.5, que la reducción en el número de nodos es notable incluso en este pequeño ejemplo. Se ha disminuido a 29 nodos en la base BDD partiendo de los 145 del BDD original, sin ordenación heurística de variables.

Algoritmo 3 Muestreo aleatorio de una base BDD

```
1: function SAMPLE(forest, heads)
2: heads es una secuencia de la posición de la variable definida para
3: la definición de cada puerta
4:     state ← secuencia booleana inicializada a falso
5:     for i ← 0 to size(heads) − 1 do
6:         trav ← roots(forest)[i]
7:         pos ← heads[i]
8:                                          ▷ Lectura valores previos de la puerta
9:         while level(trav) < pos do
10:            if state[level(trav)] then
11:                trav ← high(trav)
12:            else
13:                trav ← low(trav)
14:            end if
15:        end while
16:                                          ▷ Valor de la puerta computada
17:        if trav = 1 then                  ▷ Entrada
18:            state ← randomBool()
19:        else                              ▷ NO entrada
20:            if Pr(trav) = 1 then
21:                state[pos] = true
22:                trav ← high(trav)
23:            else
24:                trav ← low(trav)
25:            end if
26:        end if
27:    end for
28:    return state
29: end function
```

Este tipo de muestreo aleatorio siguiendo el algoritmo anterior 3 es **uniforme** gracias a que:

- Las variables de entrada (es decir, el soporte independiente) pueden tener cualquier valor, por lo que se eligen uniformemente.
- El resto de las variables son una función de las variables de entrada, por lo que la probabilidad de un estado es la misma que la probabilidad de elegir las variables de entrada, a saber, $\frac{1}{2^s}$, donde s es el tamaño de la variable de entrada.

Estos argumentos validan la afirmación de que nuestro algoritmo de muestreo *Genrandom* es *uniforme por diseño*.

Otra forma de verificar esta uniformidad de forma empírica se propone en el artículo [7]. Este artículo propone un método que compara información empírica de una muestra con información teórica sobre toda las soluciones que representa el modelo, para justificar la uniformidad de las muestras. En este sentido, esta citada prueba requiere el menor tamaño de muestra de todos los métodos existentes en la literatura, facilitando, así, la evaluación de la uniformidad de los muestreadores incluso cuando los modelos presenten un gran número de variables (como a menudo pueden ser los ICs que tratamos en esta memoria). Usando esta prueba, en la publicación se realizó una evaluación empírica de los muestreadores de última generación que también aquí se han presentado en el Capítulo 2 (aunque tomando *Unigen2* en lugar de *Unigen3*) frente al muestreador de BDDs, revelando el superior desempeño de este último.

3.3. Comparación

Las comparaciones presentadas en esta sección, y el análisis de los resultados de las mismas, se han enfocado principalmente en torno a la respuesta a las siguientes dos cuestiones:

- **C1**: ¿Cómo de factibles y escalables son los diferentes enfoques?
- **C2**: ¿Qué rendimiento puede llegar a lograrse?

3.3.1. Plan experimental

Procesamiento

Los experimentos se realizaron en un servidor HP Proliant Gen9 con dos procesadores Xeon E5-2660v4 de 28 cores cada uno y 224 Gb de memoria. Para administrar los BDD, se usa la biblioteca CUDD[3], que tiene un soporte integrado para las bases de BDD.

Datos de prueba

El Cuadro 3.1 mostraba los circuitos que se ha traducido de la conferencia ISCAS'89, junto con el número de variables y cláusulas. Cabe señalar que aunque la base BDD para *Genrandom* se construyó directamente a partir de la traducción, los circuitos tuvieron que traducirse nuevamente a CNF. Esta conversión no es necesaria para la herramienta *Genrandom*, gracias a estar basada en BDDs, como se ha mencionado anteriormente, sino para el resto de herramientas con las que se comparará los resultados que sí precisan las entradas en esta forma.

Hay una gran variedad de tamaños, lo que proporcionará la variabilidad necesaria para llevar a cabo la comparación con los enfoques competidores, que se espera que muestren un comportamiento muy diferente. También se informa del tamaño del conjunto de soporte porque *Quicksampler* y *Unigen3* dependen de él.

