Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Trabajo Final de Máster
(31105151)

Soporte automático para el testing combinatorio de sistemas software

Autor:
Juan Aymerich Vidal

Tutor académico:
Rubén Heradio Gil

Fecha de lectura: Octubre de 2020
Curso académico 2019/2020

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Trabajo Final de Máster
(31105151)

Soporte automático para el testing combinatorio de sistemas software

Tipo A: Trabajo específico propuesto por un profesor

Autor:
Juan Aymerich Vidal

Tutor académico:
Rubén Heradio Gil

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

Fecha: 10/08/2020.

Quién suscribe:

Autor(a): Juan Aymerich Vidal
D.N.I/N.I.E/Pasaporte.: 20467110P

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

Título completo del trabajo.

Soporte automático para el testing combinatorio de sistemas software

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

Resumen

En este documento se presenta la memoria del Trabajo Fin de Máster (TFM). El objetivo del proyecto es comparar la efectividad de un diagrama de decisiones binario frente a un SAT-Solver aplicado a la generación de pruebas combinatorias con imposición de restricciones. Para obtener unos resultados empíricos y analizarlos, se desarrolla una implementación propia desde cero.

Para poder llegar a una conclusión demostrable, se realiza una síntesis del campo de la generación de pruebas combinatorias, y se presenta el algoritmo combinacional In-Parameter-Order-General. Se ha demostrado que este algoritmo es capaz de resolver el problema imponiendo restricciones entre los parámetros de entrada. El problema de las restricciones se aborda como el problema de satisfacción booleana. Frente a la clásica implementación basada en llamar repetidamente a un SAT-Solver, este trabajo propone el uso de Diagramas de Decisión Binarios.

Finalmente se exponen los resultados obtenidos con la idea planteada. Se comparan las dos iteraciones que han sido necesarias para cumplir con el objetivo. La comparativa entre iteraciones demuestra la efectividad que supone la segunda iteración. Tras analizar estos datos, se pasa a comprobar el objetivo principal del proyecto. Esta segunda comparativa muestra unos resultados positivos, pero también hace ver que es necesario seguir investigando para lograr un salto significativo en este campo.

Palabras clave

Pruebas combinatorias, In-Parameter-Order, problema de Satisfacción booleana, Diagrama de decisiones binario.

Keywords

Combinatorial testing, In-Parameter-Order, Boolean satisfiability problem, Binary decision diagram.

Índice general

Índice de figuras

Índices de tablas

Capítulo 1

Introducción

En este capítulo se presenta el problema que se desea abordar. Además, se describen los objetivos que se desean cumplir durante la elaboración del proyecto. Y finalmente se explica cómo está estructurada esta memoria.

1.1. Contexto del proyecto

Desde la llegada de los sistemas informáticos y de comunicación con software integrado, el problema de probar los sistemas de software y hardware se ha vuelto un tema de gran interés. Un estudio elaborado en 2002 por el National Institute of Standards and Technology (NIST), expuso que los fallos software cuestan a la economía estadounidense 59,9 mil millones de dólares anuales [2]. El informe continúa estimando que se podrían ahorrar 22 mil millones de dólares mediante una detección de errores más temprana y efectiva. El coste de corregir errores aumenta exponencialmente a medida que un fallo pasa de la fase de desarrollo a la fase de producción. Por tanto, la detección temprana de errores es algo fundamental.

En muchas ocasiones, cuando se quiere probar un sistema, se crea una batería de pruebas exhaustiva. Es decir, que se acaba generando todas las combinaciones posibles de todos los valores disponibles de cada parámetro. Con esto nos aseguramos de conseguir una cobertura del 100 % de nuestras pruebas. Pero existen casos en los que realizar todas las pruebas es algo inviable por los diferentes recursos materiales, temporales y económicos. Cuando nos encontramos en un caso similar, hay que buscar una forma de priorizar unas pruebas sobre otras.

Un ejemplo de este tipo de casos es cuando tenemos un sistema con 10 parámetros, y cada parámetro tiene 10 valores posibles. Si se quisiera hacer unas pruebas exhaustivas y probar toda la casuística (100 % de cobertura), se tendría que probar todas las combinaciones, que serían un total de 10^{10} pruebas. Esto es algo que se vuelve inmanejable. Probablemente, muchos de estos resultados sean iguales, es decir, que los casos de prueba serían redundantes, no aportarían valor a las pruebas. Y a menudo existen limitaciones en la combinación de valores de parámetros, y es debido a que determinadas combinaciones de valores de parámetros no son válidas. Para ello se definen una serie de restricciones antes de generar el conjunto de pruebas.

Existen diferentes técnicas para intentar reducir el conjunto de pruebas. Una técnica que nos permite ceñirnos a las restricciones de los recursos, es la generación de pruebas aleatorias. Esta técnica consiste en combinar valores de los parámetros de forma aleatoria. Una técnica muy sencilla y que posee una gran ventaja, y es que se puede generar el número de pruebas que se desea. Pero el problema de esas pruebas es que puede que cubran un alto porcentaje de cobertura o, todo lo contrario. No todas las pruebas tienen la misma calidad. Por tanto, no se puede saber cómo de seguro es el sistema.

Otra técnica muy conocida es la partición de dominios. Dicha técnica, en lugar de probar cada valor posible de cada variable posible, el evaluador divide la prueba o las condiciones de prueba en diferentes conjuntos en los que cada miembro de cada conjunto es más o menos equivalente a cualquier otro miembro del mismo conjunto con el fin de descubrir los errores. Estas son las llamadas clases de equivalencia. Por ejemplo, si se desean probar impresoras, en lugar de probar cada modelo, el evaluador puede tratar todas las impresoras que tengan unas características similares (misma marca, inyección de tinta, etc.) como casi equivalentes. Esta técnica puede ahorrarnos una gran cantidad de tiempo y recursos sin arriesgar demasiado en la cobertura de las pruebas. Pero esto solo es posible siempre que podamos decir qué es equivalente a qué. Esto resulta bastante difícil en muchos casos. Y si no se hace correctamente, la cobertura será mucho menor de lo que se pueda esperar.

Una de las técnicas más empleadas para enfrentarse al problema de reducir el conjunto de pruebas, son las pruebas combinatorias. Esto es debido a que esta técnica genera pequeños conjuntos de pruebas que son relativamente fáciles de administrar y ejecutar. Estos conjuntos reducidos son una reducción significativa en el número de casos de prueba exhaustivos y no llegan a comprometer drásticamente la cobertura funcional. Por tanto, consigue reducir los costes y mejorar la efectividad de las pruebas en muchas aplicaciones. La percepción de esta eficiencia radica en la idea de que no todos los parámetros contribuyen a la detección de los fallos, como si ocurre con las pruebas exhaustivas, si no que los fallos son detectados por la interacción entre pocos parámetros.

Gracias a los avances de los últimos años, se han desarrollado nuevos algoritmos de combinación, y que, junto a los procesadores rápidos y económicos actuales, están haciendo que las pruebas combinatorias sean un enfoque práctico y que mantengan la idea de realizar pruebas a un costo menor. Gracias a esta tendencia, se ha seguido investigando para poder conseguir sortear las combinaciones entre valores que no son válidas. Para ello se definen una serie de restricciones que hay que procesar, para posteriormente evaluar las combinaciones entre valores. Las soluciones más comunes que existen hasta ahora consiguen resolver este problema. Pero dadas las limitaciones que presentan, intentar resolver este problema en el ámbito de generación de pruebas mediante un diagrama de decisiones binario, puede ser más efectivo.

1.2. Objetivo

La idea que se desea alcanzar es implementar un algoritmo de generación de pruebas combinatorias ya existente. El algoritmo escogido es In-Parameter-Order, más concretamente su variante In-Parameter-Order-General. Después se implementará un diagrama de decisiones binario para poder interpretar las restricciones, y posteriormente integrar este algoritmo con el

algoritmo combinacional de pruebas.

La integración del diagrama de decisiones binario con el algoritmo combinacional se realiza con la intención de obtener unos resultados. Y posteriormente compáralos con una implementación similar ya existente, pero con un algoritmo que procesa las restricciones de forma distinta. Dicha implementación es ACTS, que hace uso de SAT-Solvers para procesar las restricciones. Dado que es el mismo algoritmo combinacional, las dos implementaciones generarán las mismas pruebas, ya que el algoritmo es determinista. Por tanto, la comparativa consistirá en ver cuál de las dos implementaciones es más eficiente en tiempo de ejecución.

1.3. Contenido del proyecto

Con el fin de explicar cómo se ha llegado a obtener los resultados, esta memoria se divide en capítulos que abordan los distintos conceptos que forman el proyecto:

- Estado del arte: se exponen los antecedentes que existen, las ideas actuales y los desarrollos relacionados en los que se basa el proyecto.
- In-Parammeter-Order: explicación del algoritmo de combinación de pruebas.
- Problema de satisfacción booleana: explica en que consiste este problema y como se emplea para evaluar las restricciones que existen entre los valores de las pruebas. En el capítulo también se aclara como se aplica en el algoritmo de combinación de pruebas.
- Implementación: expone todo el desarrollo práctico del proyecto.
- Resultados: expone los resultados obtenidos con la implementación y se comparan con la herramienta ACTS.
- Conclusiones y trabajo futuro: finalmente, se expresan las conclusiones finales obtenidas a partir de los resultados, así como las líneas de trabajo futuro que se pueden seguir.

Capítulo 2

Estado del arte

En este capítulo se resume brevemente la evolución de las pruebas combinatorias y el algoritmo In-Parameter-Order. Se expone cuales son los últimos enfoques e ideas a los que han conseguido llegar los investigadores en este campo.

2.1. Pruebas combinatorias

Dado el problema que se desea abordar, las pruebas combinatorias es un buen enfoque para intentar resolverlo. Este enfoque no es algo nuevo, pero ha ido despertando cada vez más interés por el gran beneficio que ofrece. Actualmente podemos encontrar ya varías soluciones que implementan la idea de pruebas combinatorias. En el apéndice A se puede consultar una lista muy completa de muchas herramientas que existen actualmente. La gran mayoría están pensadas para resolver problemas por pares o 2-way.

El interés por las pruebas combinatorias surgió hace tiempo con amplios estudios empíricos. En ellas se detectó que muchos fallos se desencadenan cuando se produce una interacción combinacional de más de dos parámetros. Un estudio de dispositivos médicos encontró un caso en el que un fallo involucró una interacción de 4-way en los valores de los diferentes parámetros [3]. En posteriores investigaciones se encontraron con unas condiciones similares en la detección de fallos [4, 5, 6, 7]. A raíz de estos estudios, se demuestra que, las pruebas combinatorias con interacciones de un grado 4-way o superior son necesarias para alcanzar un mayor nivel de seguridad por ser una forma muy efectiva de detectar fallos.

Los primeros algoritmos evolucionan del campo estadístico del diseño de experimentos (en inglés Design of Experiments, DoE). DoE es una metodología estadística para la realización de experimentos controlados que permiten el cálculo de información estadística útil sobre la relación entre los factores de entrada y una variable de respuesta. El objetivo de usar DoE es mejorar el rendimiento de los sistemas, y normalmente se emplea para sistemas como la producción agrícola, los tratamientos médicos y la fabricación industrial que están sujetos a variables no controladas. DoE se basa en los trabajos de Fisher y Yates [8] de los años 20 para tratamientos agrícolas. Más tarde, en las décadas de los 40 y 50, los DoE se adaptaron para

mejorar el rendimiento de los procesos industriales y el desarrollo farmacéutico. Taguchi investigó y adaptó los métodos DoE para desarrollar productos y procesos de fabricación robustos. En sus adaptaciones usó matrices ortogonales [9], lo cual supuso un salto. En la década de los 80, con la llegada de los sistemas informáticos y de comunicación con software integrado, el problema de probar los sistemas de software y hardware se volvió importante y de gran interés. Los investigadores que conocían DoE comenzaron a investigar el uso de matrices ortogonales para probar sistemas software y hardware. En 1993, Sloan [10] definió formalmente el concepto matemático de matrices de cobertura. Y un año más tarde, Sherwood [11] desarrollo la idea de generar conjuntos de pruebas por pares que cubrían todos los pares válidos. Pero no fue hasta 1998 cuando Dalal y Mallows [12] consiguieron relacionar las matrices de cobertura con las pruebas por pares. Observaron que los DoE basados en matrices ortogonales permiten evaluar el efecto principal de cada parámetro. En las pruebas por pares, no es necesario evaluar los efectos principales de los parámetros, pero el interés que despertó DoE radica en cubrir todos los pares para determinar si el software responde correctamente a los parámetros de entrada. Por lo tanto, Dalal y Mallow llegaron a la conclusión que las matrices de cobertura, en lugar de matrices ortogonales, son más adecuadas para las pruebas por pares, dando paso a los primeros algoritmos.

Han surgido muchos algoritmos e ideas para las pruebas combinatorias. Inicialmente, resolver el problema con pares resultó ser relativamente fácil comparado con interacciones más complejas (3-way o superior). Pero los algoritmos más modernos que están dando buenos resultados, se siguen basado en un concepto matemático fundamental, la matriz de cobertura. Las matrices de cobertura permiten saber que todas las combinaciones de valores de parámetros de t-way se cubren al menos una vez [12, 13, 14]. Estos algoritmos generan las matrices de una forma más rápida que métodos anteriores, lo que hace que las matrices de cobertura de 6-way sean manejables para muchas aplicaciones.

Todas las estrategias de combinación intentan obtener una batería de pruebas reducida, pero el porcentaje de cobertura alcanzado varía. Los algoritmos que se basan en el criterio 1-way son los que obtienen las baterías de pruebas más reducidas, pero que también se obtienen una cobertura menor. Con este criterio se suele encontrar el 67 % de fallos. Los algoritmos basados en pares obtienen una cobertura mayor que la anterior, pero por lo general no aumenta mucho la batería de pruebas. El porcentaje de fallos detectados suele ser 93 %, un aumento muy significativo. Hay algoritmos que permiten trabajar con combinaciones de mayor tamaño aparte de los pares. Estos son los algoritmos t-way, que van consiguiendo mayor cobertura a medida que se aumenta el t-way (y también el tamaño de la batería de pruebas). Estos últimos algoritmos permiten modificar el tamaño del t-way en función de las exigencias. Este tamaño se puede aumentar hasta llegar al número máximo de parámetros, pero llega un punto que deja de ser rentable. De 4-way a 6-way ya se detecta el 100 % de los fallos. Estos porcentajes han sido estudiados y demostrados en [1], y varios estudios de otros investigadores [7, 15, 16] han llegado a las mismas conclusiones. El gráfico 2.1 (extraído de [1]) muestra la detección de fallos en varios casos a medida que se aumenta el t-way. Se puede observar como en los cuatros casos ya se detectan el 100 % de fallos con pruebas 6-way, aunque en dos casos ya se detectan el 100 % de fallos con pruebas 4-way.

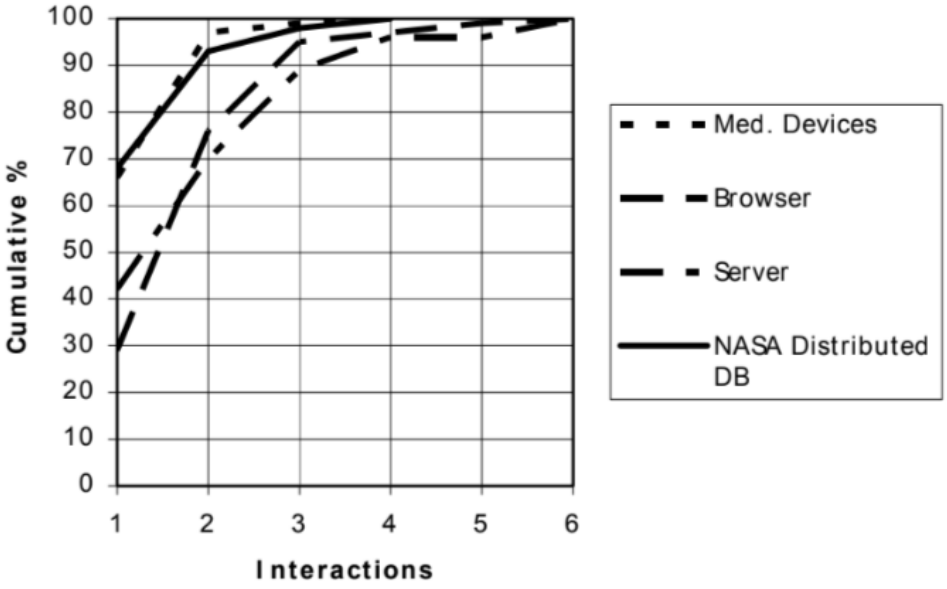

Figura 2.1: Porcentajes de detección de fallos con t-way. Gráfico extraído de [1].

Existen diferentes enfoques a la hora de abordar el problema de generar pruebas combinatorias. Existen 3 tipos: el acercamiento algebraico, el computacional y la búsqueda heurística. El primero, el acercamiento algebraico, consiste en extensiones de métodos matemáticos para construir arreglos ortogonales de Taguchi [9]. Básicamente crean conjuntos de prueba usando reglas predefinidas, incluso algunos, calculan el conjunto de pruebas directamente mediante una función matemática. Existen varias ideas en las que basarse para crea un algoritmo algebraico, dos de los más comunes son:

- Matrices ortogonales: La idea de usar matrices ortogonales [17, 18] tienen una gran ventaja, y es que su construcción puede ser muy rápida y siempre de una forma óptima. Pero existen una serie de limitaciones que hacen poco práctico su uso. No siempre existen matrices ortogonales, por tanto, no se puede aplicar a todos los problemas. Además, cada parámetro está obligado a tener el mismo número de valores. Y existe una tercera limitación, ya que con cada interacción de t-way debe cubrirse en el mismo número de veces.
- Construcción recursiva: Se basa en una estructura más general, en usar matrices de cobertura. Estas requieren que cada interacción de t-way se cubra al menos una vez. Al igual que las matrices ortogonales, la construcción recursiva también puede ser rápida, pero también tiene sus restricciones sobre el número de parámetros y el tamaño del dominio.

Las búsquedas heurísticas son algoritmos que deciden cuál de entre las varias opciones promete ser la mejor para alcanzar el resultado final. El problema es que no tiene por qué encontrar la solución más óptima. Es decir, no se garantiza de que tenga las mejores propiedades, ya bien sea que cubra la mayor cantidad de combinaciones creando el menor conjunto de pruebas posible, o que el coste sea óptimo. Uno de los algoritmos más destacados es el Stimulated Annealing [19, 20]. Este, ha conseguido ofrecer muchos de los conjuntos de prueba más pequeños para diferentes combinaciones de sistema, pero con un coste computacional muy elevado.

Por contra, el enfoque computacional, se caracteriza por no tener las restricciones que tienen los métodos algebraicos, ya que están dotados de una mayor flexibilidad. Algunos de estos algoritmos son: AETG [21, 22], TCG [23], DDA [24], IPO [25], etc.

2.2. Algoritmo In-Parameter-Order

Todos los enfoques se pueden extender a pruebas de 3-way y más allá. Pero ninguno de ellos está optimizado para este tipo de combinaciones. La complejidad de encontrar el conjunto de pruebas mínimo es un problema NP-Completo. Dada las limitaciones que presentan los enfoques algebraicos y los costes computaciones de las búsquedas heurísticas, se apuesta más por un enfoque computacional. Cada algoritmo que se ha desarrollado siguiendo el enfoque computacional, tiene sus ventajas y sus desventajas. Uno de los que acumula varias investigaciones es In-Parameter-Order (IPO). Este algoritmo surgió a manos de Lei y Tai en 1998 [25], con la idea de resolver el problema por pares. Más tarde, en 2007, se creó una variante llamada IPOG [26], que permite modificar el t-way y alcanzar mayores porcentajes de cobertura.

A partir de esta base, han surgido múltiples variantes como IPOG-D [27] en 2008, una variante que emplea D-construction, un enfoque de construcción recursiva. IPOG-D consigue reducir el número de combinaciones que deben enumerarse. Lo que permite optimizar el uso de memoria, aunque sus resultados solo son ligeramente mejores a IPOG. Ese mismo año aparecen las variantes IPOG-F e IPOG-F2 [28]. Ambas variantes modifican el algoritmo codicioso de crecimiento horizontal usando la aleatorización. Estas variantes rompen con la característica de ser un algoritmo determinista. Con IPOG-F e IPOG-F2 se observó pequeñas diferencias en el tiempo y el tamaño de la matriz de cobertura. En 2010 surge MC-MIPOG [29], la versión de IPOG que aprovecha el paradigma de programación multihilo. Pero es en 2014 cuando se incorpora el concepto de restricciones y como resolver el problema SAT. Aquí nace IPO SAT [30]. Dado que IPOG es una evolución de IPO, IPOG es totalmente compatible con SAT. Esto queda demostrado en [31] junto a la implementación de la herramienta ACTS desarrollada por el NIST.

Capítulo 3

In-Parameter-Order

En este capítulo se presenta el algoritmo IPO, uno de los algoritmos de combinación por pares, y una variante más extendida llamada IPOG de combinación por t-way. Finalmente se presenta un ejemplo de combinación de pruebas usando IPO.

3.1. Conceptos del algoritmo

Dado que la complejidad del problema de generar conjuntos de pruebas 2-way o más es un problema NP-completo, es necesario emplear una estrategia práctica que genere conjuntos lo más pequeños posibles. Por tanto, Lei y Tai idearon el algoritmo In-Parameter-Order (IPO) [25], uno de los algoritmos que han surgido en los últimos años para intentar satisfacer este problema. Dicho algoritmo destaca por generar pruebas parciales e ir ampliándolas en cada iteración. A diferencia de muchos otros algoritmos (por ejemplo, AETG) que generan una prueba con cada iteración. Esta diferencia le permite lograr un orden de complejidad menor que AETG y resultados más comparativos. Inicialmente, este algoritmo solo cubre pares o 2-way, pero han ido surgiendo variantes con múltiples optimizaciones y mejoras que permiten la generación de pruebas t-way. Dichas variantes han demostrado que se desempeñan mejor que otras herramientas, tanto en términos del tamaño del conjunto de pruebas como en el tiempo de ejecución.

