NIST SP 800-142

PRUEBAS COMBINATORIAS PRÁCTICAS

D. Richard Kuhn, Raghu N. Kacker, Yu Lei

Editora técnica: Itzel Domínguez Mendoza

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

Informes Sobre Tecnología de Sistemas Informáticos

El Laboratorio de Tecnología de la Información (ITL) del Instituto Nacional de Estándares y Tecnología (NIST) promueve la economía y el bienestar público de los EE. UU., proporcionando liderazgo técnico para la infraestructura de medición y estándares de la nación. ITL desarrolla pruebas, métodos de prueba, datos de referencia, implementaciones de prueba de concepto y análisis técnicos para avanzar en el desarrollo y el uso productivo de la tecnología de la información. Las responsabilidades de ITL incluyen el desarrollo de estándares y pautas técnicas, físicas, administrativas y de gestión para la seguridad y privacidad rentables de la información confidencial no clasificada en los sistemas informáticos federales. Esta serie de publicaciones especiales (serie 800) informa sobre los esfuerzos de investigación, orientación y divulgación de ITL en seguridad informática, y sus actividades de colaboración con la industria, el gobierno y las organizaciones académicas.

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 2010

For sale by the Superintendent of Documents, U.S. Government Printing Office
Internet: bookstore.gpo.gov — Phone: (202) 512-1800 — Fax: (202) 512-2250
Mail: Stop SSOP, Washington, DC 20402-0001Note to Readers

Nota para los Lectores

Este documento es una publicación del Instituto Nacional de Estándares y Tecnología (NIST) y no está sujeto a los derechos de autor de EE. UU. Ciertas entidades comerciales, equipos o materiales pueden identificarse en este documento para describir adecuadamente un procedimiento o concepto experimental. Dicha identificación no pretende implicar recomendación o respaldo por parte del Instituto Nacional de Estándares y Tecnología, ni pretende implicar que las entidades, materiales o equipos son necesariamente los mejores disponibles para el propósito.

Este documento es una traducción de "Practical Combinatorial Testing", NIST SP 800-142, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

Si tiene preguntas o comentarios sobre este documento, comuníquese con Rick Kuhn, kuhn@nist.gov o 301-975-3337.

Agradecimientos

Agradecimientos especiales a Tim Grance, Jim Higdon, Eduardo Miranda y Tom Wissink por el apoyo inicial y la evangelización de este trabajo, y especialmente a Jim Lawrence, quien ha sido una parte integral del equipo desde el principio. Nos hemos beneficiado enormemente de las interacciones con investigadores y profesionales, incluidos Renee Bryce, Myra Cohen, Charles Colbourn, Mike Ellims, Vincent Hu, Justin Hunter, Aditya Mathur, Josh Maximoff, Carmelo Montanez-Rivera, Jenise Reyes Rodriguez, Rick Rivello, Sreedevi Sampath, Mike Trela y Tao Xie. También agradecemos a los estudiantes de NIST SURF Michael Forbes, William Goh, Evan Hartig, Menal Modha, Kimberley O'Brien-Applegate, Michael Reilly, Malcolm Taylor y Bryan Wilkinson, quienes contribuyeron con el software y los métodos descritos en este documento.

Tabla de contenidos

Resumen Ejecutivo

Los errores de implementación de software son uno de los contribuyentes más importantes a las vulnerabilidades de seguridad de los sistemas de información, lo que hace que las pruebas de software sean una parte esencial de la garantía de los mismo. En 2003, el NIST publicó un informe, ampliamente citado, que estimaba que las pruebas de software inadecuadas le cuestan a la economía estadounidense 59 500 millones de dólares al año, aunque entre el 50 % y el 80 % de los presupuestos de desarrollo se dedican a las pruebas. Las pruebas exhaustivas (probar todas las combinaciones posibles de entradas y rutas de ejecución) son imposibles para el software del mundo real, por lo que el software de alta seguridad se prueba utilizando métodos que requieren mucho tiempo del personal y, por lo tanto, tienen un costo enorme. Para el software menos crítico, las restricciones presupuestarias a menudo limitan la cantidad de pruebas que se pueden realizar, lo que aumenta el riesgo de errores residuales que conducen a fallas del sistema y debilidades de seguridad.