Como se presentó en el Cuadro 2.2, de las seis herramientas a estudio en este Trabajo Fin de Máster, dos de ellas se basaban en grafos, que se compararán en un primer análisis del resultado de sus trasformaciones para los distintos modelos.

3.3.2. Comparación entre representación BDD y d-DNNF

KUS y nuestro enfoque, *Genramdom*, se basan en la construcción de grafos d-DNNF y una base de BDD, respectivamente. El Cuadro 3.3 muestra el número de nodos resultantes al representar cada uno de los modelos en ambos enfoques, y, por otro lado, el Cuadro 3.4 muestra el tiempo necesario para construir cada uno de estos grafos para los distintos modelos.

Atendiendo a los resultados mostrados en el Cuadro 3.3, *Genrandom* demuestra que evita el crecimiento exponencial de los nodos al simplificar el problema, ya que sólo se genera un

[3] `https://github.com/ivmai/cudd`

CUADRO 3.3: Tamaños de los BDDs y d-DNNF

Modelo	**Nodos #BDD**	**Nodos #d-DNNF**
s344	430	281
s499	492	408
s635	763	369
s938	1,234	635
s967	1,158	7,179
s991	1,338	804
s1196	1,539	6,828
s1269	1,617	1,220,102
s1512	2,045	23,032
s3271	4,270	22,379
s3330	4,606	927,096
s3384	4,441	8.978
s4863	6,435	2,227,121
s6669	8,424	74,983,801

CUADRO 3.4: Tiempo invertido en construir las bases BDDs & d-DNNFs (en segundos)

Modelo	**base #BDD**	**d-DNNF**
s344	0.207	0.023
s499	0.208	0.022
s635	0.397	0.051
s938	0.683	0.077
s967	0.588	0.304
s991	0.834	0.197
s1196	0.825	0.828
s1269	0.930	118.375
s1512	1.486	2.068
s3271	3.894	8.920
s3330	4.835	80.418
s3384	4.728	1.255
s4863	7.169	1584.111
s6669	11.474	10,605.473

BDD para cada definición de puerta, derivando en un crecimiento lineal. *KUS*, sin embargo, construye un gráfico de gran tamaño con crecimiento exponencial. Como puede observarse, el grafo d-DNNF generado a partir del circuito de mayor tamaño, *s*6669, ocupa 34 Gigabytes y casi 75 millones de nodos. Mientras tanto, nuestra base BDD solo ocupa 336KBytes de espacio con aproximadamente 8500 nodos. Claramente, *KUS* está muy cerca de los límites de su potencial de escalabilidad, mientras que *Genrandom* escala muy bien.

En términos de tiempo de computación, como se muestra en el Cuadro 3.4, pueden sacarse conclusiones similares. Son necesarias casi 3 horas para construir el grafo d-DNNF para el modelo *s*6669, mientras que *Genrandom* construye su base BDD en algo más de once segundos. Como punto a favor de los grafos d-DNNF, estos resultan más eficientes para los modelos más pequeños (con menos de 600 variables aproximadamente) tanto en número de nodos como en tiempo de generación.

3.3.3. Resultados de la generación de muestras aleatorias

Después de haber comparado las bases de representación gráfica de algunas de las herramientas a estudio, en este apartado se compararán los seis algoritmos de muestreo y su comportamiento a la hora de generar las muestras.

Como se comentó en la sección 3.1.2, se ha intentado llevar a cabo dos tipos de comparaciones entre las distintas herramientas que se detallan a continuación.

Comparación dado un número de muestras objetivo

Continuando con las cuestiones presentadas al principio de esta sección, se busca comparar la escalabilidad y el rendimiento de los enfoques. Para ello, se ha diseñado y realizado un experimento para producir mil muestras de cada uno de los circuitos con los seis algoritmos de muestreo. Se ha elegido realizar esta prueba tomando mil muestras como referencia buscando un compromiso entre un valor lo suficientemente grande como para poder observar diferencias entre las distintas herramientas (incluso pudiendo llegar a diferencias de varios órdenes de magnitud), pero no demasiado alto para que los tiempos de los muestreadores más lentos sigan siendo manejables. Se realizaron también pruebas con valores de 100 y 10000 muestras, pero 1000 representaba este consenso.