Para la resolución de un sistema con dos o más parámetros de entrada, la estrategia IPO genera un conjunto de pruebas con los dos primeros parámetros. Después amplía el conjunto de pruebas con el siguiente parámetro del sistema, cubriendo todas las combinaciones posibles no cubiertas. Este último paso se repite hasta cubrir el resto de parámetros.

La extensión de un conjunto de pruebas para la adición de un nuevo parámetro se hace en dos pasos, el crecimiento horizontal y el crecimiento vertical. El primero extiende cada prueba existente agregando un valor posible del nuevo parámetro. El segundo paso, agrega nuevas pruebas que contienen pares que hasta ahora no habían sido cubiertos con las pruebas ya generadas. Una vez cubiertos todos los parámetros, se obtiene el conjunto de pruebas final que cubre todas las combinaciones por pares. La Figura 3.1 muestra un diagrama de flujo sobre el funcionamiento descrito de IPO.

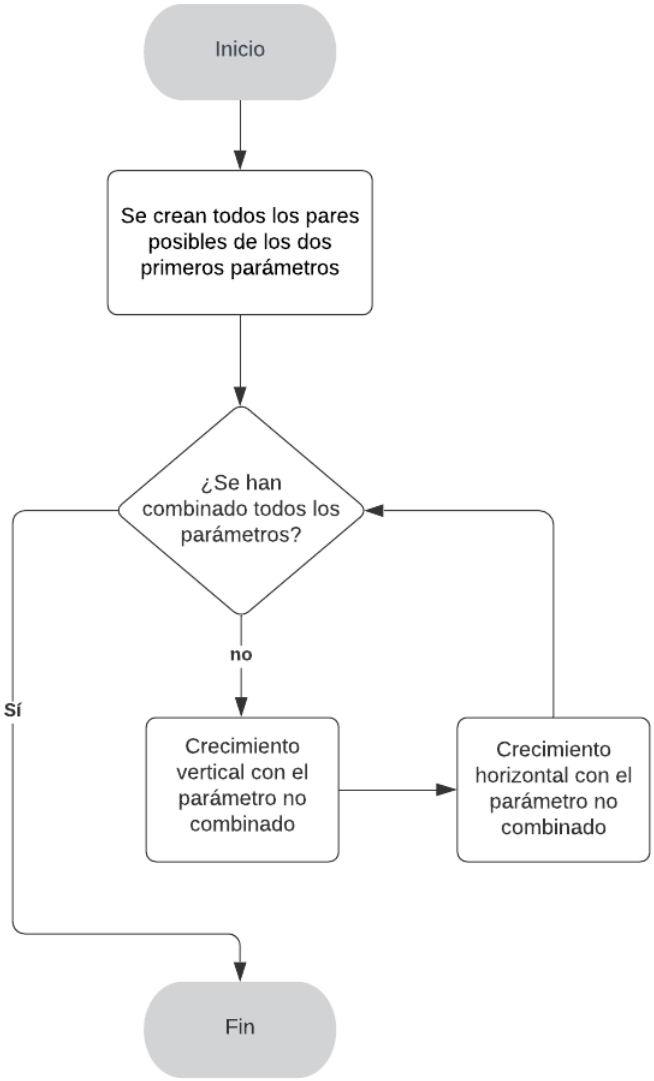

Figura 3.1: Diagrama de flujo del algoritmo IPO

Los resultados obtenidos con este algoritmo frente a AETG (siendo este un referente en el momento de la publicación de IPO) son bastante similares, pero en algún caso ligeramente superior, y otros algo pobres. La Tabla 3.1 son los resultados de la comparativa publicados en [25]. Se puede observar cómo en diferentes sistemas se comporta muy similar.

Sistema	Parámetros	FireTest (IPO)	AETG
S1	4 parámetros y 3 valores por parámetro.	9	9
S2	13 parámetros y 3 valores por parámetro.	19	15
S3	61 parámetros (15: 4 valores por parámetro, 17: 3 valores por parámetro, 29: 2 valores por parámetro)	36	41
S4	75 parámetros (1: 4 valores por parámetro, 39: 3 valores por parámetro, 35: 2 valores por parámetro,).	29	28
S5	100 parámetros y 2 valores por parámetro.	15	10
S6	20 parámetros y 10 valores por parámetro.	218	180

Tabla 3.1: IPO vs AETG

3.2. Explicación del algoritmo

Como se menciona en la sección anterior, el algoritmo IPO se divide en dos partes. Una primera que construye el problema inicial (ver línea 5 del pseudocódigo de esta sección), y una segunda que lo amplia (líneas 16-26).

```
1  TestSet Ipo() {
2
3      // Se crean todos los posibles pares de los 2 primeros
4      // parametros
5      testSet = paresIniciales();
6      if (2 == numero de parametros) {
7          // El conjunto de pruebas final es \textit{testSet}
8          return testSet;
9      }
10
11     // Se amplia el conjunto de pruebas con los parametros no
12     // cubiertos
13     for (i = 2; i < numero de parametros; ++i) {
14
15         // Se amplian las pruebas existentes con valores
16         // posibles del parametro i
17         crecimientoHorizontal(testSet);
18
19         // Se generan nuevas pruebas con las combinaciones no
20         // cubiertas hasta ahora
21         crecimentoVertical(testSet);
22
23     }
24     return testSet;
25 }
```

En la primera parte se creará una matriz, siendo las filas cada prueba generada, y cada columna representa cada uno de los parámetros del sistema. Dado que la combinación es 2-way, la matriz se creará combinando los dos primeros parámetros, es decir, que tendrá dos columnas. Pero la matriz contendrá tantas filas como combinaciones sean posibles entre estos dos parámetros.

Si en el sistema existen solo dos parámetros, solo se creará la matriz inicial (líneas 7-12) y no se podrá ampliar el conjunto final de pruebas, ya que no quedarían más parámetros que combinar.

En caso contrario, se coge el parámetro siguiente aun no combinado. Primero se realiza el crecimiento horizontal (línea 20). Se llama así porque consiste en añadir una nueva columna a la matriz. Se coge valores del *i-parámetro* y se combinan con las pruebas creadas hasta el momento. Lo normal es que no se consigan cubrir todas las combinaciones posibles con los nuevos valores. Por tanto, hay que crear nuevas pruebas que, si cubran todas estas combinaciones no cubiertas, esto es el crecimiento vertical (línea 24). Estos pasos se repiten hasta combinar todos los parámetros restantes del sistema.

Una vez combinados todos los parámetros, el conjunto de pruebas está terminado cubriendo todos los pares.

3.3. Variación In-Parameter-Order-General

La idea del algoritmo IPOG (In-Parameter-Order-General) surge ante la necesidad de generalizar el algoritmo base IPO para que permita combinaciones t-way 2 o más. Véase [26]. Por tanto, el flujo del algoritmo base varía como se puede observar en el diagrama de flujo 3.2. IPOG se convierte en el punto de partida para combinaciones t-way dentro de la familia de algoritmos IPO.

Esta variación mantiene una estructura muy similar al algoritmo base, pero cuando se crea la matriz de combinaciones iniciales de forma codiciosa, en vez de hacerlo por pares, IPOG lo hace según el t-way establecido (línea 5). Es decir, si establecemos un 3-way, se cogerán los tres primeros parámetros y se crearán todas las combinaciones posibles (de tamaño 3, tríos) y se añadirán a la matriz inicial. La matriz tendrá 3 columnas. A medida que se aumente el t-way, se irán cogiendo más parámetros y aumentando el número de columnas de la matriz.

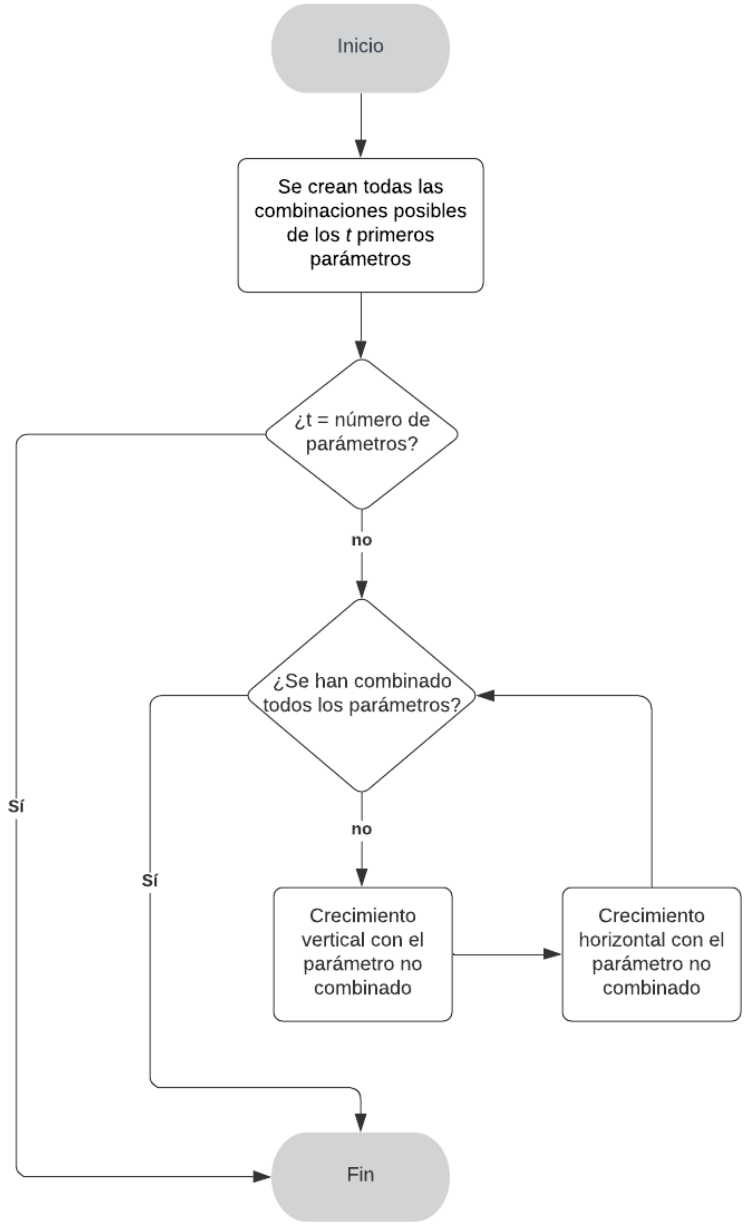

Figura 3.2: Diagrama de flujo del algoritmo IPOG

Una vez creada la matriz inicial, se combinan el resto de parámetros. Para ello se hace el crecimiento horizontal (expandir pruebas ya existentes) y el crecimiento horizontal (generar nuevas pruebas). Pero estos dos pasos se vuelven algo más complicados. Antes de combinar el parámetro, hay que crear todas las t-way combinaciones posibles con el parámetro y las pruebas ya existentes (línea 20). Una vez con todas las combinaciones, se buscan las combinaciones, que, al combinarlos con las diferentes pruebas del testSet, hagan que se cubran el mayor número de combinaciones con el resto de valores de v. Esto es el crecimiento horizontal (líneas 23-32). Una vez se añaden todas las posibles combinaciones nuevas, se eliminan para simplificar el crecimiento vertical. Con todas las combinaciones restantes, se realiza el crecimiento vertical

(líneas 36-60). Para ello se coge una combinación no cubierta, si está cubierta se elimina (línea 41). Se intenta combinar con alguna prueba ya existente (línea 48), por si no se ha podido eliminar en la línea 30. Si no se puede modificar ninguna prueba, se crea una nueva.

Estos pasos se realizan con todos los parámetros no cubiertos. y una vez combinados todos los parámetros, el conjunto de pruebas está terminado cubriendo todas las combinaciones t-way.

```
1  TestSet Ipog() {
2
3      // Se crean todas las combinaciones posibles de todos los t
4      // primeros parametros
5      testSet = combinacionesIniciales();
6
7      if (t == numero de parametros) {
8
9          // El conjunto de pruebas final es \textit{testSet}
10         return testSet;
11
12     }
13
14     // Se amplia el conjunto de pruebas con los parametros no
15     // cubiertos
16     for (i = t; i < numero de parametros; ++i) {
17
18         // Se crean todas las combinaciones de valores t-way sobre
19         // el i-parametro
20         v = combinacionesParametro(testSet, i-parametro);
21
22         // Crecimiento horizontal del i-parametro
23         for (j = 0; j < testSet; ++j) {
24
25             // Se cogen los valores de v y se combinan con j-testSet
26             expandir(j-testSet, v);
27
28             // Se eliminan las combinaciones en v ya cubiertas al
29             // combinar el valor anterior
30             eliminarCombinacionesCubiertas(v);
31
32         }
33
34         // Crecimiento vertical del i-parametro con las
35         // combinaciones restantes
36         for (k = 0; k < v; ++k) {
37
38             if (k-combinacion ya cubierta en testSet) {
39
40                 // Se elimina la combinacion
41                 eliminarCombinacion(v, k);
```

```
42
43              } else {
44
45                  // Intenta modificar una prueba existente del testSet
46                  // para cubrir la k-combinacion
47                  modificarTest(testSet, k-combinacion);
48                  if (no prueba generada) {
49
50                      // Se crea una nueva prueba
51                      crearTest(testSet, k-combinacion);
52
53                  }
54
55                  // Se elimna la k-combinacion
56                  eliminarCombinacion(v, k);
57
58              }
59          }
60      }
61
62      return testSet;
63  }
```

Los resultados obtenidos con esta variación del algoritmo base frente a otros algoritmos que permiten también combinaciones t-way son muy positivas. La comparativa publicada en [26], utiliza el sistema Traffic Collision Avoidance System (TCAS) para realizar las pruebas. Este sistema utilizado en otros estudios de pruebas de software [32, 33], cuenta con 12 parámetros de diferentes tamaños. La Tabla 3.2 muestra los resultados obtenidos (número de pruebas necesarias para cubrir todas las combinaciones) junto a sus tiempos de ejecución.

t-way	**IPOG**		**ITCH**		**Jenny**		**TConfig**		**TVG**	
	Pruebas	tpo.	Pruebas	tpo.	Pruebas	tpo.	Pruebas	tpo.	Pruebas	tpo.
2	100	0.8s	120	0.73s	108	0.001s	108	>1h	101	2.75s
3	400	0.36s	2388	1020s	413	0.71s	472	>12h	9158	3.07s
4	1361	3.05s	1484	5400s	1536	3.54s	1476	>21h	64696	127s
5	4219	18.41s	NA	>1día	4580	43.54s	NA	>1día	313056	1549s
6	10919	65.03s	NA	>1día	11625	470s	NA	>1día	1070048	12600s

Tabla 3.2: Resultados IPOG

3.4. Ejemplo de elaboración de pruebas combinatorias con IPO

Dado un sistema que se quiere probar, que consiste en una aplicación web, se desea comprobar que se visualiza correctamente en diferentes entornos. Por tanto, hay que definir todos los parámetros y los valores posibles de cada parámetros (ver Tabla 3.3).

Modo de visualización	**Navegador**	**Sistema operativo**
Escritorio	Chrome	Windows
Tableta	Firefox	Linux
Móvil	Safari	MacOS
		Android
		iOS

Tabla 3.3: Ejemplo IPO: Parámetros del sistema

Si se desea probar de forma exhaustiva, obtendríamos un total de 5 * 3 * 3 = 45 pruebas. En cambio, si aplicamos el concepto de pruebas combinatorias con el algoritmo IPO y 2-way, obtenemos 15 pruebas.

El primer paso para poder resolver el sistema, es crear la matriz con todas las combinaciones posibles de los t primeros parámetros. En este caso, se aplica 2-way, por tanto $t = 2$. Cogiendo todos los posibles valores de los dos primeros parámetros, obtenemos la matriz inicial 3.1.

$$\begin{pmatrix} Escritorio & Chrome \\ Escritorio & Firefox \\ Escritorio & Safari \\ Tableta & Chrome \\ Tableta & Firefox \\ Tableta & Safari \\ Movil & Chrome \\ Movil & Firefox \\ Movil & Safari \end{pmatrix} \tag{3.1}$$

Esta matriz inicial se convierte en el conjunto de pruebas de inicio. Después se pasa a la segunda parte del algoritmo, combinar el resto de parámetros. Por el momento se han combinado los dos primeros parámetros (Modo de visualización y Navegador), faltaría el tercer parámetro: Sistema operativo. Para ello, se realiza el crecimiento horizontal, añadir una nueva columna a la matriz ampliando las pruebas actuales, ver Matriz 3.2.

$$\begin{pmatrix} Escritorio & Chrome & \text{MacOS} \\ Escritorio & Firefox & \text{Windows} \\ Escritorio & Safari & \text{Linux} \\ Tableta & Chrome & \text{Linux} \\ Tableta & Firefox & \text{MacOS} \\ Tableta & Safari & \text{Windows} \\ Movil & Chrome & \text{Windows} \\ Movil & Firefox & \text{Linux} \\ Movil & Safari & \text{MacOS} \end{pmatrix} \tag{3.2}$$

Después de aplicar el tercer parámetro a las pruebas ya existentes, se puede ver como existen pares aún no cubiertos. Por ejemplo, los valores “Android” e “iOS” aún no han sido combinados

con ninguna prueba existente. Por tanto, se realiza el crecimiento vertical. Se añaden nuevas pruebas para cubrir todos los pares no cubiertos. Al intentar crear estas nuevas pruebas, se intentan coger valores de los dos primeros parámetros, pero ya están todos los pares cubiertos. En este caso da igual la combinación de valores que se coja. La Matriz 3.3 muestra el resultado tras realizar el crecimiento vertical. La filas marcadas en amarillo son las nuevas pruebas creadas.

$$\begin{pmatrix} Escritorio & Chrome & MacOS \\ Escritorio & Firefox & Windows \\ Escritorio & Safari & Linux \\ \text{Escritorio} & \text{Chrome} & \text{Android} \\ \text{Escritorio} & \text{Chrome} & \text{iOS} \\ Tableta & Chrome & Linux \\ Tableta & Firefox & MacOS \\ Tableta & Safari & Windows \\ \text{Tableta} & \text{Firefox} & \text{Android} \\ \text{Tableta} & \text{Firefox} & \text{iOS} \\ Movil & Chrome & Windows \\ Movil & Firefox & Linux \\ Movil & Safari & MacOS \\ \text{Movil} & \text{Safari} & \text{Android} \\ \text{Movil} & \text{Safari} & \text{iOS} \end{pmatrix} \tag{3.3}$$

En este caso ya no hay más parámetros que combinar, si no tendríamos que repetir los pasos del crecimiento horizontal y vertical hasta combinar todos los parámetros. La Matriz 3.3 ya se puede considerar como el conjunto de pruebas finales. La Tabla 3.4 muestra la misma información, pero si nos centramos en los resultados marcados, podemos observar cómo existen pruebas que son imposibles de ejecutar. Por ejemplo, la prueba 3 es un caso imposible, dado que las distribuciones Linux no tiene el navegador Safari (exclusivo de MacOS e iOS). En este caso perderíamos una prueba. Si se decide eliminar esta prueba, haría que se perdiesen dos pares: ("Escritorio", "Safari") y ("Escritorio", "Linux"). Estaríamos perdiendo cobertura en nuestras pruebas, ya que estos dos pares los podríamos combinar con otros valores ya cubiertos. Para solucionar esto, hay que aplicar restricciones. Estas permiten expresar que combinaciones son posibles o imposibles. Pero esto ya corresponde al problema de satisfacción booleana.

	Modo de visualización	**Navegador**	**Sistema operativo**
1	Escritorio	Chrome	MacOS
2	Escritorio	Firefox	Windows
3	Escritorio	Safari	Linux
4	Escritorio	Chrome	Android
5	Escritorio	Chrome	iOS
6	Tableta	Chrome	Linux
7	Tableta	Firefox	MacOS
8	Tableta	Safari	Windows
9	Tableta	Firefox	Android
10	Tableta	Firefox	iOS
11	Móvil	Chrome	Windows
12	Móvil	Firefox	Linux
13	Móvil	Safari	MacOS
14	Móvil	Safari	Android
15	Móvil	Safari	iOS

Tabla 3.4: Ejemplo IPO: Resultados

Capítulo 4

Problema de satisfacción booleana

En este capítulo se describe en qué consiste el problema de satisfacción booleana y los conceptos que están relacionados. También se explica el diagrama de decisiones binario, una estructura que sirve también como representación de fórmulas booleanas. Y finalmente, se juntan los dos conceptos, como resolver el problema SAT basándose en BDD, y como se aplica en el algoritmo IPOG.

4.1. Conceptos fundamentales SAT

El problema de SATisfacción booleana (SAT) pretende determinar si existe una solución que satisfaga una fórmula booleana. Una fórmula se puede considerar que es satisfactoria cuando existe al menos una combinación de valores de cada variable lógica, que da como resultado verdadero al evaluarla.

Este problema es el que se plantea en la Sección 3.4. Dado que existen combinaciones incompatibles, estas se podrían evitar aplicando la idea del problema SAT. Crear una implementación que resuelva este problema permite, expresado a modo de restricciones que se tienen que cumplir, usarse para comprobar cada prueba parcial o completa que es creada o modificada. Es decir, cada vez que se realiza el crecimiento horizontal, se modifica una prueba, pero no tiene por qué ser válida. Hay que comprobarla. Otro caso es cuando se realiza el crecimiento vertical. En ese momento se crea una nueva prueba y también es necesario verificar sus combinaciones. Este es el concepto de IPO SAT [30]. El conjunto de pruebas resultante que obtendríamos sería manteniendo el porcentaje de cobertura, pero todas las pruebas serían válidas.

Para poder resolver este tipo de problemas hay que tener en cuenta que su complejidad es NP-completo. Fue el primer problema descubierto por el teorema de Cook [34]. Aunque se restrinja el número de literales a tres por cláusula, el problema sigue siendo NP-Completo. Esta variante se conoce como 3-SAT. Pero si se reduce a tener dos literales, su complejidad pasa a polinomial. Variante llamada 2-SAT.

4.2. SAT-Solver

Comúnmente, cuando se habla de SAT-Solver, se entiende que son aquellas implementaciones que están basadas en el algoritmo Davis, Putname, Longemann, Loveland (DPLL) [35]. Este algoritmo es capaz de interpretar y evaluar fórmulas booleanas en la forma normal conjuntiva (en inglés Conjunctive Normal Form, CNF). Por tanto, se pasa a hablar del problema CNF-SAT. Un SAT-Solver es capaz de evaluar fórmulas booleanas en CNF, es decir, que toma como entrada una fórmula booleana en formato CNF y una asignación de las variables de dicha fórmula, y devuelve como resultado de si se satisface la fórmula.

El estándar CNF es una conjunción de cláusulas, y cada una de estas es una disyunción de literales. Cada literal está compuesto por átomos. La Fórmula 4.1 es un ejemplo de una fórmula booleana en lenguaje proposicional. Y la Fórmula 4.2 es la misma fórmula, pero expresada en CNF.

$$\neg(p \rightarrow q) \vee \neg r \tag{4.1}$$

$$(p \vee \neg r) \wedge (\neg q \vee \neg r) \tag{4.2}$$

El algoritmo DPLL sigue una estrategia Backtracking, es decir, una búsqueda recursiva en profundidad hasta encontrar la solución. Muchos de los algoritmos modernos derivan de DPLL por los buenos resultados que da. Y la mayoría de estos algoritmos, mejoran la eficiencia pero solo en ciertas aplicaciones.

4.3. Diagrama de decisiones binario

Existen diferentes formas o estándares para expresar una fórmula booleana, por ejemplo: tablas de verdad, forma normal conjuntiva, forma normal disyuntiva, etc. El diagrama de decisiones binario (en inglés Binary Decision Diagram, BDD) es otra forma más que existe para expresar una fórmula booleana, pero tiene una serie de características que le hace destacar cuando se habla del problema SAT. Este tipo de expresión permite una serie de optimizaciones haciendo que sea más simple la expresión final, ya que se elimina la redundancia. A diferencia de otras formas que también comprimen, permiten realizar las operaciones sobre el BDD, sin tener que llegar a descomprimir.

La función booleana se expresa mediante nodos de decisión para expresar las conjunciones y las disyunciones, y dos nodos terminales para expresar si es satisfactoria o no la función. Todos los nodos se relacionan formado la fórmula, y esta empieza con un nodo raíz y terminando con los nodos terminales. Todos estos nodos conectados, forman una estructura, por tanto, un BDD no solo es una forma de expresar lógica, sino que también es una estructura de datos. Esta característica, junto a la eliminación de la redundancia, hace que un BDD sea una expresión más completa que CNF (expresión que se basan los SAT-Solvers).

La Figura 4.1 muestra la representación de la expresión 4.2. Cada nodo de decisión se conecta con otros nodos mediante arcos indicando si es falso (0) o verdadero (1). Existen dos nodos terminales (0 y 1) para representar si es válida la fórmula.

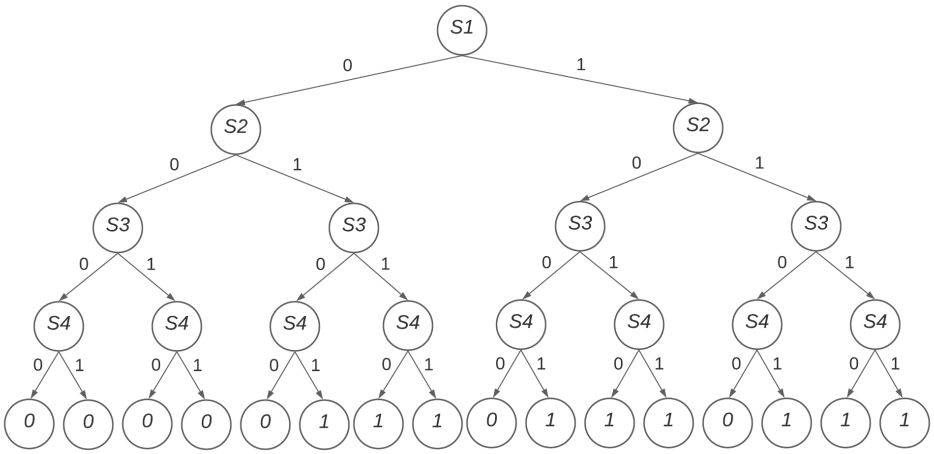

Figura 4.1: Ejemplo BDD completo

La Figura 4.1 es un BDD completo, es decir, sin ningún tipo de compresión. Pero como se menciona antes, un BDD se compone de dos nodos terminales, uno para cuando se satisface la formula booleana (nodo terminal 1) y otro para cuando no se satisface (nodo terminal 0). Estos nodos son redundantes, por tanto se puede comprimir esta información como se muestra en la Figura 4.2. Usar un BDD completo es interesante a nivel conceptual, pero a la hora de llevarlo a la práctica es algo inviable por el consumo excesivo de memoria necesaria.

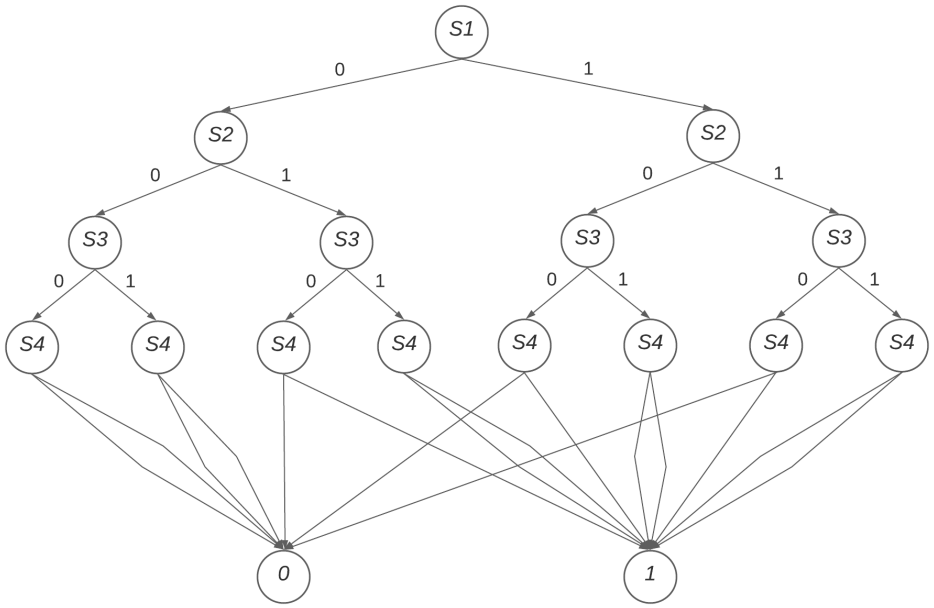

Figura 4.2: Ejemplo BDD comprimido

A la hora de realizar una implementación de un BDD como una estructura, es importante pensar en un BDD como un grafo acíclico dirigido. Es fácil al ver una representación completa de

una fórmula pensar como un árbol binario, ya que cada nodo tiene dos caminos que representan cuando se cumple la condición. Este tipo de estructuras tienen dos grandes ventajas, y es que es más sencillo construir y recorrer este tipo de estructuras. Además, los algoritmos para recorrer la estructura, se basan en estrategias de divide y vencerás, permitiendo que sea más fácil implementar una versión que aproveche el paradigma de computación paralela. Pero los árboles presentan una gran limitación para poder comprimir los BDD, es decir, que tienen tendencia a consumir más memoria.

Implementar un BDD como un grafo (en concreto como un grafo acíclico dirigido) y no como un árbol, evita el problema mencionado que tienen los árboles. Pero presentan la desventaja de ser más laboriosos. La gran ventaja que poseen permite simplificar aún más una fórmula aplicando una compresión mayor. Un árbol binario también permite simplificar la fórmula, pero en menor medida. Para ello podemos ordenar y reducir el BDD. Un BDD ordenado y reducido, o diagrama de decisiones binario reducido ordenado (en inglés Reduced Ordered Binary Decision Diagram, ROBDD), es una versión comprimida y optimizada de la fórmula booleana. Es decir, se elimina toda la información redundante. Un BDD está ordenado cundo los nodos que lo componen están en el mismo orden para los diferentes caminos que existen desde el nodo raíz. Y un BDD está reducido cuando los nodos hijo de cada nodo son eliminados si son isomorfos, y si los isomorfismos de subgrafos están unidos. La Figura 4.3 representa el ROBDD de la expresión 4.2. Normalmente, cuando se habla de BDD, se hablan de ROBDD.

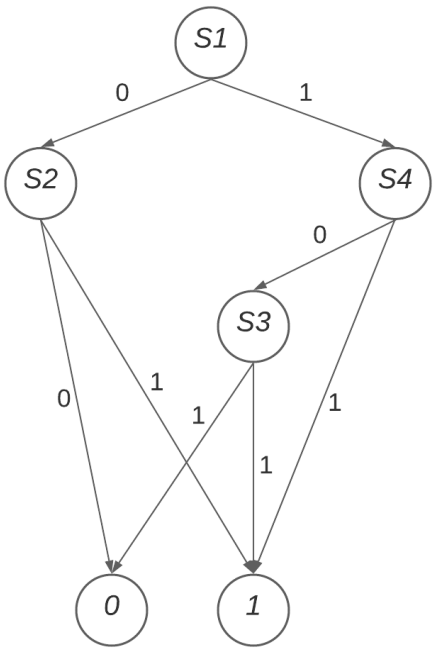

Figura 4.3: Ejemplo ROBDD

Como se menciona en la sección anterior, solo algunos de los algoritmos más modernos basados en DPLL, han conseguido ser más eficientes, pero solo en ciertas aplicaciones. Y es que no todas las implementaciones que pretenden resolver el problema SAT son igual de eficientes en todos los ámbitos. En función de la aplicación que se le quiera dar al SAT, existen diferentes enfoques que pueden resultar más eficaces. En la generación de pruebas combinatorias, puede resultar peor en tiempos de ejecución usar un SAT que se basa en CNF (SAT-Solvers) que en

un BDD. Y es que un SAT-Solver, al usar una estrategia de Backtracking, hace que sean menos eficiente. Por contra, un SAT basado en un BDD hará un consumo mayor de memoria por la estructura de datos.

4.4. IPOG SAT

En la generación de pruebas, aplicar el problema SAT para evitar aquellas pruebas inválidas es clave para promover el desempeño eficiente que permite mantener el porcentaje de cobertura. En el caso del algoritmo base IPO, el problema ya está resuelto en [30]. Así que, aplicar el mismo concepto en la variación IPOG es relativamente sencillo. La primera modificación es cuando se generan las combinaciones iniciales. Es importante partir de una matriz base con solo las combinaciones válidas (línea 5 del pseudocódigo de la función Ipog). La segunda modificación es en la línea 26, cuando se desea hacer el crecimiento horizontal de una prueba. Hay que validar si la prueba y las combinaciones posibles son válidas. Y la última modificación hay que realizarla en el crecimiento vertical (líneas 42 y 46). Antes de intentar coger la combinación pendiente y modificar una prueba, hay que comprobar si es posible hacerlo. Si no, se deja para cuando se añade una nueva prueba.

```
1  TestSet Ipog() {
2
3      // Se crean todas las combinaciones posibles de todos los t
4      // primeros parametros
5      testSet = combinacionesIniciales(sat);
6
7      if (t == numero de parametros) {
8
9          // El conjunto de pruebas final es \textit{testSet}
10         return testSet;
11
12     }
13
14     // Se amplia el conjunto de pruebas con los parametros no
15     // cubiertos
16     for (i = t; i < numero de parametros; ++i) {
17
18         // Se crean todas las combinaciones de valores t-way sobre
19         // el i-parametro
20         v = combinacionesParametro(testSet, i-parametro);
21
22         // Crecimiento horizontal del i-parametro
23         for (j = 0; j < testSet; ++j) {
24
25             // Se cogen los valores de v y se combinan con j-testSet
26             expandir(j-testSet, v, sat);
27
28         }
```

```
29          // Crecimiento vertical del i-parametro con las
30          // combinaciones restantes
31          for (k = 0; k < v; ++k) {
32
33              if (k-combinacion ya cubierta en testSet) {
34
35                  // Se elimina la combinacion
36                  eliminarCombinacion(v, k);
37
38              } else {
39
40                  // Intenta modificar una prueba existente del testSet
41                  // para cubrir la k-combinacion
42                  modificarTest(testSet, k-combinacion, sat);
43                  if (no prueba modificada) {
44
45                      // Se crea una nueva prueba
46                      crearTest(testSet, k-combinacion, sat);
47
48                  }
49
50                  // Se elimna la k-combinacion
51                  eliminarCombinacion(v, k);
52
53              }
54          }
55      }
56
57      return testSet;
58  }
```

Los resultados que se obtienen modificando IPOG con el ejemplo planteado en la Sección 3.4 quedan reflejados en la Tabla 4.1. Pero para obtener dichos resultados hay que establecer la fórmula booleana que contiene las restricciones que se deben cumplir (Fórmula 4.3).

$$(S1 \wedge S2) \vee (S2 \wedge S3)) \vee (S4 \vee S5) \tag{4.3}$$

Siendo las restricciones:

- $S1 \equiv$ navegador != Safari
- $S2 \equiv$ visualización != MacOS
- $S3 \equiv$ visualización = iOS
- $S4 \equiv$ visualización = MacOS
- $S5 \equiv$ visualización = iOS

	Modo de visualización	**Navegador**	**Sistema operativo**
1	Escritorio	Firefox	Windows
2	Escritorio	Chrome	Linux
3	Escritorio	Safari	MacOS
4	Escritorio	Chrome	Android
5	Escritorio	Safari	iOS
6	Tableta	Chrome	Windows
7	Tableta	Firefox	Linux
8	Tableta	Safari	MacOS
9	Tableta	Firefox	Android
10	Tableta	Chrome	iOS
11	Tableta	Chrome	MacOS
12	Tableta	Firefox	MacOS
13	Móvil	Firefox	Windows
14	Móvil	Chrome	Linux
15	Móvil	Safari	MacOS
16	Móvil	Chrome	Android
17	Móvil	Firefox	iOS

Tabla 4.1: Ejemplo IPO: Resultados

Capítulo 5

Implementación

En este capítulo se describe la implementación llevada a cabo con el algoritmo IPOG brevemente. Además, se explican los dos intentos de como se ha resuelto el problema SAT. El primero como un árbol binario, y el segundo como un BDD.

5.1. Implementación del algoritmo IPOG

Con el objetivo de comprobar que usar un BDD para resolver el problema SAT y aplicarlo para la generación de pruebas puede resultar efectivo ante un SAT-Solver, se propone una implementación propia basada en el algoritmo IPOG y un BDD (como se plantea en la Sección 3.3 y 4.4 respectivamente).

La aplicación por línea de comandos desarrollada recibe como parámetros:

```
1 $ ipo [file] [options]
```

El parámetro *file* es el fichero XML del sistema con los parámetros que se quieren combinar y las restricciones aplicadas a estos. En el Anexo B hay un XSD para definir la estructura que debe tener los ficheros XML. El parámetro *options* son todas las opciones adiciones que se pueden aplicar. En la Tabla 5.1 se pueden ver todas las opciones de la aplicación.

Un ejemplo:

```
1 $ ipo Example.xml --tway=4 --sat=dag
```

Opción	Descripción
tway	Configura el tamaño de las combinaciones. El valor mínimo aceptado es 2, si se intenta configurar uno menor, se desprecia. Si no se especifica esta opción, el valor por defecto es 2. Ejemplo: "−−tway=3".
sat	Permite cambiar el algoritmo SAT. Existen dos opciones: "tree" y "dag". La opción "tree" resuelve el problema SAT con un algoritmo basado en un árbol de decisiones. La opción "dag" activa la implementación como un grafo acíclico dirigido (BDD). Si no se especifica esta opción, el valor por defecto es "dag". Ejemplo: "−−sat=dag".
test	Activa el modo test. Si no se especifica esta opción, el modo test está desactivado. Ejemplo: "−−test".

Tabla 5.1: Opciones de la aplicación

La estructura de la aplicación queda mostrada en el diagrama de clases (Figura 5.1). En dicho diagrama se puede observar cómo existe la clase *Ipo*. Esta clase es la implementación del algoritmo IPOG. Básicamente, sus funciones principales son la de cargar un sistema a partir del fichero XML, resolver el problema, y guardar el resultado en un fichero. Existen dos clases auxiliares a esta, que son: *CombinationList* y *CombinationListGroup*. Son dos clases volátiles, y por tanto, no se muestran en el diagrama, pero cabe hacer una pequeña mención. Estas dos clases se emplean para crear todas las combinaciones posibles de una prueba parcial con un parámetro al realizar el crecimiento horizontal y vertical de IPO. Todas las pruebas creadas y modificadas en estos dos pasos, se almacenan en la clase *TestSet*.

Pero toda la implementación del algoritmo IPO necesita una serie de clases de dominio para almacenar toda la información de entrada y salida. Estas clases son: *Parameter*, *Value* y *Test*. Que respectivamente son los parámetros de entrada del sistema, los valores posibles de los parámetros, y las pruebas finales generadas.

La implementación del SAT queda reflejada en varias clases, ya que existen dos implementaciones diferentes. Dado que esto está sujeto a posibles modificaciones y mejoras futuras, se opta por implementar un patrón de diseño Strategy con el objetivo de flexibilizar dichas modificaciones. Dicho patrón consta de la interfaz *ISat* y de las clases *BddSat* y *TreeSat*. La interfaz define las tres funciones principales que deben implementar los SAT, y estas son: añadir los parámetros del sistema, añadir las restricciones definidas, y comprobar si una prueba completa o parcial es válida. Estas tres funciones son implementadas de forma diferente en las clases mencionadas anteriormente. *BddSat* es la implementación del SAT como un BDD, es decir, como un grafo acíclico dirigido. Mientras que *TreeSat* es una implementación que resuelve el problema SAT usando un árbol para las decisiones. Esta última clase necesita una clase auxiliar para los nodos que forman el árbol, *NodeTree*. Las dos implementaciones usan la clase *Expression* para poder procesar las expresiones de texto plano y convertirlas a nodos.

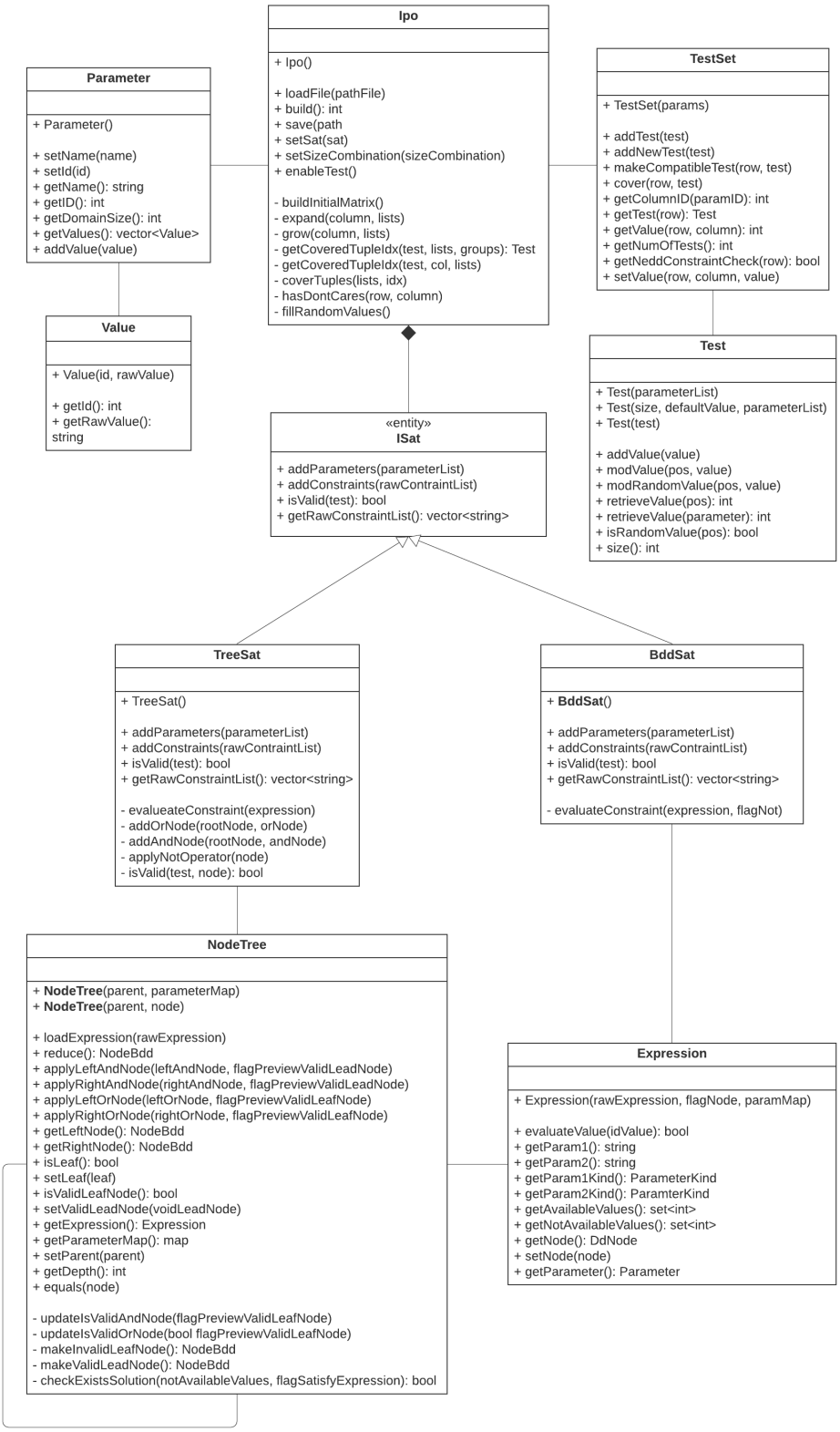

Figura 5.1: Diagrama de clases

5.2. SAT

Las dos implementaciones creadas del SAT, intentan resolver el mismo problema. Es decir, dadas las restricciones del sistema en texto plano, crear una estructura con la fórmula booleana y comprobar si las pruebas generada son válidas. Para ayudar la entrada de las expresiones de las restricciones, se intenta usar un formato similar al CNF. Pero dado que este estándar usa caracteres complicados de introducir ($\wedge$, $\vee$ y $\neg$), se usan los siguientes caracteres para realizar las operaciones lógicas:

- Conjunción: &&
- Disyunción: ||
- Negación: !