Los resultados de esta evaluación se muestran en el Cuadro 3.5.

Comenzando con *SMARCH*, se puede ver que el rendimiento es muy deficiente comparado con el resto de herramientas ya que es el muestreador más lento por varios órdenes

CUADRO 3.5: Tiempo de muestro en segundos para mil muestras

Model	Smarch	Unigen3	Quick sampler	SPUR	KUS	Genrandom
s344	3826.574	0.160	0.353	0.990	0.311	0.065
s499	8571.137	0.010	0.536	0.813	0.362	0.055
s635	22623.871	0.010	2.335	2.499	0.532	0.129
s938	37917.224	1.600	0.797	3.271	0.903	0.135
s967	34159.838	0.630	0.991	3.149	3.027	0.120
s991	41241.301	8.130	0.896	1.960	1.277	0.144
s1196	44089.649	0.420	1.857	3.818	3.897	0.180
s1269	Error	0.580	1.511	873.440	760.700	0.182
s1512	104928.073	1.750	1.393	11.466	16.571	0.278
s3271	181816.511	1.110	2.975	111.652	2831.127	0.465
s3330	Error	1.740	7.783	Error	8630.214	0.584
s3384	361704.470	2.150	3.301	111.119	6.343	0.574
s4863	Error	2.480	453.015	376.705	2477.425	0.741
s6669	Error	7.150	329.365	Error	Error	0.770

de magnitud. Su escalabilidad también es bastante pobre, ya que no consigue generar muestras para cuatro de los sistemas, que además son aquellos de mayor tamaño. El problema es que sharpSAT no consigue contar la mayor cantidad de soluciones de estos modelos, lo que significa que el conteo del modelo DPLL no logra escalar.

Unigen3 y *Quicksampler* muestran muy buen desempeño excepto por los dos circuitos más grandes. Ambos muestreadores dependen en gran medida de conjuntos de soporte independiente y, cuando alcanzan cierto tamaño, el rendimiento comienza a degradarse. Aun así, la escalabilidad es buena en la medida en que todos los circuitos se muestrearon correctamente.

SPUR presenta buenos resultados para los circuitos más pequeños. Sin embargo, su rendimiento empeora considerablemente para los circuitos más grandes, siendo mucho más lento que *Unigen3* y *Quicksampler*. Tampoco completa las muestras para dos de los modelos de tamaño superior, lo que demuestra que su escalabilidad no es tan buena.

KUS revela una clara dependencia del número de nodos del d-DNNF: cuando es pequeño, el sistema es bastante rápido, cuando es grande, el sistema es muy lento. El número de nodos depende de la complejidad del circuito, más que del número de puertas, lo que produce resultados impredecibles. Por esta razón, *KUS* falla al muestrear el circuito *s*6669. El d-DNNF para este modelo ha llegado a generarse, pero es tan grande que generar las muestras lleva demasiado tiempo.

Por último, pero no menos importante, está *Genrandom*. No sólo supera a todos los demás muestreadores en términos de rendimiento, sino que también muestra retrasos mínimos cuando crece el tamaño del circuito, lo que lo convierte también en el enfoque más escalable. Concretamente, se tarda menos de un segundo en muestrear cada uno de los sistemas. Aunque atendiendo a los resultados de este cuadro *Genrandom* sólo supera a *Unigen3* y *Quicksampler* en un orden de magnitud para los circuitos más grandes, vale la pena mencionar que estos dos muestreadores por defecto sólo muestrean las variables de los soportes independientes, lo que significa que calcular el valor del resto de las variables probablemente llevaría bastante más tiempo.

Comparación dado un período de tiempo determinado

La finalidad de esta prueba era cambiar la perspectiva de la comparación previa y poner un límite temporal, pero sin limitar el número de muestras generadas, con el objetivo de verificar si alguno de los algoritmos empeorase su rendimiento con el tiempo, y la dependencia temporal no fuera lineal, y comprobar el consumo de memoria que cada herramienta presentaba en un mismo rango temporal.

Sin embargo, esta prueba no ha resultado interesante debido a que los algoritmos *Quick sampler*, *KUS* y *Genrandom* calculan en primer lugar todas las muestras, y cuando han alcanzado el número requerido, se escriben. Por ello, no pueden ejecutarse de la manera aquí planteada.