Estos caracteres son muy familiares en el ámbito de la programación. Un ejemplo sería:

$$(S1 \ \&\& \ S2) \ \| \ !S3$$

Cada expresión de la fórmula (*S1*, *S2*, etc), evalúa si un valor de un parámetro es válido. Pero los parámetros pueden ser de diferentes tipos. El SAT está pensado para soportar parámetros: booleanos, números enteros, números decimales, caracteres, y cadenas. Las comparaciones que se realizan en cada uno de los tipos de parámetros pueden ser diferentes. Todas estas comparaciones se realizan con los operadores relaciones, siendo los soportados: =, !=, >, >=, <, <=.

Ejemplo:

$$p > 3 \ \&\& \ q = \text{“rana”}$$

5.2.1. Interpretar las restricciones en texto plano

Con el fin de interpretar el texto plano de las restricciones, se sigue una estrategia de divide y vencerás. El objetivo que se persigue es simplificar lo máximo cada expresión. Una vez se obtiene la mínima expresión, se intenta construir el nodo de esa expresión, para posteriormente combinar todos los nodos de las expresiones hasta obtener una estructura que representa a la fórmula booleana. Para poder llevar a cabo esto, se desarrolla una función recursiva que va buscando los operadores lógicos y de prioridad, líneas 15 (disyunción), 25 (conjunción), 35 (operador de prioridad) y 41 (operador de negación) del pseudocódigo de la función *evaluateConstraint*. Se tiene en cuenta el caso especial de cuando es cadena, línea 9, ya que hay que ignorar todos los caracteres hasta encontrar el fin de cadena. Cada vez que se encuentra un operador de prioridad o lógico, se buscan las subexpresiones y se llama de forma recursiva a la función para crear el nodo, y sus descendientes, para posteriormente aplicar el operador.

Cuando se llega hasta la expresión mínima, la indivisible, significa que no se puede aplicar ningún operador lógico o de prioridad. Por tanto, se crea un nodo que representa dicha expresión, líneas 53 a 57.

```
1  Nodo evaluateConstraint(expression) {
2
3      node = null
4
5      // Se recorre la expresion caracter a caracter
6      for (c = 0; c < expression.size(); ++c) {
7
8          // Se comprueba el tipo de caracter
9          if (c-expression == '') {
10
11             // Se identifica el caracter de cadena
12             Se ignoran el resto de caracteres hasta encontrar el
13             caracter de final de cadena
14
15         } else if (c-expression == '&') {
16
17             // Operador logico AND
18             // Se evalua la parte anterior y posterior del operador
19             nodoAnterior = evaluateConstraint(subExpresionIzq)
20             nodoPosterior = evaluateConstraint(subExpresionDer)
21
22             // Se aplica el AND entre los nodos
23             node = addAndNode(nodoAnterior, nodoPosterior)
24
25         } else if (c-expression == '|') {
26
27             // Operador logico OR
28             // Se evalua la parte anterior y posterior del operador
29             nodoAnterior = evaluateConstraint(subExpresionIzq)
30             nodoPosterior = evaluateConstraint(subExpresionDerecha)
31
32             // Se aplica el OR entre los nodos
33             node = addOrNode(nodoAnterior, nodoPosterior)
34
35         } else if (c-expression == '(') {
36
37             // Se busca el cierre para evaluar la subexpresion y se
38             // evalua
39             node = evaluateConstraint(subExpresion)
40
41         } else if (c-expression == '!') {
42
43             // Operador de negacion
44             // Se busca hasta donde afecta el operador para evaluar
```

```
45              // la subexpresion
46              node = evaluateConstraint(subExpresion)
47              node = applyNotOperator(node)
48
49          }
50      }
51
52      // Se comprueba si hay nodo
53      if (node == null) {
54
55          // No se ha aplicado ninguna operacion logica, es una
56          // expresion atomica
57          node = new Node(expression)
58
59      }
60
61      return node
62  }
```

5.2.2. Construcción de la estructura: Primera iteración

El primer intento de construir una estructura que equivalga a la fórmula booleana del problema, es construir esa estructura como un árbol binario. Esta implementación es la clase *TreeSat* presentada en la Sección 5.1. En la sección anterior se presenta la idea de cómo interpretar las fórmulas booleanas en texto plano. Pero para poder desarrollar esta implementación de la estructura de datos, es preciso modificar el algoritmo anterior.

La primera modificación es el tipo de nodo con el que trabaja. Como se presenta en el diagrama de clases 5.1, la implementación de un árbol binario trabaja con nodos *NodeTree*. Esta clase son los nodos del árbol. Y cada uno de estos almacenan la información de su nodo padre, su nodo hijo izquierdo, y su nodo hijo derecho. Sobre estos nodos se aplican las operaciones de disyunción, conjunción y negación.

La función de negación es la más simple de las tres, ya que invierte si es válido o no un nodo terminal, línea 4-7 de la función *applyNotOperator*. En caso de no ser un nodo terminal, pasa a los siguientes nodos hijos haciendo una llama recursiva, líneas 12 y 13.

```
1  void applyNotOperator(node) {
2
3      // Se comprueba si el nodo izquierdo es nodo hoja
4      if (Si el nodo es hoja) {
5
6          // Se invierte
7          node.reverse()
8
9      } else {
```

```
10
11          // No es hoja, se pasa al siguiente nivel
12          applyNotOperator(node.getLefttNode())
13          applyNotOperator(node.getRightNode())
14
15      }
16
17  }
```

La operación de disyunción funciona aplicándose sobre el nodo que representa la parte anterior del operador. Es decir, que el nodo posterior al operador se añade abajo del todo del nodo anterior. Por tanto, hay que recorrer todo el nodo anterior hasta llegar a los nodos terminales mediante una función recursiva:

```
1  void addAndNode(rootNode, andNode) {
2
3      // Se comprueba si el nodo izquierdo es nodo hoja
4      izdNode = rootNode.getLeftNode()
5      if (izdNode es hoja y si izdNode es un nodo valido) {
6
7          // Se aplica la disyuncion sobre el nodo izquierda de root
8          rootNode.applyLeftAndNode(andNode)
9
10     } else if (izdNode no es hoja) {
11
12         // No es nodo hoja, se pasa al siguiente nivel
13         addAndNode(izqNode, andNode)
14
15     }
16
17     // Se comprueba si el nodo derecho es nodo hoja
18     dchaNode = rootNode.getRightNode()
19     if (dchaNode es hoja y si dchaNode es un nodo valido) {
20
21         // Se aplica la disyuncion sobre el nodo derecho de root
22         rootNode.applyRightAndNode(andNode)
23
24     } else if (dchaNode no es hoja) {
25
26         // No es nodo hoja, se pasa al siguiente nivel
27         addAndNode(dchaNode, andNode)
28
29     }
30
31 }
```

Para evitar el consumo excesivo de memoria que pueda presentar el árbol, hay que diseñar optimizaciones a la hora de aplicar los operadores. En la Sección 4.3 se presentaba el concepto de

un BDD reducido, pues para un árbol el concepto es muy similar. Intentar evitar la redundancia. La primera reducción que se puede aplicar es antes de construir el árbol. Es decir, al aplicar una disyunción y tener que combinar dos nodos, si un nodo terminal del primer nodo ya es inválido, es innecesario combinar esa rama del primer nodo. Esto es debido a que es necesario que sean válidos los nodos terminales en una disyunción para que sea válida. Esto queda reflejado en el pseudocódigo de la función *addAndNode*, en la línea 5 (nodo izquierdo) y línea 19 (nodo derecho). En caso de que el nodo terminal ya fuera inválido, no es necesario aplicar la disyunción ya que siempre daría falso. Si el nodo que se evalúa no es un nodo terminal, se pasa al siguiente nivel del árbol, líneas 10 y 13 para el nodo izquierdo, y líneas 24 y 27 para el nodo derecho.

La operación de conjunción es similar a la operación de disyunción, pero presenta una modificación sobre el algoritmo base de interpretación de texto plano. Y es que la operación no se aplica sobre el nodo anterior, si no sobre el nodo posterior. Es decir, hay que recorrer el nodo posterior para encontrar los nodos terminales y añadir el nodo anterior a estos. Esto se realiza de esta forma dado que aporta una mejora de rendimiento muy considerable. Es ocurre porque el nodo anterior siempre será una expresión mínima. Mientras que el nodo posterior puede ser una expresión mínima o una combinación de varias. Es menos costoso aplicar una operación booleana si el nodo sobre el que se realiza la operación es el que más nodos terminales tiene.

Dado que la disyunción tiene optimizaciones de reducción y compresión en tiempo de construcción, en la conjunción también se pueden aplicar. Las líneas 5 y 19 del pseudocódigo de la función *addOrNode* comprueban si el nodo terminal no es válido. En una conjunción, con que uno de los nodos terminales sea válido, el resultado ya es verdadero. Por tanto, si ya es válido un nodo terminal, no es necesario evaluar más. Si el nodo que se evalúa no es un nodo terminal, se pasa al siguiente nivel del árbol, líneas 10 y 13 para el nodo izquierdo, y líneas 24 y 27 para el nodo derecho.

```
1  void addOrNode(rootNode, orNode) {
2
3      // Se comprueba si el nodo izquierdo es nodo hoja
4      izdNode = rootNode.getLeftNode()
5      if (izdNode es hoja y si izdNode no es un nodo valido) {
6
7          // Se aplica la conyuncion sobre el nodo izquierda de root
8          rootNode.applyLeftOrNode(orNode)
9
10     } else if (izdNode no es hoja) {
11
12         // No es nodo hoja, se pasa al siguiente nivel
13         addOrNode(izqNode, orNode)
14
15     }
16
17     // Se comprueba si el nodo derecho es nodo hoja
18     dchaNode = rootNode.getRightNode()
19     if (dchaNode es hoja y si dchaNode no es un nodo valido) {
20
21         // Se aplica la conyuncion sobre el nodo derecho de root
```

```
22          rootNode.applyRightOrNode(orNode)
23
24      } else if (dchaNode no es hoja) {
25
26          // No es nodo hoja, se pasa al siguiente nivel
27          addOrNode(dchaNode, orNode)
28
29      }
30
31  }
```

En las operaciones de disyunción y conjunción, hay otra mejora que se puede aplicar. Y es que puede darse el caso de que exista una contradicción entre las diferentes expresiones de la fórmula booleana. Por ejemplo, cuando se dice que x > 0 y que x == 0, no existen valores posibles para x. En este caso, también es innecesario seguir, dado que este camino del árbol ya es irresoluble. Aparte de ser una optimización, esto se convierte en algo necesario, ya que puede provocar inconsistencias al intentar validar pruebas parciales. Cuando se valida una prueba parcial, no vale con decir si cumple con la fórmula booleana, porque es posible que no existan valores para todos los parámetros involucrados. Por tanto, es preciso comprobar si hay valores válidos para aquellos parámetros que aún no tienen un valor asignado. Para poder realizar estas comprobaciones, se pueden realizar cuando se crea el árbol o cuando se recorre. Pero realizar estas comprobaciones cada vez que se recorre es algo muy costoso, ya que se recorre múltiples veces. Es más eficiente realizar esto cuando se construye el árbol. En concreto hay que comprobar previamente si hay valores validos tras aplicar una disyunción o una conjunción. Hay que ir a los nodos terminales (líneas 9-22 del pseudocódigo de la función *updateValidOrNode*). Cuando se llega al nodo terminal, se realiza la comprobación de si es válido (línea 7). Para indicar si un nodo es válido, en el caso de la conjunción (función *updateValidOrNode*), previamente tiene que ser válido dicho nodo terminal o tiene que descender de una rama válida. Además, tiene que tener al menos un valor posible para cada parámetro. Esto se realiza con la función *checkExistsSolution*. En el caso de una disyunción es igual salvo una diferencia, y es que el nodo terminal tiene que ser válido y tiene que descender de una rama válida. Es decir, se tienen que cumplir las dos condiciones, a diferencia de la conjunción, que con una es suficiente.

```
1  void updateIsValidOrNode(flagPreviewValidLeafNode) {
2
3      // Se comprueba si es nodo hoja
4      if (El nodo es hoja) {
5
6          // Se comprueba si el nodo hoja es valido
7          validLeafNode = (validLeafNode || flagPreviewValidLeafNode)
8              && parent.checkExistsSolution()
9
10     } else {
11
12         // No es nodo hoja, se pasa al siguiente nivel
13         // Se recorre el nodo izquierdo
14         if (hay nodo a la izquierda) {
15
```

```
16              leftNode.updateIsValidOrNode(flagPreviewValidLeafNode)
17
18          }
19
20          // Se recorre el nodo derecho
21          if (hay nodo a la derecha) {
22
23              rightNode.updateIsValidOrNode(flagPreviewValidLeafNode)
24
25          }
26      }
27  }
```

La función *checkExistsSolution* es la encargada de comprobar si hay al menos un valor posible para cada parámetro. Esta función recorre el camino de forma inversa, es decir, de abajo hacia arriba en el árbol hasta llegar al nodo raíz. En cada nodo se comprueba si tiene una expresión asociada (los nodos terminales no tienen), y se comprueba si el camino del árbol que se está recorriendo cumple con la expresión (línea 8). En caso de cumplir la expresión, se cogen todos los valores que no cumplen la expresión (línea 11). En caso contrario, se cogen los valores que si cumplen (línea 16). Estos valores se cogen como valores que no son válidos para ese camino. Si no existen valores válidos para un parámetro (línea 22 y 23), el camino se dice que no es resoluble (línea 38), y por tanto el nodo terminal de ese camino será no válido. En caso de si existir valores válidos y aun no haber llegado al nodo raíz, se pasa al nodo padre (línea 26). Pero si no hay nodo ya es nodo raíz, ya no se puede recorrer más y se da como válido el camino (línea 32). En este caso se dará como válido el nodo terminal.

```
1  bool checkExistsSolution(notAvailableValues) {
2
3      // Se comprueba si el nodo tiene una expresion asociada
4      if (hay expresion) {
5
6          // Hay una expresion y se comprueba si se cumple o no para
7          // coger los valores no disponibles
8          if (Se cumple la expresion) {
9
10             // Se cogen los valores que no cumplen
11             notAvailableValues.add(expression.getNotAvailableValuess())
12
13         } else {
14
15             // Se cogen los valores que si cumplen
16             notAvailableValues.add(expression.getAvailableValues())
17
18         }
19
20         // Se comprueba si aun es resoluble con los valores no
21         // validos
22         if (Aun existen valores posibles para cada parametro en
```

```
23              notAvailableValues y hay padre) {
24
25              // Se llama al nodo padre para seguir recorriendo
26              return parent.checkExistsSolution(notAvailableValues)
27
28          } else if (no hay padre, pero existen valores) {
29
30              // Es resoluble y no hay nodo padre, ya no se puede
31              // comprobar mas
32              return true
33
34          } else {
35
36              // No hay valores para todos los parametros, no es
37              // resoluble
38              return false
39
40          }
41
42      } else if (hay nodo padre) {
43
44          // Se llama al nodo padre para segir recorriendo
45          return parent.checkExistsSolution(notAvailableValues)
46
47      } else {
48
49          // Es el nodo raiz
50          return true
51
52      }
53  }
```

Tras ir combinando cada nodo de las expresiones más pequeñas mediante las conjunciones y las disyunciones, se pueden ir quedando nodos terminales y subcaminos repetidos. Después de aplicar cualquier operación, incluso una negación, hay que comprobar si es posible reducir aún más el árbol. Antes se mostraba como se reducía el árbol sin tener que llegar a construirlo con las funciones *addAndNode* (disyunción) y *addOrNode* (conjunción). Ahora se comprueba con los nodos ya construidos si es posible reducirlos. Esta función *reduce*, primero intenta reducir los caminos de su nodo izquierdo (líneas 4-7) y su nodo derecho (líneas 12-15). Después de reducir sus nodos hijo, se comparan si sus nodos son iguales (línea 20). En caso de ser iguales, se elimina uno de los nodos hijo ya que es redundante y se remplaza el nodo actual por el otro nodo hijo (líneas 23 y 26). De esta forma se elimina un nodo que representa una expresión que su resultado, indistintamente de que sea verdadero o falso, siempre da el mismo resultado. Por tanto, este nodo se vuelve totalmente innecesario.

```
NodeBdd reduce() {

    // Se comprueba si hay nodo izquierdo para reducirlo
    if (hay nodo izquierdo) {

        // Se reduce el nodo izquierdo
        leftNode = leftNode.reduce()

    }

    // Se comprueba si hay nodo derecho para reducirlo
    if (hay nodo derecho) {

        // Se reduce el nodo derecho
        rightNode = rightNode.reduce()

    }

    // Se comprueba si los nodos hijo son iguales
    if (nodo izquierdo es igual al nodo derecho) {

        // Se elimina el nodo derecho y se pasa el nodo izquierdo
        // hijo para sustituir el actual
        delete rightNode
        return leftNode

    } else{

        // No son iguales, no se reemplaza el nodo actual
        return this

    }
}
```

Una vez terminadas las diferentes operaciones lógicas, las optimizaciones y reducciones, el árbol está listo para ser utilizado. Es decir, que la fórmula booleana del problema ya estará construida y se podrá evaluar las pruebas, ya bien sean completas o parciales. Pero si se analiza esta estructura para representar fórmulas booleanas, se puede observar que no es un BDD. Cuando se habla de BDD, se habla de ROBDD. A pesar de aplicar reducciones, sigue existiendo redundancia. Esto es debido a que cada nodo tiene un solo nodo padre. Si un nodo, con sus correspondientes nodos descendientes, es igual a otro nodo en alguna subrama del árbol, no se puede reducir ya que ese nodo solo puede descender de un nodo. Este problema afecta tanto a los nodos de decisión como a los nodos terminales. Además, esta implementación no está ordenada.

La intención de esta primera iteración, persigue intentar representar fórmulas booleanas como un BDD, pero el resultado es un árbol de decisiones, una forma alternativa de expresión de fórmulas booleanas.

5.2.3. Validar pruebas: Primera iteración

La validación de una prueba consiste en recorrer el árbol binario desde el nodo raíz hasta terminar en un nodo terminal. Si el nodo terminal es válido, la prueba se considera válida (línea 7). Pero el recorrido del árbol solo se puede hacer si la prueba está completa, o en caso de ser parcial, que todos los parámetros que están involucrados en la fórmula booleana tengan un valor asignado. Esto queda reflejado en las líneas 12-20. En función de si el valor cumple o no cumple la restricción, se escoge un nodo para seguir recorriendo el árbol. Pero en caso de tener un parámetro que no tiene un valor asignado, hay que explorar los dos caminos, tanto para cuando se cumple la expresión, como cuando no (líneas 27 y 28).

La validación de una prueba mediante un árbol es sencilla, ya que la predicción de si una prueba parcial puede ser válida, ya se realiza en la construcción del árbol.

```
1  bool isValid(test, node) {
2
3      //Se comprueba si se ha llegado al final
4      if (node es nodo terminal) {
5
6          // Se devuelve si el nodo terminal es valido
7          return node.isValidLeafNode()
8
9      } else {
10
11         // Se comprueba si hay un valor asignado al parametro
12         if (hay un valor en la prueba asignado al parametro) {
13
14             // Se comprueba si se cumple la expresion del nodo
15             if (valor del parametro cumple la restriccion) {
16
17                 // Se pasa al nodo hijo derecho
18                 return isValid(test, node.getRightNode())
19
20             } else {
21
22                 // Se pasa al nodo hijo izquierdo
23                 return isValid(test, node.getLeftNode())
24
25             }
26         } else {
27
28             // No se puede validar el nodo actual. Se validan los
29             // dos caminos
30             return isValid(test, node.getLeftNode())
31                 || isValid(test, node.getRightNode())
32         }
33     }
34 }
```

5.2.4. Construcción del BDD: Segunda iteración

En un segundo intento de implementar un BDD y superar las limitaciones que presenta la primera implementación y mejorar los algoritmos codiciosos desarrollados, se implementa la idea de un BDD usando un grafo acíclico dirigido como estructura. Una implementación de un grafo puede resultar más compleja, pero existen ya una serie de soluciones que realizan ya este trabajo. Para este caso se elige la librería Cudd[1], ya que esta es una librería muy optimizada y muy madura para gestionar un BDD.

La implementación se basa en el algoritmo que interpreta el texto plano presentado en la Sección 5.2.1, al igual que la implementación del árbol binario. Pero esta implementación hace uso de los nodos que proporciona la librería Cudd. A parte de esto, también se hacen uso de las funciones de disyunción, conjunción y negación ya proporcionadas. Cada vez que se realizan estas operaciones, la librería Cudd ya se encarga de reducir y ordenar el BDD.

Con esto ya estaría creado el BDD para parámetros del tipo booleanos, pero a diferencia de la implementación anterior, no se realiza nada para ayudar en la predicción de pruebas parciales. Esto se realiza cuando se comprueba si una prueba es válida.

5.2.5. Validar pruebas: Segunda iteración

Dado el BDD base de la fórmula booleana, se pretende añadir nodos para indicar que expresiones se cumplen y cuáles no, con el fin de saber si una prueba, ya bien sea completa o parcial, es válida. Para ello se recorre la lista de restricciones del problema (línea 7). En cada restricción se comprueba si hay un valor válido en la prueba para evaluar (líneas 11-14). Si un valor es válido, se comprueba si satisface la expresión de la restricción. En caso de satisfacerla, se indica al BDD que el parámetro que evalúa la expresión tiene que cumplir ese valor (línea 21). Para ello se realiza una disyunción sobre el BDD. En caso de no satisfacer la expresión, se realiza también una disyunción, pero en este caso se niega, indicando que no se cumple la expresión (líneas 27-28).

Finalmente, con el BDD que tiene las restricciones que se cumplen y las que no, se comprueba si hay valores válidos para los parámetros que no tienen un valor asignado (línea 34).

El problema que presenta este algoritmo voraz es la cantidad de veces que hay que recorrer la lista de todas las restricciones y las veces que hay que añadir nodos para comprobar si es válida la prueba.

[1]Código fuente: https://github.com/vscosta/cudd

```
1 bool isValid(test) {
2
3     // Se coge la referencia del BDD construido inicial
4     current = root
5
6     // Se recorre la lista de expresiones
7     for (i = 0; i < expressionList; ++i) {
8
9         // Se recupera el valor asociado al parametro que evalua la
10        // i-expressionList
11        parameter = i-expressionList.getParameter()
12        value = test.retrieveValue(parameter)
13
14        if (value es valido) {
15
16            // Si es un valor valido, se comprueba si cumple la
17            // i-expressionList
18            if (value cumple la i-expressionList) {
19
20                // Se indica al BDD que la i-expressionList se cumple
21                disyuncion(current, i-expressionList.getNode())
22
23            } else {
24
25                // Se indica al BDD que la i-expressionList no se
26                // cumple
27                disjuncion(current,
28                    negation(i-expressionList.getNode()))
29
30            }
31        }
32    }
33
34    return se comprueba si es valido current
35 }
```

Capítulo 6

Resultados

En este capítulo se exponen los resultados obtenidos con las dos implementaciones del SAT. Después se pasa a realizar una comparativa para averiguar cuál de las dos es más eficiente. Finalmente se realiza una segunda comparativa con la herramienta ACTS, para comprobar si es más eficiente un SAT basado en un BDD o un SAT-Solver aplicados a la generación de pruebas.

6.1. Resultados primera iteración

Para comprobar la efectividad de la primera iteración, se mide el tiempo que tarda en resolver 14 sistemas diferentes. Se puede consultar todos los detalles de los sistemas de pruebas en el Anexo C. La medición de tiempos se realiza en dos partes. La primera consiste en medir el tiempo que tarda en interpretar todas las restricciones y construir el árbol binario. Y la segunda consiste en la medición del tiempo que se tarda en combinar todos los parámetros y verificar que las combinaciones son válidas. La Tabla 6.1 muestra los resultados de la construcción del árbol binario. Para cada caso se especifica el número de parámetros, el número de restricciones y el tiempo que se tarda en construir la estructura. Los tiempos de ejecución no varían en función del t-way, ya que el árbol es siempre el mismo. Los tiempos son relativamente bajos excepto en dos casos: *Berkeley* y *Gg4*. En el caso de *Berkeley* se puede ver que es el caso resuelto con más parámetros y restricciones, y por tanto el que tiene mayor tiempo de construcción. Al basar el SAT en un árbol binario, el crecimiento de la estructura es exponencial. Cuantas más restricciones existan más niveles existen en el árbol.

El caso más extremo se puede ver en el sistema *Violet*. Este sistema es el que posee más parámetros y restricciones. Al imponer tantas restricciones, el crecimiento de árbol es tan grande que hace que sea inviable la construcción. Su tiempo de construcción es mayor de un día, y su consumo de memoria es tan elevado que supera la memoria del entorno de pruebas. Este sistema es irresoluble con un SAT basado en un árbol binario.

Sistemas	Parámetros	Restricciones	T. construcción BDD
aircraft_fm	13	19	0.012 s
Apl	25	37	0.087 s
Berkeley	78	151	29189.300 s
Car	9	15	0.003 s
connector_fm	20	37	0.040 s
fame_dbms_fm	21	37	0.035 s
Gg4	38	99	32.723 s
Graph-product-line-fm	20	45	0.340 s
movies_app_fm	13	23	0.011 s
REAL-FM-12	14	23	0.018 s
smart_home_fm	35	49	0.009 s
stack_fm	17	28	0.110 s
TightVNC	30	41	0.760 s
Violet	101	203	Error

Tabla 6.1: Primera iteración: Tiempos de construcción

Con las estructuras ya construidas de los sistemas, se pasa a medir los tiempos de combinación. El caso de *Violet* no se puede llegar a medir porque no se puede procesar las restricciones. Pero para el resto de casos, la Tabla 6.2 muestra los tiempos de ejecución para cada tamaño de combinación t-way. A medida que el t-way aumenta, el tiempo de ejecución también lo hace. Esto es algo normal, ya que, al aumentar el t-way, el número de combinaciones posibles que hay que comprobar también aumenta. Si se observa la tendencia de crecimiento que existe en todos los sistemas, se puede ver cómo es una tendencia exponencial. La Figura 6.1 representa una muestra de los datos de la Tabla 6.2 mediante un gráfico.

Hay que destacar el caso de *Berkeley*. Cuando se intenta generar las pruebas con combinaciones de 4-way, el tiempo es superior a un 1 día. Dado que ya es un tiempo tan elevado, ni se intenta generar las pruebas con un t-way superior.

Sistemas	2-way	3-way	4-way	5-way	6-way
aircraft_fm	0.007 s	0.035 s	0.137 s	0.424 s	0.985 s
Apl	0.080 s	0.948 s	9.206 s	68.076 s	409.244 s
Berkeley	380.54 s	16804.300 s	>1 día	NA	NA
Car	0.004 s	0.010 s	0.020 s	0.030 s	0.032 s
connector_fm	0.075 s	0.442 s	3.042 s	15.506 s	63.719 s
fame_dbms_fm	0.035 s	0.325 s	2.381 s	14.096 s	66.835 s
Gg4	0.697 s	12.464 s	197.229 s	2309.110 s	22810.2 s
Graph-product-line-fm	0.095 s	0.573 s	3.596 s	17.392 s	71.254 s
movies_app_fm	0.006 s	0.031 s	0.124 s	0.357 s	0.729 s
REAL-FM-12	0.013 s	0.062 s	0.277 s	0.841 s	2.161 s
smart_home_fm	0.114 s	1.618 s	22.517 s	243.170 s	2244.360 s
stack_fm	0.034 s	0.234 s	1.310 s	5.791 s	20.434 s
TightVNC	0.307 s	4.642 s	52.859 s	480.878 s	3413.93 s
Violet	NA	NA	NA	NA	NA

Tabla 6.2: Primera iteración: Tiempos de ejecución IPOG

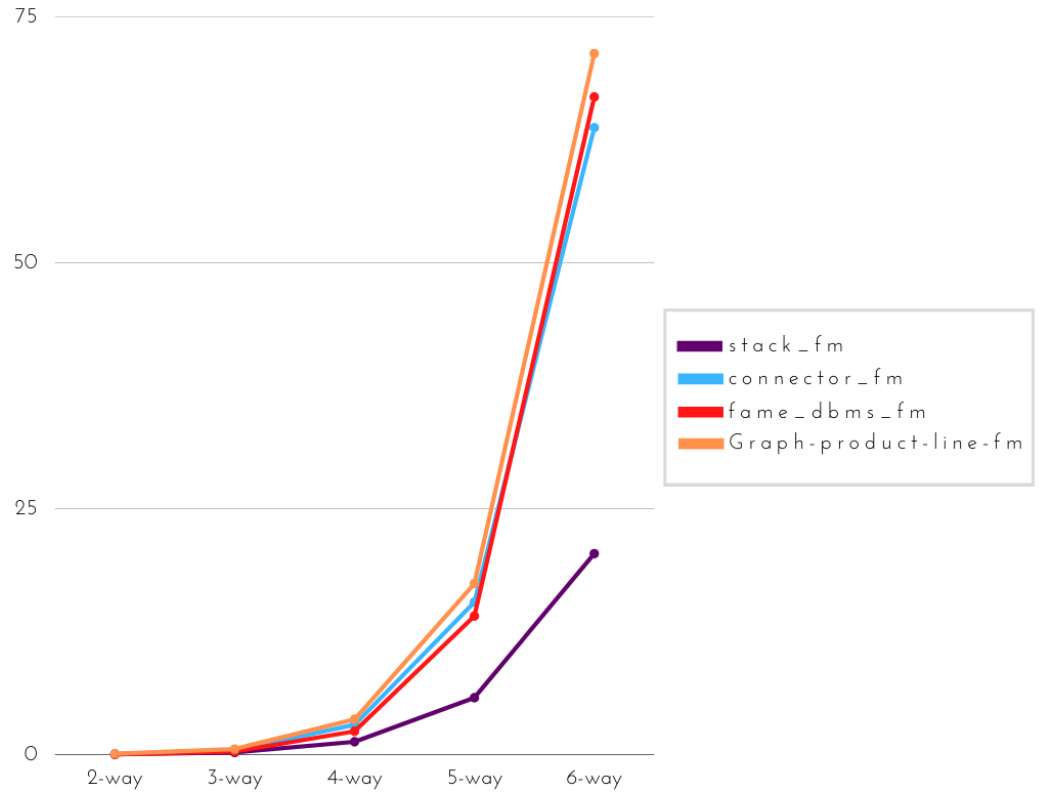

Figura 6.1: Primera iteración: Gráfico con una muestra de tiempos de ejecución IPOG

6.2. Resultados segunda iteración

Se sigue el mismo procedimiento que en la primera iteración. Se mide el tiempo que tarda la segunda iteración en resolver los mismos 14 sistemas. En este caso, la primera parte mide el tiempo que tarde en interpretar todas las restricciones y construir el BDD. Y la segunda toma de tiempos consiste en proceder de la misma forma que la anterior, pero esta vez con el BDD. La Tabla 6.2 muestra los resultados de la construcción del BDD. Al igual que en las pruebas anteriores, los tiempos de ejecución no varían en función del t-way, ya que el BDD es siempre el mismo. Los tiempos son todos muy similares, excepto el sistema *Violet*, siendo este el mayor tiempo para construir el BDD.

Sistemas	Parámetros	Restricciones	T. construcción BDD
aircraft_fm	13	19	0.001 s
Apl	25	37	0.001 s
Berkeley	78	151	0.008 s
Car	9	15	0.001 s
connector_fm	20	37	0.001 s
fame_dbms_fm	21	37	0.001 s
Gg4	38	99	0.002 s
Graph-product-line-fm	20	45	0.001 s
movies_app_fm	13	23	0.001 s
REAL-FM-12	14	23	0.001 s
smart_home_fm	35	49	0.001 s
stack_fm	17	28	0.001 s
TightVNC	30	41	0.001 s
Violet	101	203	0.193 s

Tabla 6.3: Segunda iteración: Tiempos de construcción BDD

Con las restricciones ya procesadas, se pasa a medir los tiempos de combinación. La Tabla 6.4 muestra los tiempos de ejecución para cada tamaño de combinación t-way. Al igual que en la primera iteración, a medida que el t-way aumenta, el tiempo de ejecución también lo hace. Si se observa la tendencia de crecimiento que existe en todos los sistemas, sigue existiendo una tendencia exponencial. La Figura 6.2 representa una muestra de los datos de la Tabla 6.4 mediante un gráfico.

Hay que destacar dos casos: *Berkeley* y *Violet*. En el caso de *Berkeley*, cuando se intenta generar las pruebas con combinaciones de 6-way, el consumo de memoria es superior al entorno de pruebas. Esto hace que el tiempo de ejecución se dispare a días. En este caso de combinación, resolver este problema es inviable. En el caso de *Violet* ocurre el mismo problema. Pero en este sistema ocurre antes ya que existen más parámetros y restricciones.

Sistemas	2-way	3-way	4-way	5-way	6-way
aircraft_fm	0.004 s	0.021 s	0.092 s	0.333 s	0.94 s
Apl	0.026 s	0.260 s	2.625 s	22.318 s	157.035 s
Berkeley	2.287 s	71.077 s	2481.11 s	69311.6 s	Error
Car	0.001 s	0.004 s	0.012 s	0.024 s	0.035 s
connector_fm	0.021 s	0.117 s	0.893 s	5.809 s	31.44 s
fame_dbms_fm	0.017 s	0.131 s	1.09 s	7.641 s	42.874 s
Gg4	0.21 s	2.717 s	41.434 s	588.956 s	6953.74 s
Graph-product-line-fm	0.027 s	0.151 s	1.092 s	7.285 s	37.837 s
movies_app_fm	0.004 s	0.018 s	0.087 s	0.311 s	0.878 s
REAL-FM-12	0.006 s	0.031 s	0.142 s	0.54 s	1.723 s
smart_home_fm	0.055 s	0.821 s	13.039 s	165.955 s	1772.52 s
stack_fm	0.014 s	0.083 s	0.473 s	2.289 s	9.303 s
TightVNC	0.061 s	0.702 s	8.352 s	88.501 s	781.381 s
Violet	78.563 s	4042.11 s	145403 s	Error	Error

Tabla 6.4: Segunda iteración: Tiempos de ejecución IPOG

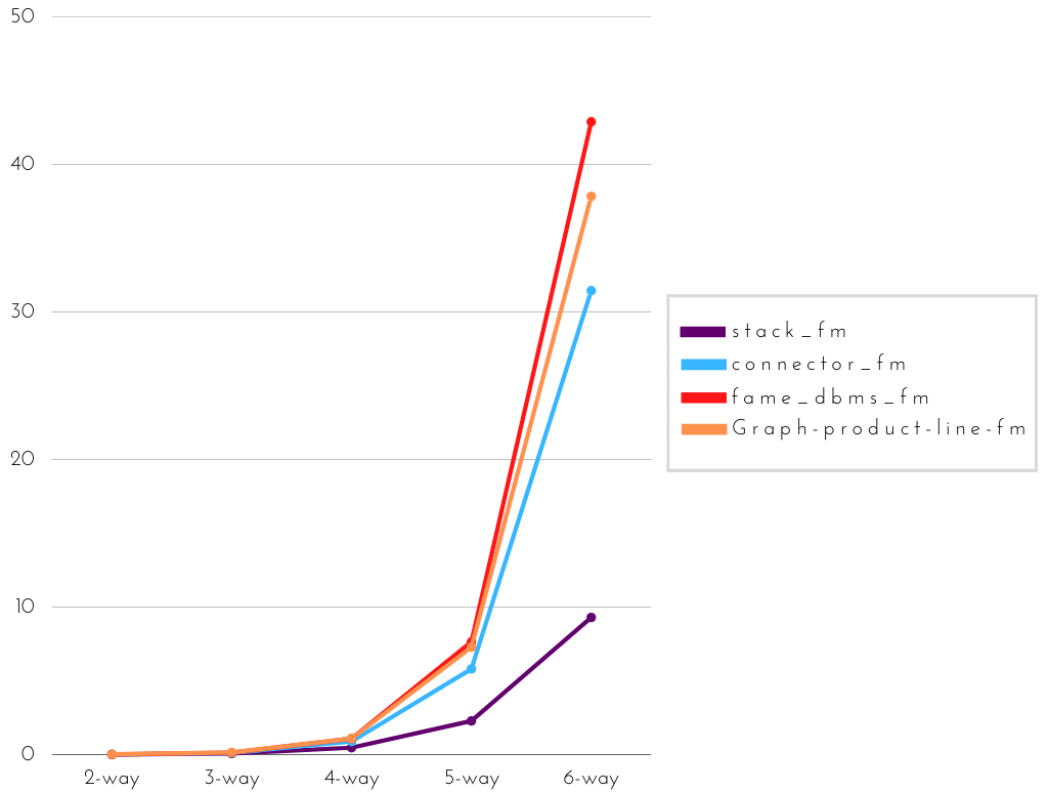

Figura 6.2: Segunda iteración: Gráfico con una muestra de tiempos de ejecución IPOG

6.3. Comparativa

Tras obtener y analizar las mediciones de las dos implementaciones, se pasa a realizar una comparativa. Si se compara el tiempo de construcción de la estructura que representa la fórmula booleana, se puede observar en la Tabla 6.1 y Tabla 6.2, que construir un BDD es más rápido que un árbol binario. Además, si se compara el tiempo de combinación de pruebas, el BDD sigue siendo más eficiente en cuanto tiempo. Esto queda reflejado en la Tabla 6.2 y Tabla 6.4.

Una vez se sabe cuál de las dos implementaciones es más eficiente, se pasa a completar el objetivo del proyecto. Se compara la implementación más eficiente frente a una implementación basada en un SAT-Solver. Se busca averiguar si un SAT basado en un BDD es más eficiente que un SAT-Solver. Para ello se coge la herramienta ACTS, una herramienta que tiene implementado el algoritmo IPOG y que hace uso de un SAT-Solver llamado Choco. Con esta herramienta se combinan los mismos 14 sistemas de las pruebas anteriores. La Tabla 6.5 muestra los resultados obtenidos.

En los resultados se puede ver como el sistema *Berkeley* y *Violet* son irresolubles. El problema es el consumo de memoria excesivo. En el momento que se consume toda la memoria asignada, la herramienta cancela la ejecución. Esto hace ver que el problema no es del SAT, sino que es debido al algoritmo de combinación. Al intentar generar todas las combinaciones posibles durante el crecimiento horizontal y vertical, el consumo de memoria puede resultar excesivo.

Sistemas	2-way	3-way	4-way	5-way	6-way
aircraft_fm	0.254 s	0.260 s	0.270 s	0.292 s	0.336 s
Apl	0.314 s	0.345 s	0.473 s	1.232 s	9.292 s
Berkeley	6.921 s	7.524 s	43.697 s	2336.330 s	Error
Car	0.242 s	0.245 s	0.248 s	0.249 s	0.252 s
connector_fm	0.557 s	0.576 s	0.610	0.798 s	1.497 s
fame_dbms_fm	0.354 s	0.367 s	0.432 s	0.678 s	1.909 s
Gg4	477.359 s	477.885 s	477.973 s	484.947 s	724.162 s
Graph-product-line-fm	1.488 s	1.512 s	1.618 s	1.730 s	2.409 s
movies_app_fm	0.260 s	0.264 s	0.275 s	0.293 s	0.328 s
REAL-FM-12	0.261 s	0.270 s	0.283 s	0.314 s	0.389 s
smart_home_fm	0.328 s	0.376 s	0.686 s	4.996 s	74.581 s
stack_fm	0.280 s	0.288 s	0.321 s	0.410 s	0.735 s
TightVNC	0.457 s	0.499 s	0.696 s	3.141 s	47.470 s
Violet	2.840 s	3.958 s	184.295 s	Error	Error

Tabla 6.5: Tiempos de ejecución ACTS

En las pruebas anteriores se ha comparado los tiempos de ejecución y la tendencia del crecimiento del tiempo de ejecución (Figura 6.3). En este caso, se puede observar que sigue existiendo esa tendencia exponencial al aumentar el t-way. Pero la curva es menor. Al comparar los tiempos, se puede ver como en la mayoría de casos, cuando el t-way es bajo, la implementación basada en un BDD es más rápida. Existe un caso extremo, donde el BDD es más de 2000 veces más rápido (*Gg4*). Pero a medida que el t-way crece, la ganancia de tiempo se invierte, haciendo

que la implementación del SAT-Solver sea más rápida. Esto es debido a que el SAT-Solver Choco proporciona una funcionalidad de comprobar un solo valor de la prueba. Esta comprobación parcial es mucho más eficiente que comprobar todos los valores de una prueba generada. El algoritmo para verificar pruebas del BDD comprueba cada vez todos los valores. Por tanto, a medida que el t-way aumenta, existen más combinaciones que hay que verificar, y el BDD acaba resultando más lento.

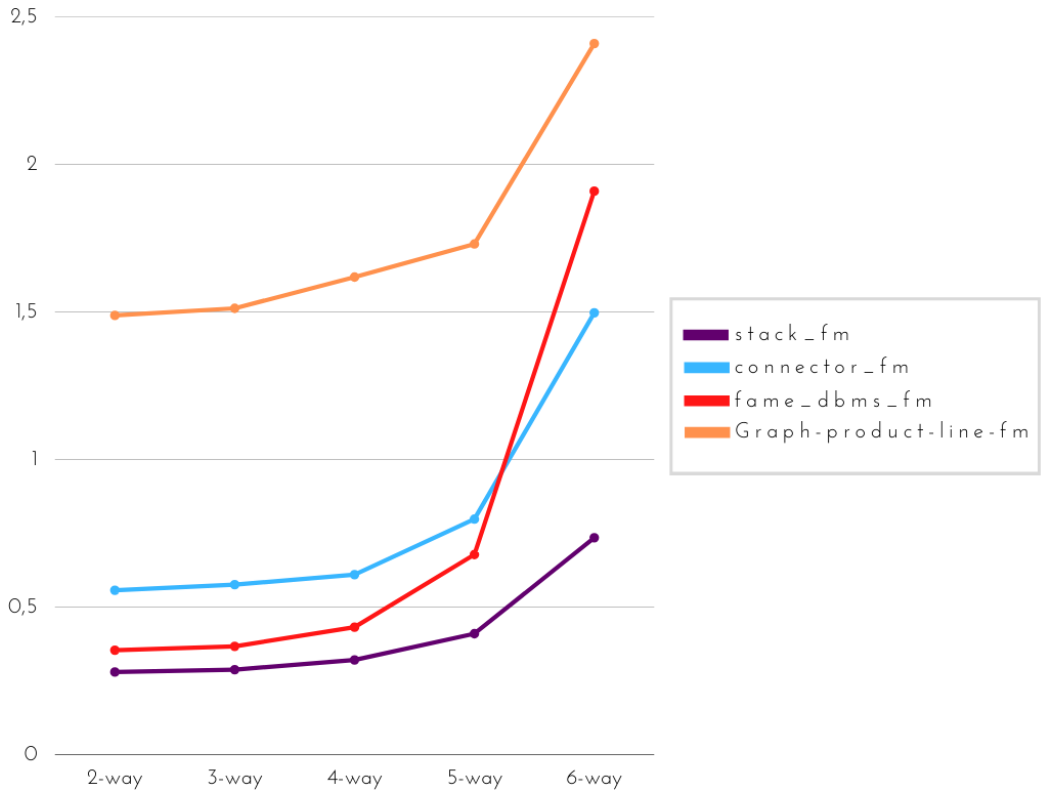

Figura 6.3: Gráfico con una muestra de tiempos de ejecución ACTS

Las Figuras 6.4, 6.5, 6.6 y 6.7 muestran una comparativa de ejecución de una muestra de sistemas. Los sistemas son respectivamente: *connector_fm*, *fame_dbms_fm*, *Graph-product-line-fm* y *stack_fm*. En las gráficas se puede ver como la implementación propia basada en un BDD es más eficiente frente a una implementación basada en un SAT-Solveren t-way bajos. Pero en t-way altos, es más efectivo el SAT-Solver.