Además de la inicial limitación de estos tres algoritmos, la disparidad en tiempos de generación, que ya ha sido observada en la sección anterior, hace complicada la elección de un tiempo límite adecuado y relevante para todos ellos. Mientras los muestreadores más lentos necesitaban largos períodos de tiempo para que el número de muestras generadas fuese considerable, los muestreadores más rápidos generaban demasiadas muestras en estos períodos, llegando a ocupar todo nuestro espacio libre en disco.

Cuestiones

Como respuesta a **C1**, se puede afirmar que, salvo *Smarch* (ya que necesita más de dos horas para muestrear el circuito más pequeño de 175 variables), todos los enfoques son factibles para circuitos de pequeño a mediano tamaño. Sin embargo, para circuitos grandes, sólo se consideran aplicables *Quicksampler* y *Genrandom*. Teniendo en cuenta los circuitos de referencia usados en este Trabajo Fin de Máster, dados los resultados podría establecerse el umbral en 1900 variables aproximadamente.

En términos de escalabilidad, el sistema más escalable es *Genrandom*, seguido de *Unigen3*, *Quicksampler*, *KUS*, *SPUR* y *SMARCH*, respectivamente.

La respuesta a **C2** es que *Genrandom* mostró el mejor rendimiento, seguido de *Unigen3*, lo que le da un buen uso al conjunto de soporte independiente. El resto de los sistemas se degradan rápidamente con el tamaño de las variables de entrada.

FIGURA 3.2: Circuito ejemplo en DIMACS CNF

```
c ind 1 2 3 4 0
p cnf 17 28
c     1 G0
c     2 G1
c     3 G2
c     4 G3
c     5 G5
c     6 G6
c     7 G7
c     8 G14
c     9 G17
c    10 G8
c    11 G15
c    12 G16
c    13 G9
c    14 G10
c    15 G11
c    16 G12
c    17 G13
-8 -1 0
1 8 0
-10 8 0
-10 6 0
-8 -6 10 0
-12 4 10 0
-4 12 0
-10 12 0
-16 -2 0
-16 -7 0
2 7 16 0
-17 -3 0
-17 -16 0
16 3 17 0
-11 16 10 0
-16 11 0
-10 11 0
-13 -11 -12 0
12 13 0
11 13 0
-15 -5 0
-15 -13 0
5 13 15 0
-9 -15 0
15 9 0
-14 -8 0
-14 -15 0
15 8 14 0
```

FIGURA 3.3: Ejemplo BDD con probabilidades asociadas a sus nodos

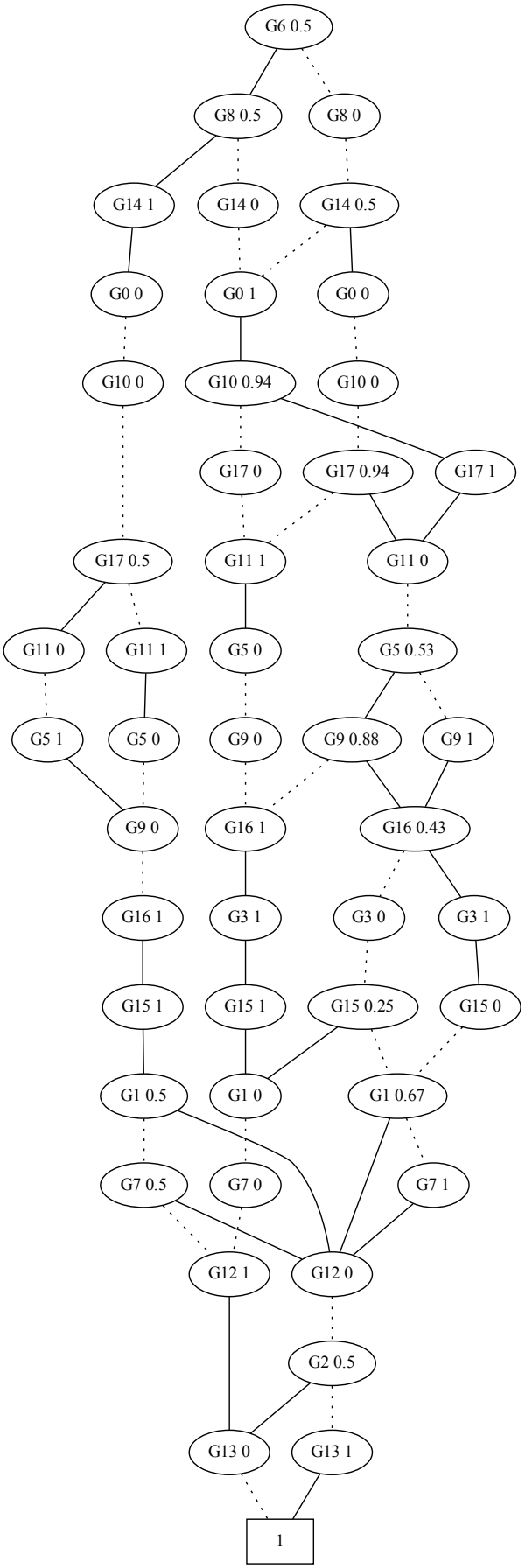
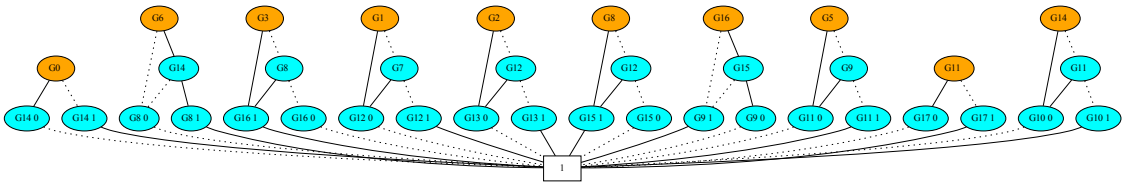


FIGURA 3.4: Circuito ejemplo en formato expresivo

```
# G0
# G5
# G6
# G7
# G14
not G14 or not G0
G0 or G14
# G8
not G8 or G14
not G8 or G6
not G14 or not G6 or G8
# G16
not G16 or G3 or G8
not G3 or G16
not G8 or G16
# G12
not G12 or not G1
not G12 or not G7
G1 or G7 or G12
# G13
not G13 or not G2
not G13 or not G12
G12 or G2 or G13
# G15
not G15 or G12 or G8
not G12 or G15
not G8 or G15
# G9
not G9 or not G15 or not G16
G16 or G9
G15 or G9
# G11
not G11 or not G5
not G11 or not G9
G5 or G9 or G11
# G17
not G17 or not G11
G11 or G17
# G10
not G10 or not G14
not G10 or not G11
G11 or G14 or G10
```

FIGURA 3.5: Base BDD anotada del circuito ejemplo



Capítulo 4

Conclusiones

En este Trabajo Fin de Máster, se ha presentado una forma novedosa de basar un algoritmo de muestreo aleatorio en el uso de una base BDD, y se ha propuesto su aplicación sobre traducciones de circuitos integrados a lógica booleana, donde las muestras resultantes tendrían una aplicación directa en técnicas de pruebas difusas (*fuzz* testing). A su vez, este tipo de pruebas constituye un método muy usado en las fases de verificación dentro del proceso de diseño de hardware.

La herramienta aquí propuesta, *Genrandom*, presenta un diseño muy uniforme y no ha mostrado ninguna penalización en su rendimiento al aumentar el número de variables de entradas en los modelos usados como base de pruebas, evidenciando su idónea escalabilidad, y demostrando linealidad frente al aumento exponencial del tiempo de generación que han presentado los enfoques competidores.

Por un lado, una colección de 14 circuitos de la conferencia ISCAS'89 se tradujo a lógica booleana para usarse como punto de referencia. Por otro lado, cinco herramientas de muestreo aleatorio de gran relevancia en la actualidad y basadas en algoritmos de naturaleza diversa se presentan como contrapartida: *Smarch, Unigen3, Quick sampler, SPUR* y *KUS*.