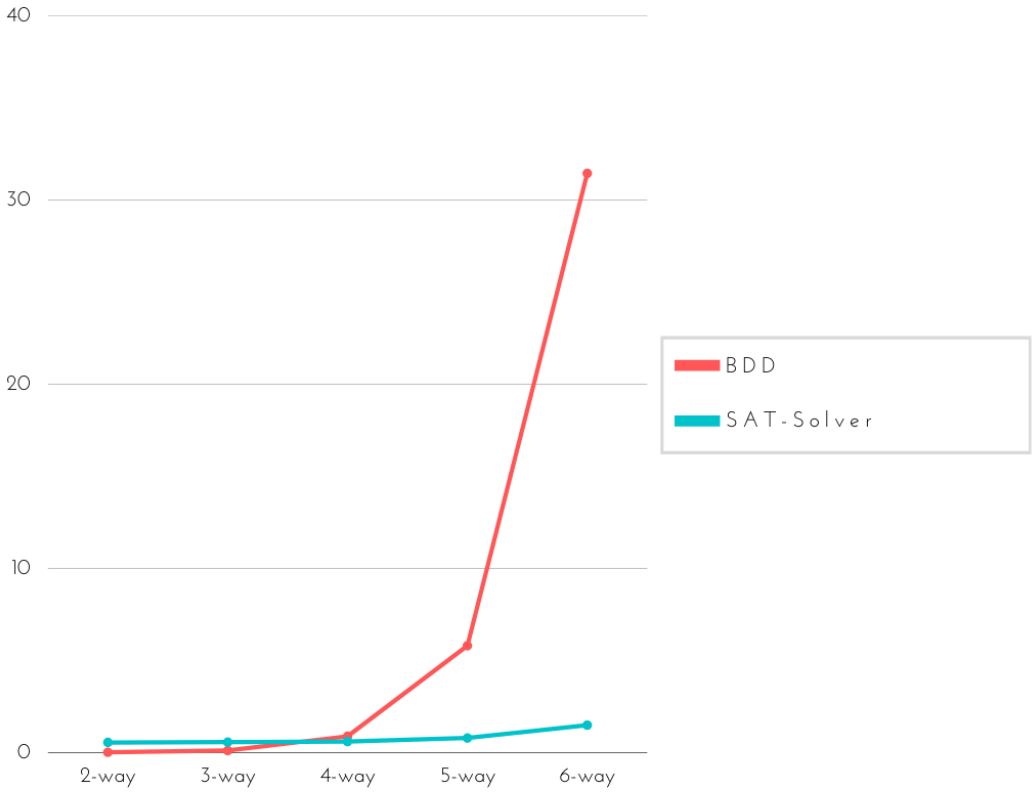

Figura 6.4: Gráfico comparativa tiempos BDD vs Solver: connector_fm

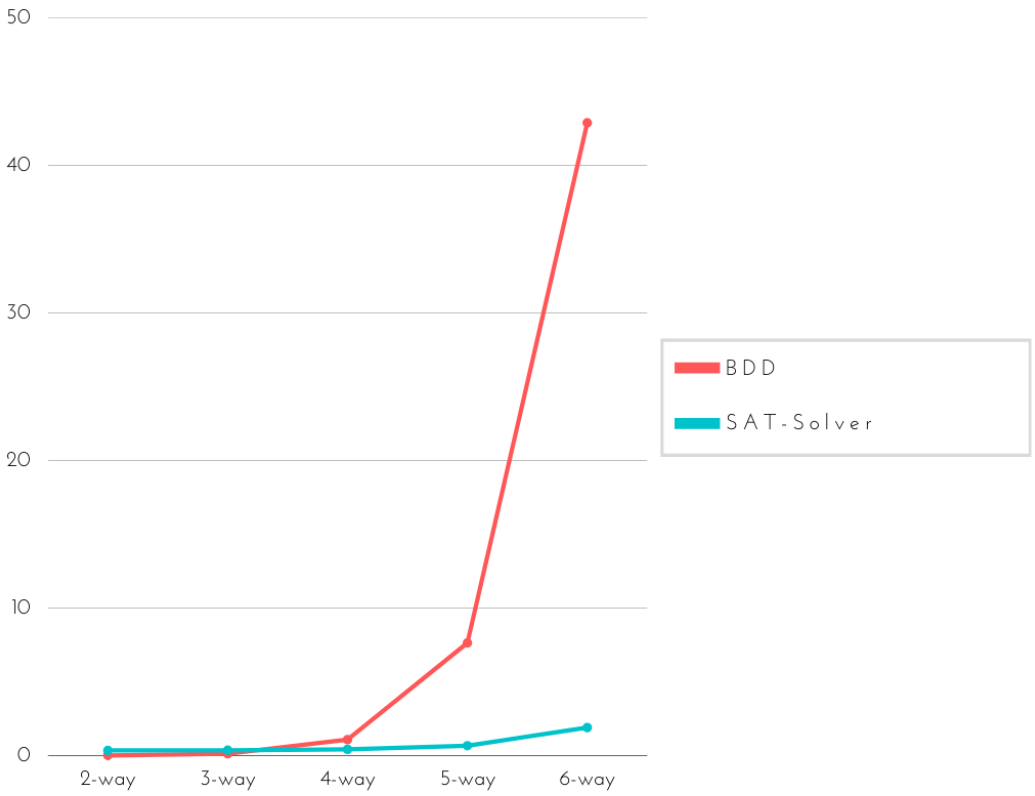

Figura 6.5: Gráfico comparativa tiempos BDD vs Solver: fame_dbms_fm

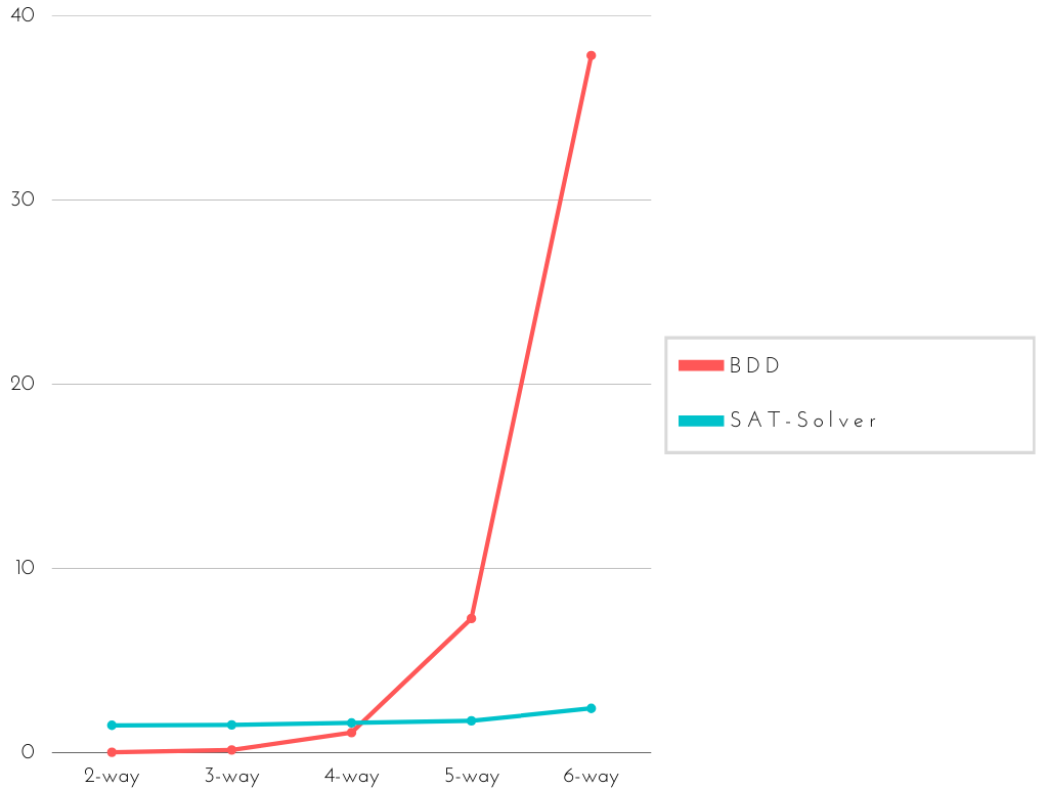

Figura 6.6: Gráfico comparativa tiempos BDD vs Solver: Graph-product-line-fm

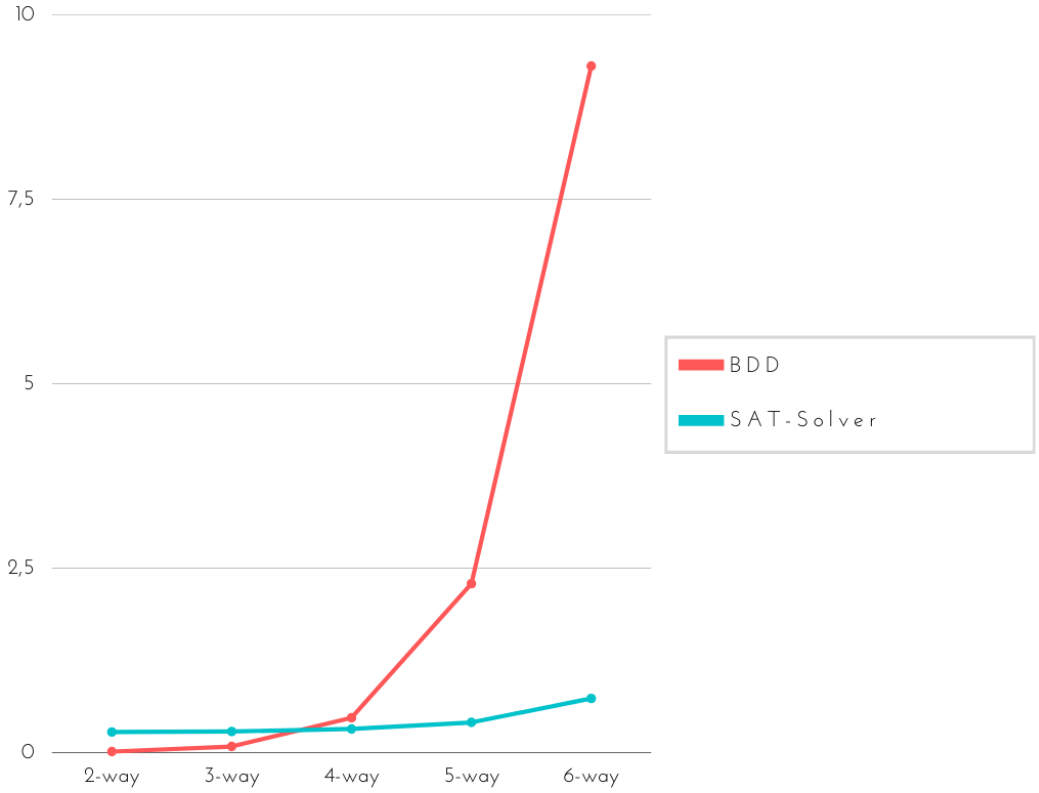

Figura 6.7: Gráfico comparativa tiempos BDD vs Solver: stack_fm

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo final se presenta la conclusión obtenida del proyecto. Además, se idean cuales son los siguientes pasos que debe seguir la investigación.

7.1. Conclusiones

Tras obtener y analizar los resultados del trabajo desarrollado, se puede concluir que la idea de un SAT basado en un BDD es una buena línea para seguir investigando. Los tiempos que se han obtenido con t-way bajos son muy reducidos. Mejores que la implementación de la herramienta ACTS basada en un SAT-Solver. Pero a medida que el tamaño de combinación aumenta, esa ventaja desaparece. El motivo es el algoritmo de validación de pruebas que se presenta en la Sección 5.2.5. El algoritmo está construyendo un BDD a partir del BDD inicial de la fórmula booleana. El nuevo BDD representa el estado actual de la prueba que se quiere validar. Este proceso es el que consume más tiempo de ejecución. Una vez se termina de validar la prueba, el nuevo BDD se elimina. Este proceso se realiza cada vez que se quiere validar una prueba. Sin duda el coste computacional es muy alto.

En el caso del ACTS, este problema no existe. Y esto es debido a que el SAT-Solver que integra, es capaz de realizar comprobaciones de un solo valor. Esta funcionalidad aporta esa diferencia de rendimiento, y hace mejorar la tendencia exponencial del coste.

7.2. Trabajo futuro

El siguiente paso es lograr mejorar el algoritmo de validación de pruebas. Para ello hay que desarrollar las validaciones de un solo valor de la prueba. Esta mejora provocará que los tiempos de validación se reduzcan. Siendo en los tamaños de combinación mayores donde más se notará esta optimización. Para ello habrá que evitar reconstruir el BDD que representa una prueba, y guardarlo.

Pero a pesar de realizar la optimización de validar pruebas, existe una gran cantidad de comprobaciones innecesarias. Actualmente, se generan todas las combinaciones posibles de los valores con el test que se amplia. Luego se comprueba si el test modificado cumplen con las restricciones o no. Es decir, el BDD actúa como una caja negra, y es el algoritmo IPOG quien lo usa para comprobar. Sería mucho más eficiente generar directamente las combinaciones que sí cumplen con las restricciones a partir del estado de la prueba. De esta forma, todas las pruebas que se modifiquen, ya serían válidas. Para poder llevar esto acabo, sería necesario no usar el BDD como caja negra, sino que el algoritmo IPOG se tendría que basar en el BDD. Si se consigue esto, se desarrollaría una nueva variante del algoritmo IPOG más eficiente. Intentar plantear esta idea con un SAT-Solver es inviable. Basar el algoritmo de combinación en un BDD puede suponer el siguiente gran paso en las investigaciones que acumula el algoritmo IPO.

Bibliografía

[1] D Richard Kuhn, Raghu N Kacker, and Yu Lei. Practical combinatorial testing. *NIST special Publication*, 800(142):142, 2010.

[2] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[3] D. Richard Wallace, Dolores R.; Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(04):351–371, 2001.

[4] D Richard Kuhn and Michael J Reilly. An investigation of the applicability of design of experiments to software testing. In *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, pages 91–95, Greenbelt, MD, USA, USA, 2002. IEEE.

[5] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[6] D Richard Kuhn and Vadim Okum. Pseudo-exhaustive testing for software. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 153–158, Columbia, MD, USA, 2006. IEEE.

[7] Kera Zakiyah Bell et al. Optimizing effectiveness and efficiency of software testing: A hybrid approach. *Unpublished PhD dissertation, North Carolina State University*, 2006.

[8] EP George, J Stuart Hunter, William Gordon Hunter, Roma Bins, Kay Kirlin IV, and Destiny Carroll. *Statistics for experimenters: design, innovation, and discovery.* Wiley New York, 2005.

[9] Resit Unal and Edwin B Dean. Taguchi approach to design optimization for quality and cost: an overview. In *Proceedings of the International Society of Parametric Analyst 13th Annual*, 1991.

[10] Myra B Cohen, Charles J Colbourn, and Alan CH Ling. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, 308(13):2709–2722, 2008.

[11] George Sherwood. Effective testing of factor combinations. In *Proc. Third International Conference on Software Testing, Analysis and Review (STAR'94)*, 1994.

[12] Siddhartha R Dalal and Colin L Mallows. Factor-covering designs for testing software. *Technometrics*, 40(3):234–243, 1998.

[13] Neil JA Sloane. Covering arrays and intersecting codes. *Journal of combinatorial designs*, 1(1):51–63, 1993.

[14] Charles J Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, 59(1, 2):125–172, 2004.

[15] Kera Z Bell and Mladen A Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *2005 International Conference on Information and Communication Technology*, pages 221–235, Cairo, Egypt, 2005. IEEE.

[16] Charles Antony Richard Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[17] Robert Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, 1985.

[18] Jeremy M Harrell. Orthogonal array testing strategy (oats) technique. *Seilevel*, 2001.

[19] Manisha Patil and PJ Nikumbh. Pair-wise testing using simulated annealing. *Procedia Technology*, 4:778–782, 2012.

[20] Kun Wang, Yichen Wang, and Liyan Zhang. Software testing method based on improved simulated annealing algorithm. In *2014 10th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pages 418–421, Guangzhou, China, 2014. IEEE.

[21] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[22] David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The combinatorial design approach to automatic test generation. *IEEE software*, 13(5):83–88, 1996.

[23] Nabilah Kamarul Bahrin and Radziah Mohamad. Tcg algorithm approach for uml sequence diagram. In *2015 9th Malaysian Software Engineering Conference (MySEC)*, pages 43–48, Kuala Lumpur, Malaysia, 2015. IEEE.

[24] Renée C Bryce and Charles J Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[25] Yu and Lei. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pages 254–261, Washington, DC, USA, USA, 1998. IEEE.

[26] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 549–556, Tucson, AZ, USA, 2007. IEEE.

[27] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[28] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287, 2008.

[29] Mohammed I Younis and Kamal Z Zamli. Mc-mipog: a parallel t-way test generation strategy for multicore systems. *ETRI journal*, 32(1):73–83, 2010.

[30] Shiwei Gao, Binglei Du, Yaruo Jiang, Jianghua Lv, and Shilong Ma. An efficient algorithm for pairwise test case generation in presence of constraints. In *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pages 406–410. IEEE, 2014.

[31] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.

[32] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200, Sorrento, Italy, Italy, 1994. IEEE.

[33] D Richard Kuhn and Vadim Okum. Pseudo-exhaustive testing for software. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 153–158, Columbia, MD, USA, 2006. IEEE.

[34] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. Association for Computing Machinery, 1971.

[35] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[36] Jacek Czerwonka. Pairwise testing. `http://www.pairwise.org/tools.asp`, October 2019. Última modificación: octubre 2019.

Siglas, abreviaturas y acrónimos

DoE: Design of Experiments.

ACTS: Advanced Combinatorial Testing System.

NIST: National Institute of Standards and Technology.

AETG: Automatic Efficient Test Generator.

IPO: In-Parameter-Order.

IPOG: In-Parameter-Order-General.

TCG: Test Case Generator.

SAT: problema de SATisfacción booleana.

DPLL: algoritmo Davis-Putnam-Logemann-Loveland.

CNF: Conjunctive Normal Form.

BDD: Binary Decision Diagram.

ROBDD: Reduced Ordered Binary Decision Diagram.

Anexo A

Herramientas disponibles

Herramientas disponibles para la generación de pruebas combinatorias, información extraída de [36]:

Nombre	Propietario	
CATS (Constrained Array Test System)	[Sherwood] Bell Labs.	
OATS (Orthogonal Array Test System)	[Phadke] ATT	
AETG	Telecordia	Web-based, commercial
IPO (PairTest)	[Tai/Lei]	
TConfig	[Williams]	Java-applet
TCG (Test Case Generator)	NASA	
AllPairs	Satisfice	Perl script, free, GPL
Pro-Test	SigmaZone	GUI, commercial
CTS (Combinatorial Test Services)	IBM	Free for non-commercial use
Jenny	[Jenkins]	Command-line, free, public-domain
ReduceArray2	STSC, U.S. Air Force	Spreadsheet-based, free
TestCover	Testcover.com	Web-based, commercial
DDA	[Colburn/Cohen/Turban]	
Test Vector Generator		GUI, free
OA1	k sharp technology	
TESTONA	Assystem Germany	GUI, free for non-comercial use
AllPairs	[McDowell]	Command-line, free
Intelligent Test Case Handler (replaces CTS)	IBM	Free for non-commercial use
CaseMaker	Díaz & Hilterscheid	GUI, commercial
PICT	Microsoft Corp.	Command-line, open source at http://github.com/microsoft/pict
rdExpert	Phadke Associates, Inc.	
OATSGen	Motorola	

SmartTest	Smartware Technologies Inc.	GUI, commercial
EXACT	[Yan/Zhang]	
AllPairs	MetaCommunications	Free
ATD	AtYourSide Consulting	GUI, commercial
ACTS [formerly: FireEye]	NIST	GUI
Bender RBT Inc.	BenderRBT	GUI, commercial
Pairwise Test Case Generator	TestersDesk	Web-based
Combo-Test	The Australian eHealth Research Centre	Command-line, free
IPO-s	[Calvagna/Gargantini]	
VPTAG	[Robert Vanderwall]	
SpecExplorer	Microsoft Corp.	GUI, free
IBM Functional Coverage Unified Solution hline CombTestWeb	IBM Universidad de Castilla-La Mancha	GUI, commercial Web-based, free
Hexawise	Hexawise	Web-based, free & commercial
PictMaster	IWATSU System & Software	Spreadsheet-based, free
NTestCaseBuilder	[Murphy]	.NET library
tcases	[Kimbrough]	Command-line, free
Pairwiser	Inductive AS	Web-based, free & commercial
NUnit	Poole et al	Unit test framework
ecFeed	ecFeed AS	Standalone, Eclipse plug-in, and jUnit runner
TechQA		Web-based, free
Pairwise Online Tool	[Dementiev]	Web-based, free
JCUnit	[Ukai]	Unit test framework
CAGen	SBA Research	Web-based and command-line
CTWedge	University of Bergamo	Web-based
CAMetrics	SBA Research	Web-based
SQA Mate Tools: Pairwise	[Sotskov]	Web-based
AllPairsPy	[Hombashi]	Python library
Pairwise Pict Online	[Tamura]	PICT on the web
CoverTable	[Yasuyuki]	Python and TypeScript. Open source

Tabla A.1: Herramientas disponibles para la generación de pruebas combinatorias

Anexo B

XSD fichero del sistema

```
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="System">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Parameters">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Parameter"
                          maxOccurs="unbounded"
                          minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="values">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element type="xs:byte"
                                      name="value"
                                      maxOccurs="unbounded"
                                      minOccurs="0"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute type="xs:byte"
                                name="id"
                                use="optional"/>
                  <xs:attribute type="xs:string"
                                name="name"
                                use="optional"/>
                </xs:complexType>
              </xs:element>
```

```
34              </xs:sequence>
35            </xs:complexType>
36          </xs:element>
37          <xs:element name="Constraints">
38            <xs:complexType>
39              <xs:sequence>
40                <xs:element name="Constraint"
41                            maxOccurs="unbounded"
42                            minOccurs="0">
43                  <xs:complexType>
44                    <xs:simpleContent>
45                      <xs:extension base="xs:string">
46                        <xs:attribute type="xs:string"
47                                      name="text"
48                                      use="optional"/>
49                      </xs:extension>
50                    </xs:simpleContent>
51                  </xs:complexType>
52                </xs:element>
53              </xs:sequence>
54            </xs:complexType>
55          </xs:element>
56        </xs:sequence>
57        <xs:attribute type="xs:string"
58                      name="name"/>
59      </xs:complexType>
60    </xs:element>
61  </xs:schema>
```

Anexo C

Sistemas de pruebas

C.1. aircraft_fm

Parámetros:

Nombre	Valores
Low	true, false
Materials	true, false
Piston	true, false
Jet	true, false
Wood	true, false
Plastic	true, false
Engine	true, false
Cloth	true, false
Aircraft	true, false
Metal	true, false
Shoulder	true, false
High	true, false
Wing	true, false

Restricciones:

- Aircraft
- !Wing || Aircraft
- !Aircraft || Wing
- !Engine || Aircraft
- !Materials || Aircraft
- !Aircraft || Materials

- !High || Wing
- !Shoulder || Wing
- !Low || Wing
- !Wing || High || Shoulder || Low
- !Jet || Engine
- !Piston || Engine
- !Engine || Jet || Piston
- !Jet || !Piston
- !Metal || Materials
- !Wood || Materials
- !Cloth || Materials
- !Plastic || Materials
- !Materials || Metal || Wood || Cloth || Plastic

C.2. Apl

Nombre	Valores
mainBc	true, false
xak	true, false
mainGUI	true, false
Java	true, false
mmatrix	true, false
reform	true, false
bcjak2java	true, false
jak2aj	true, false
APL	true, false
jak2java	true, false
xc	true, false
mainAj	true, false
mainJava	true, false
me	true, false
bc	true, false
drc	true, false
bali	true, false
cpp	true, false
Gui	true, false
web	true, false
jedi	true, false

aj	true, false
Xml	true, false
guidsl	true, false
mainAPL	true, false

Restricciones:

- APL
- !mainAPL || APL
- !APL || mainAPL
- !Java || mainAPL
- !mainAPL || Java
- !Xml || mainAPL
- !cpp || mainAPL
- !Gui || mainAPL
- !drc || mainAPL
- !bc || mainAPL
- !aj || mainAPL
- !mainJava || Java
- !Java || mainJava
- !xc || Xml
- !xak || Xml
- !Xml || xc || xak
- !xc || !xak
- !mainGUI || Gui
- !Gui || mainGUI
- !mainBc || bc
- !bc || mainBc
- !mainAj || aj
- !aj || mainAj
- !reform || mainJava

- !jedi || mainJava
- !jak2java || mainJava
- !mainJava || jak2java
- !bali || mainJava
- !mainJava || bali
- !me || mainGUI
- !mmatrix || mainGUI
- !guidsl || mainGUI
- !web || mainGUI
- !bcjak2java || mainBc
- !jak2aj || mainAj
- !bc || mmatrix
- !me || mmatrix

C.3. Berkeley

Nombre	Valores
BerkeleyDb	true, false
Statistics	true, false
featureINCompressor	true, false
featureDirectNIO	true, false
FConcurrency	true, false
featureChunkedNIO	true, false
featureLatch	true, false
featureStatisticsTransaction	true, false
BTree	true, false
featureStatisticsEnvCaching	true, false
FLogging	true, false
featureTransaction	true, false
featureStatisticsEnvINCompressor	true, false
Evictor	true, false
FPersistency	true, false
featureStatisticsEnvFSync	true, false
featureIO	true, false
featureStatisticsBase	true, false
featureCriticalEviction	true, false
featureStatisticsEnvCleaner	true, false
FBtree	true, false

featureStatisticsEnvBase	true, false
featureLoggingBase	true, false
featureLoggingFiner	true, false
FPersistencyFeatures	true, false
featureStatisticsPreload	true, false
IO	true, false
featureStatisticsEnvCheckpointer	true, false
featureStatisticsEnvEvictor	true, false
featureLoggingDbLog	true, false
featureCustomizableCheckpointerBytes	true, false
featureSynchronizedIO	true, false
BASE	true, false
featureLoggingFinest	true, false
FIOFeature	true, false
featureLoggingFile	true, false
NotChunked	true, false
FDbOperation	true, false
featureHandleFullDiscError	true, false
FEvictor	true, false
featureLoggingConsole	true, false
featureLoggingConfig	true, false
featureCheckLeaks	true, false
Logging	true, false
featureStatisticsLock	true, false
featureStatisticsDatabase	true, false
dummyFeatureLocking	true, false
featureLoggingSevere	true, false
featureCheckpointerDaemon	true, false
featureCustomizableCheckpointerTime	true, false
FStatistics	true, false
featureChecksum	true, false
featureEvictorDaemon	true, false
featureEvictor	true, false
featureFSync	true, false
featureEnvironmentLock	true, false
featureLoggingInfo	true, false
Checkpointer	true, false
NIO	true, false
BerkeleyDB	true, false
Chunked	true, false
featureDeleteDb	true, false
Persistency	true, false
featureStatisticsEnvLog	true, false
featureVerifier	true, false
EnvStats	true, false
FStatisticsFeatures	true, false

featureLookAheadCache	true, false
featureLoggingFine	true, false
featureTreeVisitor	true, false
Cleaner	true, false
FNIOType	true, false
featureStatisticsSequence	true, false
featureTruncateDb	true, false
featureNIO	true, false
featureMemoryBudget	true, false
featureCleanerDaemon	true, false
featureFileHandleCache	true, false

Restricciones:

- BerkeleyDb
- !BerkeleyDB || BerkeleyDb
- !BerkeleyDb || BerkeleyDB
- !FLogging || BerkeleyDB
- !FPersistency || BerkeleyDB
- !BerkeleyDB || FPersistency
- !FStatistics || BerkeleyDB
- !featureMemoryBudget || BerkeleyDB
- !FConcurrency || BerkeleyDB
- !FDbOperation || BerkeleyDB
- !FBtree || BerkeleyDB
- !BerkeleyDB || FBtree
- !BASE || BerkeleyDB
- !BerkeleyDB || BASE
- !Logging || FLogging
- !FLogging || Logging
- !Persistency || FPersistency
- !FPersistency || Persistency
- !Statistics || FStatistics
- !FStatistics || Statistics

- !featureLatch || FConcurrency
- !featureFSync || FConcurrency
- !featureTransaction || FConcurrency
- !dummyFeatureLocking || FConcurrency
- !featureCheckLeaks || FConcurrency
- !FConcurrency || featureLatch || featureFSync || featureTransaction || dummyFeatureLocking || featureCheckLeaks
- !featureDeleteDb || FDbOperation
- !featureTruncateDb || FDbOperation
- !FDbOperation || featureDeleteDb || featureTruncateDb
- !BTree || FBtree
- !FBtree || BTree
- !featureLoggingFile || Logging
- !featureLoggingConsole || Logging
- !featureLoggingDbLog || Logging
- !featureLoggingFinest || Logging
- !featureLoggingFiner || Logging
- !featureLoggingFine || Logging
- !featureLoggingInfo || Logging
- !featureLoggingConfig || Logging
- !featureLoggingSevere || Logging
- !featureLoggingBase || Logging
- !Logging || featureLoggingBase
- !FPersistencyFeatures || Persistency
- !FIOFeature || Persistency
- !Persistency || FIOFeature
- !FStatisticsFeatures || Statistics
- !Statistics || FStatisticsFeatures
- !featureStatisticsBase || Statistics
- !Statistics || featureStatisticsBase

- !featureVerifier || BTree
- !featureTreeVisitor || BTree
- !featureINCompressor || BTree
- !FEvictor || BTree
- !featureChecksum || FPersistencyFeatures
- !featureFileHandleCache || FPersistencyFeatures
- !featureHandleFullDiscError || FPersistencyFeatures
- !featureEnvironmentLock || FPersistencyFeatures
- !Checkpointer || FPersistencyFeatures
- !Cleaner || FPersistencyFeatures
- !FPersistencyFeatures || featureChecksum || featureFileHandleCache || featureHandleFullDiscError || featureEnvironmentLock || Checkpointer || Cleaner
- !NIO || FIOFeature
- !IO || FIOFeature
- !FIOFeature || NIO || IO
- !NIO || !IO
- !EnvStats || FStatisticsFeatures
- !featureStatisticsDatabase || FStatisticsFeatures
- !featureStatisticsLock || FStatisticsFeatures
- !featureStatisticsPreload || FStatisticsFeatures
- !featureStatisticsSequence || FStatisticsFeatures
- !featureStatisticsTransaction || FStatisticsFeatures
- !FStatisticsFeatures || EnvStats || featureStatisticsDatabase || featureStatisticsLock || featureStatisticsPreload || featureStatisticsSequence || featureStatisticsTransaction
- !Evictor || FEvictor
- !FEvictor || Evictor
- !featureCustomizableCheckpointerTime || Checkpointer
- !featureCustomizableCheckpointerBytes || Checkpointer
- !featureCheckpointerDaemon || Checkpointer
- !featureLookAheadCache || Cleaner
- !featureCleanerDaemon || Cleaner

- !featureDirectNIO || NIO
- !FNIOType || NIO
- !NIO || FNIOType
- !featureSynchronizedIO || IO
- !featureIO || IO
- !IO || featureIO
- !featureStatisticsEnvLog || EnvStats
- !featureStatisticsEnvINCompressor || EnvStats
- !featureStatisticsEnvFSync || EnvStats
- !featureStatisticsEnvEvictor || EnvStats
- !featureStatisticsEnvCleaner || EnvStats
- !featureStatisticsEnvCheckpointer || EnvStats
- !featureStatisticsEnvCaching || EnvStats
- !featureStatisticsEnvBase || EnvStats
- !EnvStats || featureStatisticsEnvBase
- !featureCriticalEviction || Evictor
- !featureEvictorDaemon || Evictor
- !featureEvictor || Evictor
- !Evictor || featureEvictor
- !NotChunked || FNIOType
- !Chunked || FNIOType
- !FNIOType || NotChunked || Chunked
- !NotChunked || !Chunked
- !featureNIO || NotChunked
- !NotChunked || featureNIO
- !featureChunkedNIO || Chunked
- !Chunked || featureChunkedNIO
- !featureEvictor || featureMemoryBudget
- !featureEvictorDaemon || featureMemoryBudget
- !featureLookAheadCache || featureMemoryBudget

- !featureStatisticsEnvCaching || featureMemoryBudget
- !featureCheckLeaks || featureStatisticsLock
- !featureCriticalEviction || featureINCompressor
- !featureCustomizableCheckpointerBytes || featureCustomizableCheckpointerTime
- !featureDeleteDb || dummyFeatureLocking
- !featureDeleteDb || featureEvictor
- !featureDeleteDb || featureINCompressor
- !featureDeleteDb || featureMemoryBudget
- !featureLatch || dummyFeatureLocking
- !featureLatch || featureCheckLeaks
- !featureLatch || featureDeleteDb
- !featureLatch || featureEvictor
- !featureLatch || featureFileHandleCache
- !featureLatch || featureFSync
- !featureLatch || featureINCompressor
- !featureLatch || featureMemoryBudget
- !featureLatch || featureStatisticsLock
- !featureLatch || featureTreeVisitor
- !featureLatch || featureTruncateDb
- !featureLatch || featureVerifier
- !featureLoggingSevere || featureEnvironmentLock
- !featureLoggingFine || dummyFeatureLocking
- !featureLoggingFine || featureEvictor
- !featureLoggingFine || featureINCompressor
- !featureLoggingInfo || featureChecksum
- !featureLoggingInfo || featureMemoryBudget
- !featureLoggingBase || featureTransaction
- !featureLoggingFinest || featureTransaction
- !featureMemoryBudget || featureEvictor
- !featureMemoryBudget || featureLatch

- !featureStatisticsLock || dummyFeatureLocking
- !featureStatisticsTransaction || dummyFeatureLocking
- !featureStatisticsEnvEvictor || featureEvictor
- !featureStatisticsEnvFSync || featureFSync
- !featureStatisticsEnvINCompressor || featureINCompressor
- !featureStatisticsTransaction || featureTransaction
- !featureStatisticsDatabase || featureTreeVisitor
- !featureTransaction || dummyFeatureLocking
- !featureTransaction || featureDeleteDb
- !featureTransaction || featureTruncateDb
- !featureTruncateDb || featureDeleteDb
- !featureVerifier || featureINCompressor
- !featureVerifier || featureTreeVisitor

C.4. Car

Nombre	Valores
parallel_parking	true, false
Automated_Driving_Controller	true, false
Car	true, false
enhanced_avoidance	true, false
Sensors	true, false
Standard_Avoidance	true, false
Collision_Avoidance_Braking	true, false
lateral_range_finder	true, false
forward_range_finder	true, false

Restricciones:

- Car
- !Automated_Driving_Controller || Car
- !Sensors || Car
- !Car || Sensors
- !Collision_Avoidance_Braking || Automated_Driving_Controller

- !Automated_Driving_Controller || Collision_Avoidance_Braking
- !parallel_parking || Automated_Driving_Controller
- !lateral_range_finder || Sensors
- !forward_range_finder || Sensors
- !Standard_Avoidance || Collision_Avoidance_Braking
- !enhanced_avoidance || Collision_Avoidance_Braking
- !Collision_Avoidance_Braking || Standard_Avoidance || enhanced_avoidance
- !Standard_Avoidance || !enhanced_avoidance
- !enhanced_avoidance || forward_range_finder
- !parallel_parking || lateral_range_finder

C.5. connector_fm

Nombre	Valores
Non_Queued	true, false
Non_blocking	true, false
Message_Based	true, false
Queued	true, false
Timeout	true, false
Local	true, false
Pull	true, false
Connector	true, false
Paradigm	true, false
Push	true, false
Synchronous	true, false
Sender	true, false
Blocking	true, false
Polling	true, false
Client_Server	true, false
Receiver	true, false
Asynchronous	true, false
Callback	true, false
CAN	true, false
Technology	true, false

Restricciones:

- Connector

- !Paradigm || Connector
- !Connector || Paradigm
- !Technology || Connector
- !Connector || Technology
- !Client_Server || Paradigm
- !Message_Based || Paradigm
- !Paradigm || Client_Server || Message_Based
- !Client_Server || !Message_Based
- !CAN || Technology
- !Local || Technology
- !Technology || CAN || Local
- !CAN || !Local
- !Synchronous || Client_Server
- !Asynchronous || Client_Server
- !Client_Server || Synchronous || Asynchronous
- !Synchronous || !Asynchronous
- !Sender || Message_Based
- !Message_Based || Sender
- !Receiver || Message_Based
- !Message_Based || Receiver
- !Timeout || Synchronous
- !Polling || Asynchronous
- !Callback || Asynchronous
- !Asynchronous || Polling || Callback
- !Polling || !Callback
- !Pull || Receiver
- !Push || Receiver
- !Receiver || Pull || Push
- !Pull || !Push
- !Blocking || Polling

- !Non_blocking || Polling
- !Polling || Blocking || Non_blocking
- !Blocking || !Non_blocking
- !Queued || Pull
- !Non_Queued || Pull
- !Pull || Queued || Non_Queued

C.6. fame_dbms_fm

Nombre	Valores
BPlus_Tree	true, false
LFU	true, false
famedbms	true, false
Put	true, false
Memory_Allocation	true, false
Static	true, false
OS	true, false
LRU	true, false
Unindexed	true, false
Debug_Logging	true, false
Storage	true, false
Page_Replication	true, false
In_Memory	true, false
API	true, false
Win	true, false
Nut_OS	true, false
Dynamic	true, false
Buffer_Manager	true, false
Delete	true, false
Get	true, false
Persistent	true, false

Restricciones:

- famedbms
- !OS || famedbms
- !famedbms || OS
- !Buffer_Manager || famedbms
- !famedbms || Buffer_Manager

- !Debug_Logging || famedbms
- !Storage || famedbms
- !famedbms || Storage
- !Nut_OS || OS
- !Win || OS
- !OS || Nut_OS || Win
- !Nut_OS || !Win
- !Persistent || Buffer_Manager
- !In_Memory || Buffer_Manager
- !Buffer_Manager || Persistent || In_Memory
- !Persistent || !In_Memory
- !API || Storage
- !Storage || API
- !BPlus_Tree || Storage
- !Unindexed || Storage
- !Storage || BPlus_Tree || Unindexed
- !BPlus_Tree || !Unindexed
- !Memory_Allocation || Persistent
- !Persistent || Memory_Allocation
- !Page_Replication || Persistent
- !Persistent || Page_Replication
- !Get || API
- !Put || API
- !Delete || API
- !Static || Memory_Allocation
- !Dynamic || Memory_Allocation
- !Memory_Allocation || Static || Dynamic
- !Static || !Dynamic
- !LRU || Page_Replication
- !LFU || Page_Replication
- !Page_Replication || LRU || LFU
- !LRU || !LFU

C.7. Gg4

Nombre	Valores
Base	true, false
Connected	true, false
HiddenGtp	true, false
HiddenWgt	true, false
Directed	true, false
Src	true, false
UndirectedOnlyVertices	true, false
DirectedWithEdges	true, false
WithEdges	true, false
Wgt	true, false
MSTPrim	true, false
DFS	true, false
BFS	true, false
MainGpl	true, false
WeightOptions	true, false
WeightedOnlyVertices	true, false
DirectedOnlyVertices	true, false
Undirected	true, false
DirectedWithNeighbors	true, false
Number	true, false
Weighted	true, false
Transpose	true, false
MSTKruskal	true, false
UndirectedWithEdges	true, false
WeightedWithEdges	true, false
WithNeighbors	true, false
Implementation	true, false
StronglyConnected	true, false
Gtp	true, false
GPL	true, false
StrongC	true, false
Alg	true, false
Cycle	true, false
OnlyVertices	true, false
Unweighted	true, false
TestProg	true, false
WeightedWithNeighbors	true, false
UndirectedWithNeighbors	true, false

Restricciones:

- GPL

- !MainGpl || GPL
- !GPL || MainGpl
- !TestProg || MainGpl
- !MainGpl || TestProg
- !Alg || MainGpl
- !MainGpl || Alg
- !Src || MainGpl
- !HiddenWgt || MainGpl
- !MainGpl || HiddenWgt
- !Wgt || MainGpl
- !MainGpl || Wgt
- !HiddenGtp || MainGpl
- !MainGpl || HiddenGtp
- !Gtp || MainGpl
- !MainGpl || Gtp
- !Implementation || MainGpl
- !MainGpl || Implementation
- !Base || MainGpl
- !MainGpl || Base
- !Number || Alg
- !Connected || Alg
- !StrongC || Alg
- !Cycle || Alg
- !MSTPrim || Alg
- !MSTKruskal || Alg
- !Alg || Number || Connected || StrongC || Cycle || MSTPrim || MSTKruskal
- !BFS || Src
- !DFS || Src
- !Src || BFS || DFS
- !BFS || !DFS

- !WeightOptions || HiddenWgt
- !HiddenWgt || WeightOptions
- !Weighted || Wgt
- !Unweighted || Wgt
- !Wgt || Weighted || Unweighted
- !Weighted || !Unweighted
- !DirectedWithEdges || HiddenGtp
- !DirectedWithNeighbors || HiddenGtp
- !DirectedOnlyVertices || HiddenGtp
- !UndirectedWithEdges || HiddenGtp
- !UndirectedWithNeighbors || HiddenGtp
- !UndirectedOnlyVertices || HiddenGtp
- !HiddenGtp || DirectedWithEdges || DirectedWithNeighbors || DirectedOnlyVertices || UndirectedWithEdges || UndirectedWithNeighbors || UndirectedOnlyVertices
- !DirectedWithEdges || !DirectedWithNeighbors
- !DirectedWithEdges || !DirectedOnlyVertices
- !DirectedWithEdges || !UndirectedWithEdges
- !DirectedWithEdges || !UndirectedWithNeighbors
- !DirectedWithEdges || !UndirectedOnlyVertices
- !DirectedWithNeighbors || !DirectedOnlyVertices
- !DirectedWithNeighbors || !UndirectedWithEdges
- !DirectedWithNeighbors || !UndirectedWithNeighbors
- !DirectedWithNeighbors || !UndirectedOnlyVertices
- !DirectedOnlyVertices || !UndirectedWithEdges
- !DirectedOnlyVertices || !UndirectedWithNeighbors
- !DirectedOnlyVertices || !UndirectedOnlyVertices
- !UndirectedWithEdges || !UndirectedWithNeighbors
- !UndirectedWithEdges || !UndirectedOnlyVertices
- !UndirectedWithNeighbors || !UndirectedOnlyVertices
- !Directed || Gtp

- !Undirected || Gtp
- !Gtp || Directed || Undirected
- !Directed || !Undirected
- !OnlyVertices || Implementation
- !WithNeighbors || Implementation
- !WithEdges || Implementation
- !Implementation || OnlyVertices || WithNeighbors || WithEdges
- !OnlyVertices || !WithNeighbors
- !OnlyVertices || !WithEdges
- !WithNeighbors || !WithEdges
- !Transpose || StrongC
- !StrongC || Transpose
- !StronglyConnected || StrongC
- !StrongC || StronglyConnected
- !WeightedWithEdges || WeightOptions
- !WeightedWithNeighbors || WeightOptions
- !WeightedOnlyVertices || WeightOptions
- !Number || Gtp
- !Number || Src
- !Connected || Undirected
- !Connected || Src
- !StrongC || Directed
- !StrongC || DFS
- !Cycle || Gtp
- !Cycle || DFS
- !MSTKruskal || Undirected
- !MSTKruskal || Weighted
- !MSTPrim || Undirected
- !MSTPrim || Weighted
- !MSTKruskal || !MSTPrim

- !OnlyVertices || !Weighted || WeightedOnlyVertices
- !WithNeighbors || !Weighted || WeightedWithNeighbors
- !WithEdges || !Weighted || WeightedWithEdges
- !OnlyVertices || !Directed || DirectedOnlyVertices
- !WithNeighbors || !Directed || DirectedWithNeighbors
- !WithEdges || !Directed || DirectedWithEdges
- !OnlyVertices || !Undirected || UndirectedOnlyVertices
- !WithNeighbors || !Undirected || UndirectedWithNeighbors
- !WithEdges || !Undirected || UndirectedWithEdges

C.8. Graph-product-line-fm

Nombre	Valores
cycle	true, false
shortest	true, false
weight	true, false
dfs	true, false
connected	true, false
stronglyc	true, false
number	true, false
bfs	true, false
mstprim	true, false
gtp	true, false
mstkruskal	true, false
gpl	true, false
undirected	true, false
weighted	true, false
benchmark	true, false
search	true, false
directed	true, false
unweighted	true, false
driver	true, false
algorithms	true, false

Restricciones:

- gpl
- !driver || gpl

- !gpl || driver
- !gtp || gpl
- !gpl || gtp
- !weight || gpl
- !search || gpl
- !algorithms || gpl
- !gpl || algorithms
- !benchmark || driver
- !driver || benchmark
- !directed || gtp
- !undirected || gtp
- !gtp || directed || undirected
- !directed || !undirected
- !weighted || weight
- !unweighted || weight
- !weight || weighted || unweighted
- !weighted || !unweighted
- !bfs || search
- !dfs || search
- !search || bfs || dfs
- !bfs || !dfs
- !number || algorithms
- !connected || algorithms
- !stronglyc || algorithms
- !cycle || algorithms
- !mstprim || algorithms
- !mstkruskal || algorithms
- !shortest || algorithms
- !algorithms || number || connected || stronglyc || cycle || mstprim || mstkruskal || shortest
- !number || bfs

- !number || dfs
- !connected || undirected
- !connected || weighted
- !stronglyc || directed
- !stronglyc || dfs
- !cycle || dfs
- !mstkruskal || undirected
- !mstkruskal || unweighted
- !mstprim || undirected
- !mstprim || unweighted
- !mstkruskal || !mstprim
- !shortest || directed
- !shortest || unweighted

C.9. movies_app_fm

Nombre	Valores
Local	true, false
Bluetooth	true, false
Rich	true, false
DB	true, false
Thin	true, false
Built_In	true, false
Movies_App	true, false
GUI	true, false
Network	true, false
Remote	true, false
Cache_Policy	true, false
GPS	true, false
Wifi	true, false

Restricciones:

- Movies_App
- !GUI || Movies_App
- !Movies_App || GUI

- !DB || Movies_App
- !Movies_App || DB
- !Network || Movies_App
- !Movies_App || Network
- !GPS || Movies_App
- !Thin || GUI
- !Rich || GUI
- !GUI || Thin || Rich
- !Thin || !Rich
- !Local || DB
- !Remote || DB
- !DB || Local || Remote
- !Local || !Remote
- !Wifi || Network
- !Bluetooth || Network
- !Network || Wifi || Bluetooth
- !Wifi || !Bluetooth
- !Built_In || GPS
- !Cache_Policy || Remote
- !Remote || Cache_Policy

C.10. REAL-FM-12

Nombre	Valores
_id_0	true, false
svg	true, false
_id_2	true, false
_id_1	true, false
_id_11	true, false
_id_3	true, false
_id_10	true, false
page_translation	true, false
_id_5	true, false
_id_6	true, false

_id_7	true, false
search_by_language	true, false
page_preview	true, false
_id_9	true, false

Restricciones:

- _id_0
- !page_translation || _id_0
- !_id_1 || _id_0
- !_id_0 || _id_1
- !search_by_language || _id_0
- !page_preview || _id_0
- !_id_2 || _id_1
- !_id_1 || _id_2
- !_id_3 || _id_1
- !_id_7 || _id_1
- !_id_9 || search_by_language
- !_id_10 || search_by_language
- !_id_11 || search_by_language
- !search_by_language || _id_9 || _id_10 || _id_11
- !_id_5 || _id_3
- !_id_6 || _id_3
- !svg || _id_3
- !_id_3 || _id_5 || _id_6 || svg
- !_id_5 || !_id_6
- !_id_5 || !svg
- !_id_6 || !svg
- !search_by_language || page_translation
- !page_preview || !svg

C.11. smart_home_fm

Nombre	Valores
Other_Group	true, false
Siren	true, false
Bell	true, false
Card_Reader	true, false
Environment_Control	true, false
_Fahrenheit	true, false
Measurement_Units	true, false
Touch_Screen	true, false
First_Aid_Group	true, false
Fire_Alarm	true, false
Manual_Windows	true, false
Automatic_Windows	true, false
Smart_Home	true, false
Mobile	true, false
Presence_Simulator	true, false
Light	true, false
Windows_Management	true, false
Music_Simulation	true, false
Simple_Control	true, false
Smart_Heating_Management	true, false
Smart_Light_Management	true, false
Fire_Control	true, false
_Celsius	true, false
Heating_Management	true, false
Temperature_Management	true, false
GUI	true, false
Light_Management	true, false
Pre_defined_Values	true, false
Fingerprint_Reader	true, false
Door_Lock	true, false
Blind_Simulation	true, false
Keypad_Reader	true, false
Fire_Department	true, false
Internet	true, false
Light_Simulation	true, false

Restricciones:

- Smart_Home
- !Environment_Control || Smart_Home
- !Smart_Home || Environment_Control

- !Light_Management || Smart_Home
- !Smart_Home || Light_Management
- !GUI || Smart_Home
- !Smart_Home || GUI
- !Presence_Simulator || Smart_Home
- !Smart_Home || Presence_Simulator
- !Fire_Control || Smart_Home
- !Smart_Home || Fire_Control
- !Door_Lock || Smart_Home
- !Smart_Home || Door_Lock
- !Measurement_Units || Environment_Control
- !Environment_Control || Measurement_Units
- !Temperature_Management || Environment_Control
- !Environment_Control || Temperature_Management
- !Windows_Management || Environment_Control
- !Environment_Control || Windows_Management
- !Smart_Light_Management || Light_Management
- !Pre_defined_Values || Light_Management
- !Simple_Control || Light_Management
- !Light_Management || Simple_Control
- !Touch_Screen || GUI
- !Mobile || GUI
- !Internet || GUI
- !Light_Simulation || Presence_Simulator
- !Blind_Simulation || Presence_Simulator
- !Music_Simulation || Presence_Simulator
- !First_Aid_Group || Fire_Control
- !Fire_Control || First_Aid_Group
- !Fire_Alarm || Fire_Control
- !Fire_Control || Fire_Alarm

- !Keypad_Reader || Door_Lock
- !Card_Reader || Door_Lock
- !Fingerprint_Reader || Door_Lock
- !_Celsius || Measurement_Units
- !_Fahrenheit || Measurement_Units
- !Heating_Management || Temperature_Management
- !Temperature_Management || Heating_Management
- !Smart_Heating_Management || Temperature_Management
- !Manual_Windows || Windows_Management
- !Windows_Management || Manual_Windows
- !Automatic_Windows || Windows_Management
- !Fire_Department || First_Aid_Group
- !Other_Group || First_Aid_Group
- !Siren || Fire_Alarm
- !Bell || Fire_Alarm
- !Light || Fire_Alarm

C.12. stack_fm

Nombre	Valores
Stack	true, false
Float	true, false
Element_Type	true, false
Memory_Usage	true, false
Value	true, false
Additional_Features	true, false
Type_Check	true, false
Counter	true, false
Bounds_Check	true, false
Speed	true, false
String	true, false
Dynamic	true, false
Integer	true, false
Thread_Safety	true, false
Optimization	true, false
Fixed	true, false
Size	true, false

Restricciones:

- Stack
- !Size || Stack
- !Element_Type || Stack
- !Stack || Element_Type
- !Optimization || Stack
- !Counter || Stack
- !Additional_Features || Stack
- !Fixed || Size
- !Dynamic || Size
- !Size || Fixed || Dynamic
- !Fixed || !Dynamic
- !Integer || Element_Type
- !Float || Element_Type
- !String || Element_Type
- !Element_Type || Integer || Float || String
- !Integer || !Float
- !Integer || !String
- !Float || !String
- !Speed || Optimization
- !Memory_Usage || Optimization
- !Optimization || Speed || Memory_Usage
- !Speed || !Memory_Usage
- !Thread_Safety || Additional_Features
- !Bounds_Check || Additional_Features
- !Type_Check || Additional_Features
- !Additional_Features || Thread_Safety || Bounds_Check || Type_Check
- !Value || Fixed
- !Fixed || Value

C.13. TightVNC

Nombre	Valores
VncViewer	true, false
Base	true, false
Opt	true, false
Type	true, false
OptionsButtonFeat	true, false
OpShareFeat	true, false
OpMouse23Feat	true, false
OpCopyRectFeat	true, false
OpCursorShapeFeat	true, false
RefreshButtonFeat	true, false
AboutButtonFeat	true, false
RecordingFeat	true, false
OpRestrictedColorsFeat	true, false
OptionsFeat	true, false
MOpt	true, false
BOpt	true, false
DisconnectButtonFeat	true, false
ButtonFeature	true, false
OpEncodingFeat	true, false
OpViewFeat	true, false
OptionsMenuFeatures	true, false
VncViewerMain	true, false
RecordButtonFeat	true, false
OpJPEGqualityFeat	true, false
CtrlAltDelButtonFeat	true, false
VncViewerType	true, false
AltTabButtonFeat	true, false
OpCompressionFeat	true, false
ClipboardFeat	true, false
ClipboardButtonFeat	true, false

Restricciones:

- VncViewer
- !VncViewerMain || VncViewer
- !VncViewer || VncViewerMain
- !Type || VncViewerMain
- !VncViewerMain || Type
- !Base || VncViewerMain

- !VncViewerMain || Base
- !VncViewerType || Type
- !Type || VncViewerType
- !Opt || VncViewerType
- !OptionsMenuFeatures || Opt
- !ButtonFeature || Opt
- !ClipboardFeat || Opt
- !RecordingFeat || Opt
- !Opt || OptionsMenuFeatures || ButtonFeature || ClipboardFeat || RecordingFeat
- !MOpt || OptionsMenuFeatures
- !OptionsFeat || OptionsMenuFeatures
- !OptionsMenuFeatures || OptionsFeat
- !BOpt || ButtonFeature
- !OpEncodingFeat || MOpt
- !OpCompressionFeat || MOpt
- !OpJPEGqualityFeat || MOpt
- !OpCursorShapeFeat || MOpt
- !OpCopyRectFeat || MOpt
- !OpRestrictedColorsFeat || MOpt
- !OpMouse23Feat || MOpt
- !OpViewFeat || MOpt
- !OpShareFeat || MOpt
- !MOpt || OpEncodingFeat || OpCompressionFeat || OpJPEGqualityFeat || OpCursorShapeFeat || OpCopyRectFeat || OpRestrictedColorsFeat || OpMouse23Feat || OpViewFeat || OpShareFeat
- !AboutButtonFeat || BOpt
- !AltTabButtonFeat || BOpt
- !RefreshButtonFeat || BOpt
- !CtrlAltDelButtonFeat || BOpt
- !RecordButtonFeat || BOpt
- !ClipboardButtonFeat || BOpt

- !OptionsButtonFeat || BOpt
- !DisconnectButtonFeat || BOpt
- !BOpt || AboutButtonFeat || AltTabButtonFeat || RefreshButtonFeat || CtrlAltDelButtonFeat || RecordButtonFeat || ClipboardButtonFeat || OptionsButtonFeat || DisconnectButtonFeat
- !RecordButtonFeat || RecordingFeat
- !OptionsButtonFeat || OptionsFeat
- !ClipboardButtonFeat || ClipboardFeat

C.14. Violet

Nombre	Valores
VioletDef	true, false
SequenceDiagram	true, false
StateTransitionEdge	true, false
ClassInterfaceEdge	true, false
InitialStateNode	true, false
ExtensionFilter	true, false
SequenceD	true, false
DiagramSupport	true, false
ObjectFieldNode	true, false
SelectNext	true, false
NewFile	true, false
Window	true, false
CloseWindow	true, false
ClipDrawingArea	true, false
SetTitle	true, false
ExtendRelationshipEdge	true, false
EditProperties	true, false
LookAndFeel	true, false
StateDiagram	true, false
ObjectD	true, false
ClassAssociationEdge	true, false
InterfaceNode	true, false
UseCaseAssociationEdge	true, false
ObjectNoteEdge	true, false
MenuResources	true, false
RecentFile	true, false
ObjectDiagram	true, false
ObjectNode	true, false
StateD	true, false
StateNode	true, false

VioletFilter	true, false
NextWindow	true, false
SmallerGrid	true, false
HideGrid	true, false
SequenceParameterNode	true, false
WindowMenu	true, false
UseCaseNoteEdge	true, false
ZoomOut	true, false
GraphUtility	true, false
ClassDependencyEdge	true, false
EditMenu	true, false
SequenceCallEdge	true, false
SaveAs	true, false
ImageFilter	true, false
License	true, false
base	true, false
Additional	true, false
UseCaseDiagram	true, false
StateNoteEdge	true, false
Exit	true, false
StateNoteNode	true, false
ClassAggregationEdge	true, false
GrowDrawingArea	true, false
FileMenu	true, false
InternalFrame	true, false
UseCaseD	true, false
Actor	true, false
FinalStateNode	true, false
VersionChecker	true, false
UseCaseNode	true, false
About	true, false
ClassDiagram	true, false
SequenceCallNode	true, false
ObjectNoteNode	true, false
ObjectReferenceEdge	true, false
RestoreWindow	true, false
CommandLine	true, false
SequenceNoteEdge	true, false
Print	true, false
ZoomIn	true, false
Read	true, false
MaximizeWindow	true, false
SequenceNoteNode	true, false
SequenceReturnEdge	true, false
ClassNoteNode	true, false
IncludeRelationshipEdge	true, false

ClassNoteEdge	true, false
View	true, false
Help	true, false
HelpMenu	true, false
SaveFile	true, false
Preferences	true, false
ExportImage	true, false
OpenFile	true, false
DeleteItem	true, false
PackageNode	true, false
ClassD	true, false
PreviousWindow	true, false
ClassCompositionEdge	true, false
Edit	true, false
SelectPrevious	true, false
File	true, false
ClassInheritanceEdge	true, false
ViewMenu	true, false
UseCaseNoteNode	true, false
ClassNode	true, false
Violet	true, false
UseCaseGeneralizationEdge	true, false
ObjectAssociationEdge	true, false
LargerGrid	true, false
FileUtility	true, false

Restricciones:

- Violet
- !VioletDef || Violet
- !Violet || VioletDef
- !Additional || VioletDef
- !Help || VioletDef
- !HelpMenu || VioletDef
- !Window || VioletDef
- !WindowMenu || VioletDef
- !View || VioletDef
- !ViewMenu || VioletDef
- !Edit || VioletDef

- !EditMenu || VioletDef
- !ClassDiagram || VioletDef
- !ClassD || VioletDef
- !SequenceDiagram || VioletDef
- !SequenceD || VioletDef
- !StateDiagram || VioletDef
- !StateD || VioletDef
- !ObjectDiagram || VioletDef
- !ObjectD || VioletDef
- !UseCaseDiagram || VioletDef
- !UseCaseD || VioletDef
- !DiagramSupport || VioletDef
- !File || VioletDef
- !NewFile || VioletDef
- !FileMenu || VioletDef
- !Read || VioletDef
- !InternalFrame || VioletDef
- !FileUtility || VioletDef
- !ExtensionFilter || VioletDef
- !GraphUtility || VioletDef
- !MenuResources || VioletDef
- !base || VioletDef
- !VioletDef || base
- !CommandLine || Additional
- !VersionChecker || Additional
- !VioletFilter || Additional
- !ImageFilter || Additional
- !SetTitle || Additional
- !Preferences || Additional

- !Additional || CommandLine || VersionChecker || VioletFilter || ImageFilter || SetTitle || Preferences
- !About || Help
- !License || Help
- !Help || About || License
- !CloseWindow || Window
- !RestoreWindow || Window
- !MaximizeWindow || Window
- !PreviousWindow || Window
- !NextWindow || Window
- !Window || CloseWindow || RestoreWindow || MaximizeWindow || PreviousWindow || NextWindow
- !LookAndFeel || View
- !ZoomOut || View
- !ZoomIn || View
- !GrowDrawingArea || View
- !ClipDrawingArea || View
- !SmallerGrid || View
- !LargerGrid || View
- !HideGrid || View
- !View || LookAndFeel || ZoomOut || ZoomIn || GrowDrawingArea || ClipDrawingArea || SmallerGrid || LargerGrid || HideGrid
- !EditProperties || Edit
- !DeleteItem || Edit
- !SelectNext || Edit
- !SelectPrevious || Edit
- !Edit || EditProperties || DeleteItem || SelectNext || SelectPrevious
- !ClassNode || ClassDiagram
- !InterfaceNode || ClassDiagram
- !PackageNode || ClassDiagram
- !ClassNoteNode || ClassDiagram

- !ClassDependencyEdge || ClassDiagram
- !ClassInheritanceEdge || ClassDiagram
- !ClassAggregationEdge || ClassDiagram
- !ClassAssociationEdge || ClassDiagram
- !ClassCompositionEdge || ClassDiagram
- !ClassInterfaceEdge || ClassDiagram
- !ClassNoteEdge || ClassDiagram
- !ClassDiagram || ClassNode || InterfaceNode || PackageNode || ClassNoteNode || ClassDependencyEdge || ClassInheritanceEdge || ClassAggregationEdge || ClassAssociationEdge || ClassCompositionEdge || ClassInterfaceEdge || ClassNoteEdge
- !SequenceParameterNode || SequenceDiagram
- !SequenceCallNode || SequenceDiagram
- !SequenceNoteNode || SequenceDiagram
- !SequenceCallEdge || SequenceDiagram
- !SequenceReturnEdge || SequenceDiagram
- !SequenceNoteEdge || SequenceDiagram
- !SequenceDiagram || SequenceParameterNode || SequenceCallNode || SequenceNoteNode || SequenceCallEdge || SequenceReturnEdge || SequenceNoteEdge
- !StateNode || StateDiagram
- !InitialStateNode || StateDiagram
- !FinalStateNode || StateDiagram
- !StateNoteNode || StateDiagram
- !StateTransitionEdge || StateDiagram
- !StateNoteEdge || StateDiagram
- !StateDiagram || StateNode || InitialStateNode || FinalStateNode || StateNoteNode || StateTransitionEdge || StateNoteEdge
- !ObjectNode || ObjectDiagram
- !ObjectFieldNode || ObjectDiagram
- !ObjectNoteNode || ObjectDiagram
- !ObjectReferenceEdge || ObjectDiagram
- !ObjectAssociationEdge || ObjectDiagram

- !ObjectNoteEdge || ObjectDiagram
- !ObjectDiagram || ObjectNode || ObjectFieldNode || ObjectNoteNode || ObjectReferenceEdge || ObjectAssociationEdge || ObjectNoteEdge
- !Actor || UseCaseDiagram
- !UseCaseNode || UseCaseDiagram
- !UseCaseNoteNode || UseCaseDiagram
- !UseCaseAssociationEdge || UseCaseDiagram
- !ExtendRelationshipEdge || UseCaseDiagram
- !IncludeRelationshipEdge || UseCaseDiagram
- !UseCaseGeneralizationEdge || UseCaseDiagram
- !UseCaseNoteEdge || UseCaseDiagram
- !UseCaseDiagram || Actor || UseCaseNode || UseCaseNoteNode || UseCaseAssociationEdge || ExtendRelationshipEdge || IncludeRelationshipEdge || UseCaseGeneralizationEdge || UseCaseNoteEdge
- !Exit || File
- !Print || File
- !ExportImage || File
- !SaveFile || File
- !SaveAs || File
- !RecentFile || File
- !OpenFile || File
- !File || Exit || Print || ExportImage || SaveFile || SaveAs || RecentFile || OpenFile
- !File || MenuResources
- !File || GraphUtility
- !File || ExtensionFilter
- !File || FileUtility
- !File || InternalFrame
- !File || Read
- !NewFile || MenuResources
- !NewFile || GraphUtility
- !NewFile || ExtensionFilter

- !NewFile || FileUtility
- !NewFile || InternalFrame
- !NewFile || Read
- !Edit || MenuResources
- !Edit || GraphUtility
- !Edit || ExtensionFilter
- !Edit || FileUtility
- !Edit || InternalFrame
- !Edit || Read
- !View || MenuResources
- !View || GraphUtility
- !View || ExtensionFilter
- !View || FileUtility
- !View || InternalFrame
- !View || Read
- !Window || MenuResources
- !Window || GraphUtility
- !Window || ExtensionFilter
- !Window || FileUtility
- !Window || InternalFrame
- !Window || Read
- !Help || MenuResources
- !Help || GraphUtility
- !Help || ExtensionFilter
- !Help || FileUtility
- !Help || InternalFrame
- !Help || Read
- !Additional || MenuResources
- !Additional || GraphUtility
- !Additional || ExtensionFilter

- !Additional || FileUtility
- !Additional || InternalFrame
- !Additional || Read
- !File || FileMenu
- !NewFile || FileMenu
- !Edit || EditMenu
- !View || ViewMenu
- !Window || WindowMenu
- !Help || HelpMenu
- !ClassDiagram || DiagramSupport
- !ClassDiagram || NewFile
- !SequenceDiagram || DiagramSupport
- !SequenceDiagram || NewFile
- !StateDiagram || DiagramSupport
- !StateDiagram || NewFile
- !ObjectDiagram || DiagramSupport
- !ObjectDiagram || NewFile
- !UseCaseDiagram || DiagramSupport
- !UseCaseDiagram || NewFile
- !ClassDiagram || ClassD
- !SequenceDiagram || SequenceD
- !StateDiagram || StateD
- !ObjectDiagram || ObjectD
- !UseCaseDiagram || UseCaseD
- !RecentFile || OpenFile
- !SaveFile || SaveAs
- !ImageFilter || ExtensionFilter
- !ImageFilter || ExportImage
- !VioletFilter || ExtensionFilter
- !CommandLine || OpenFile

- !ClassDependencyEdge || ClassNode
- !ClassInheritanceEdge || ClassNode
- !ClassAggregationEdge || ClassNode
- !ClassAssociationEdge || ClassNode
- !ClassCompositionEdge || ClassNode
- !ClassInterfaceEdge || ClassNode
- !ClassNoteEdge || ClassNoteNode
- !SequenceCallEdge || SequenceCallNode
- !SequenceReturnEdge || SequenceCallNode
- !SequenceNoteEdge || SequenceNoteNode
- !StateTransitionEdge || StateNode
- !StateNoteEdge || StateNoteNode
- !ObjectReferenceEdge || ObjectNode
- !ObjectAssociationEdge || ObjectNode
- !ObjectNoteEdge || ObjectNoteNode
- !UseCaseAssociationEdge || Actor
- !ExtendRelationshipEdge || Actor
- !IncludeRelationshipEdge || Actor
- !UseCaseGeneralizationEdge || Actor
- !UseCaseNoteEdge || UseCaseNoteNode