Generalmente, los algoritmos de muestreo aleatorio pueden basarse en árboles de búsqueda o en grafos. Dentro de este último grupo se clasifica nuestra herramienta *Genramdom* al estar basada en BDDs, al igual que la herramienta KUS, que a su vez se basa en grafos d-DNNF. En una primera comparación se han equiparado ambas representaciones para los modelos de referencia, donde la base BDD ha demostrado ser una representación naturalmente más escalable al necesitar menos nodos para los modelos de mayor número de variables.

Posteriormente, *Genrandom* se ha comparado frente a las cinco herramientas en un intento de generar 1000 muestras aleatorias de cada uno de los modelos de prueba. Como resultado, se evidencia que nuestra propuesta constituye la herramienta más rápida, superando a la segunda mejor (*Unigen3*) en un orden de magnitud (y muestreando *Genrandom* la totalidad de las

variables y no sólo las del soporte independiente). Además, nuestro algoritmo puede realizar muestras incluso del circuito más grande, *s*6669, con facilidad, mientras otras herramientas no lo han logrado. Se concluye así que nuestro enfoque es el mejor entre las opciones evaluadas en términos de rendimiento y escalabilidad.

Por último, nos gustaría destacar que la evaluación experimental descrita en este trabajo ha dado lugar al siguiente artículo:

Elena Pinilla Sediles, David Fernandez-Amoros, Ruben Heradio. Circuit Testing Based on Fuzzy Sampling with BDD Bases. Hawaii International Conference on System Sciences (HICSS), 2023.

Dicho artículo ha superado un proceso de revisión "por pares anónimo" y será presentado en el congreso Internacional de Ciencias de Sistemas de Hawaii ((HICSS 56)), que se considera como de Clase 2 según el ranking GII-GRIN-SCIE (GGS) (http://scie.lcc.uma.es/gii-grin-scie-rating/).

Capítulo 5

Trabajo Futuro

Por un lado, atendiendo a la verificación de hardware, esta se enfrenta a desafíos adicionales con la ampliación y el crecimiento progresivo del tamaño y el alcance del diseño de hardware. Nuevamente, es la escalabilidad el mayor desafío en la verificación moderna de circuitos debido a la complejidad y el tamaño masivo de sus diseños.

Para abordar la escalabilidad, se requiere una plataforma de verificación automatizada y sistemática. Sin embargo, la automatización en la verificación de circuitos es muy difícil por varias razones. En primer lugar, el ingeniero de verificación se enfrenta a desafíos de gran envergadura para evaluar con precisión las políticas de seguridad de un diseño de hardware debido a la enorme cantidad de componentes que interactúan entre sí. En segundo lugar, también hace frente a grandes dificultades a la hora de modelar los escenarios de ataque que pueden ocurrir en el circuito, donde estos ataques pueden incluir ataques directos de hardware o software. Por lo tanto, la verificación de circuitos electrónicos sigue siendo una tarea costosa que exige una investigación significativa para desarrollar una solución robusta y escalable.

En general, el *fuzzing* puede ser una solución prometedora como mecanismo de extremo a extremo para la verificación completa del sistema ya que requiere una cantidad mínima de ajustes. Sin embargo, siguen sin poder cubrir una amplia variedad de modelos de amenazas y la identificación de la fuente de las vulnerabilidades detectadas, lo que demuestra que es inevitable una mayor investigación en este dominio. Debe tenerse en cuenta también que el modelo de amenazas de los diseños de hardware es realmente diferente al de software.

Un problema aún no abordado adecuadamente se debe al hecho de que muchos estudios recientes se enfocan en construir un enfoque general que funcione para todas las posibles vulnerabilidades en el diseño. Sin embargo, las métricas pueden ser más eficientes para detectar vulnerabilidades de hardware si considera el alcance de la vulnerabilidad objetivo en cada sesión de prueba. Por ejemplo, no es necesaria una exploración completa del sistema mientras se generan muchos casos de prueba para garantizar la seguridad total del sistema cuando la

interfaz de memoria es la única entidad no confiable en el sistema. Esto también puede ayudar con la escalabilidad del enfoque de *fuzzing*.

Por otro lado, tratando la propuesta presentada en este Trabajo Fin de Máster, quedan algunos puntos abiertos para continuar con la investigación.

En primer lugar, en la sección 3.2.3 se mencionó el novedoso método presentado en la publicación [7] que podría resultar interesante aplicar sobre las muestras generadas sobre los circuitos. Como se comentó, esta prueba precisa de una muestra de reducido tamaño, factor muy favorable en el campo de los ICs. Asimismo, podría añadir valor adicional al algoritmo de muestreo *Genrandom* ya que se ha demostrado su sobresaliente uniformidad al evaluarlo con el citado método.

En segundo lugar, la técnica de ingeniería inversa *cnf2aig* de [32] presentada en la sección 3.1.1 puede llegar a ofrecer alternativas al problema común de no disponer de los circuitos de referencia. Esta herramienta podría conferir la posibilidad de poder trabajar sobre los circuitos en formato CNF (más frecuentes), y tras su transformación, permitir la generación de grafos BDD a partir de ellos. No obstante, su funcionamiento debería verificarse. La preparación de los datos requerida para este algoritmo ya se ha realizado en este trabajo.

Bibliografía

[1] Ari Takanen. «Fuzzing: the Past, the Present and the Future». En: *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. 2009, págs. 202-212.

[2] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach y Koushik Sen. «Efficient Sampling of SAT Solutions for Testing». En: *40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden, 2018, págs. 549-559.

[3] Shubham Sharma, Rahul Gupta, Subhajit Roy y Kuldeep S. Meel. «Knowledge Compilation meets Uniform Sampling». En: *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Awassa, Ethiopia, 2018, págs. 620-636.

[4] Dimitris Achlioptas, Zayd S. Hammoudeh y Panos Theodoropoulos. «Fast Sampling of Perfectly Uniform Satisfying Assignments». En: *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Oxford, UK, 2018, págs. 135-147.

[5] Sourav Chakraborty y Kuldeep S. Meel. «On testing of Uniform Samplers». En: *33rd Conference on Artificial Intelligence (AAAI)*. Honolulu, Hawaii, USA, ene. de 2019, págs. 7777-7784.

[6] Jeho Oh, Don S. Batory, Marijn J. H. Heule, Margaret Myers y Paul Gazzillo. *Uniform Sampling from Kconfig Feature Models*. Inf. téc. TR-19-02. Department of Computer Science. The University of Texas at Austin, 2019.

[7] Ruben Heradio, David Fernandez-Amoros, JoséA. Galindo, David Benavides y Don Batory. «Uniform and scalable sampling of highly configurable systems». En: *Empirical Software Engineering* 27.2 (2022), pág. 44.

[8] Gordon E. Moore. «Cramming more components onto integrated circuits». En: *Electronics* 38.8 (abr. de 1965).

[9] Dan Luo, Tun Li, Liqian Chen, Hongji Zou y Mingchuan Shi. «Grammar-based fuzz testing for microprocessor RTL design». En: *Integration* 86 (2022), págs. 64-73. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2022.05.001. URL: https://www.sciencedirect.com/science/article/pii/S0167926022000529.

[10] Barton P. Miller, Lars Fredriksen y Bryan So. «An Empirical Study of the Reliability of UNIX Utilities». En: *Commun. ACM* 33.12 (dic. de 1990), págs. 32-44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[11] K. Z. Azar, M. M. Hossain, A. Vafaei, H. Al Sheikh, N. Mondol, F. Rahman, M. Tehranipoor y F. Farahmandi. «Fuzz, Penetration, and AI Testing for SoC Security Verification: Challenges and Solutions». En: *IACR Cryptology ePrint Archive* (mar. de 2022). URL: https://eprint.iacr.org/2022/394.pdf.

[12] *Fuzzing | OWASP Foundation*. Consultado el 20 de agosto de 2022. URL: https://owasp.org/www-community/Fuzzing#.

[13] Martin Davis, George Logemann y Donald Loveland. «A Machine Program for Theorem-Proving». En: *Commun. ACM* 5.7 (jul. de 1962), págs. 394-397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[14] C. Y. Lee. «Representation of switching circuits by binary-decision programs». En: *The Bell System Technical Journal* 38.4 (1959), págs. 985-999. DOI: 10.1002/j.1538-7305.1959.tb01585.x.

[15] Akers. «Binary Decision Diagrams». En: *IEEE Transactions on Computers* C-27.6 (1978), págs. 509-516. DOI: 10.1109/TC.1978.1675141.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[17] Randal E. Bryant. «Graph-Based Algorithms for Boolean Function Manipulation». En: *IEEE Transactions on Computers* C-35.8 (1986), págs. 677-691.

[18] Martin Davis e Hilary Putnam. «A Computing Procedure for Quantification Theory». En: *J. ACM* 7.3 (jul. de 1960), págs. 201-215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[19] Dipartimento di Ingegneria informatica, automatica e gestionale. *DPLL algorithm*. URL: http://www.diag.uniroma1.it//~liberato/ar/dpll/dpll.html.

[20] Supratik Chakraborty, Kuldeep S. Meel y Moshe Y. Vardi. «Balancing Scalability and Uniformity in SAT Witness Generator». En: *51st Annual Design Automation Conference (DAC)*. San Francisco, CA, USA, 2014, págs. 1-6.

[21] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia y Moshe Y. Vardi. «On Parallel Scalable Uniform SAT Witness Generation». En: *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. London, UK, 2015, págs. 304-319.

[22] Supratik Chakraborty, Kuldeep S. Meel y Moshe Y. Vardi. «A Scalable and Nearly Uniform Generator of SAT Witnesses». En: *25th International Conference on Computer Aided Verification (CAV)*. Saint Petersburg, Russia, 2013, págs. 608-623.

[23] Stefano Ermon, Carla P Gomes, Ashish Sabharwal y Bart Selman. «Embed and Project: Discrete Sampling with Universal Hashing». En: *Advances in Neural Information Processing Systems*. Ed. por C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani y K.Q. Weinberger. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/6d70cb65d15211726dcce4c0e971e21c-Paper.pdf.

[24] Mate Soos, Stephan Gocht y Kuldeep S. Meel. «Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling». En: *Computer Aided Verification*. Springer International Publishing, 2020, págs. 463-484. DOI: 10.1007/978-3-030-53288-8_22. URL: https://doi.org/10.1007%2F978-3-030-53288-8_22.

[25] Jeho Oh, Don Batory, Margaret Myers y Norbert Siegmund. «Finding Near-optimal Configurations in Product Lines by Random Sampling». En: *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany, 2017, págs. 61-71.

[26] Marc Thurley. «sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP». En: *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Seattle, WA, USA, 2006, págs. 424-429.

[27] Christian Muise, Sheila A. McIlraith, J. Christopher Beck y Eric I. Hsu. «Fast d-DNNF Compilation with sharpSAT». En: *Abstraction, Reformulation, and Approximation*. 2010.

[28] Chu Min Li. «Integrating Equivalency Reasoning into Davis-Putnam Procedure». En: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 2000, págs. 291-296. ISBN: 0262511126.

[29] Richard Ostrowski, Eric Gregoire, Bertrand Mazure y Lakhdar Sais. «Recovering and Exploiting Structural Knowledge from CNF Formulas». En: vol. 2470. Sep. de 2002, págs. 185-199. ISBN: 978-3-540-44120-5. DOI: 10.1007/3-540-46135-3_13.

[30] Jarrod Roy, Igor Markov y Valeria Bertacco. «Restoring Circuit Structure from SAT Instances». En: *ACM/IEEE Intl. Workshop on Logic and Synthesis, Temecula, CA*. 2004, págs. 361-368.

[31] Zhaohui Fu y Sharad Malik. «Extracting Logic Circuit Structure from Conjunctive Normal Form Descriptions». En: *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. 2007, págs. 37-42. DOI: 10.1109/VLSID.2007.81.

[32] Harald Seltner. «Extracting Hardware Circuits from CNF Formulas». Tesis de mtría. Johannes Kepler Universität Linz, 2014.

[33] Gregory S. Tseitin. «On the Complexity of Derivation in Propositional Calculus». En: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer Berlin Heidelberg, 1983, págs. 466-483.

[34] Christopher Clapham y James Nicholson. *The Concise Oxford Dictionary of Mathematics*. Oxford University Press, 2009. ISBN: 9780191727122. DOI: 10.1093/acref/9780199235940.001.0001. URL: https://www.oxfordreference.com/view/10.1093/acref/9780199235940.001.0001/acref-9780199235940.