Las pruebas combinatorias son un método que puede reducir costos y aumentar la efectividad de las pruebas de software para muchas aplicaciones. La idea clave que subyace a esta forma de prueba es que no todos los parámetros contribuyen a todas las fallas y la mayoría de las fallas son causadas por interacciones entre relativamente pocos parámetros. Los datos empíricos recopilados por NIST y otros sugieren que las fallas de software son provocadas por solo unas pocas variables que interactúan (6 o menos). Este hallazgo tiene implicaciones importantes para las pruebas porque sugiere que probar combinaciones de parámetros puede proporcionar una detección de fallas altamente efectiva. Las pruebas por pares (combinaciones de 2 vías) a veces se usan para obtener resultados razonablemente buenos a bajo costo, pero las pruebas por pares pueden pasar por alto del 10% al 40% o más de los errores del sistema y, por lo tanto, no son suficientes para el software de misión crítica. Las pruebas combinatorias más allá de 2 vías han sido limitadas, principalmente debido a la falta de buenos algoritmos para niveles de interacción más altos, como las pruebas de 4 a 6 vías. Sin embargo, los nuevos algoritmos han hecho que las pruebas combinatorias vayan más allá de la práctica por pares para el uso industrial.

Esta publicación proporciona un tutorial independiente sobre el uso de pruebas combinatorias para software del mundo real. Presenta los conceptos y métodos clave, explica el uso de herramientas de software para generar pruebas combinatorias (disponibles gratuitamente en el sitio web del NIST csrc.nist.gov/acts) y analiza temas avanzados como el uso de modelos formales de software para determinar los resultados esperados para cada conjunto de entradas de prueba. Con cada tema, una sección sobre costos y consideraciones prácticas explica las ventajas y desventajas y las limitaciones que pueden afectar los recursos o la financiación. El material es accesible para un estudiante de licenciatura en ciencias de la computación o ingeniería, e incluye un extenso conjunto de referencias a artículos que brindan mayor profundidad en cada tema.

1 INTRODUCCIÓN

Los errores de implementación de software son uno de los contribuyentes más importantes a las vulnerabilidades de seguridad de un sistema de información, lo que hace que las pruebas de software sean una parte esencial de la garantía del mismo. Los métodos combinatorios pueden ayudar a reducir el costo y aumentar la efectividad de las pruebas de software para muchas aplicaciones. Esta publicación proporciona un tutorial independiente sobre el uso de pruebas combinatorias para software del mundo real. Introduce los conceptos y métodos clave, explica el uso de herramientas de software para generar pruebas combinatorias (disponibles gratuitamente en el sitio web del NIST csrc.nist.gov/acts) y analiza temas avanzados como el uso de modelos formales de software para determinar los resultados esperados para cada conjunto posible de entradas de prueba. El material es accesible para un estudiante de licenciatura en ciencias de la computación o ingeniería, e incluye un extenso conjunto de referencias a artículos que brindan mayor profundidad en cada tema.

1.1 Autoridad

El Instituto Nacional de Estándares y Tecnología (NIST) desarrolló este documento en cumplimiento de sus responsabilidades estatutarias bajo la Ley Federal de Administración de Seguridad de la Información (FISMA) de 2002, Ley Pública 107-347.

NIST es responsable de desarrollar estándares y pautas, incluidos los requisitos mínimos, para brindar seguridad de la información adecuada para todas las operaciones y activos de la agencia, pero dichos estándares y pautas no se aplicarán a los sistemas de seguridad nacional. Esta pauta es consistente con los requisitos de la Circular A-130 de la Oficina de Administración y Presupuesto (OMB), Sección 8b(3), “Sistemas de información de la agencia de seguridad”, como se analiza en A-130, Apéndice IV: Análisis de secciones clave. Se proporciona información complementaria en A-130, Apéndice III.

Esta guía ha sido preparada para uso de las agencias federales. Puede ser utilizado por organizaciones no gubernamentales de forma voluntaria y no está sujeto a derechos de autor, aunque se desea la atribución. Nada de lo contenido en este documento debe tomarse en contradicción con las normas y pautas que la Secretaría de Comercio hizo obligatorias y vinculantes para las agencias federales en virtud de la autoridad legal, ni estas pautas deben interpretarse como que alteran o reemplazan las autoridades existentes de la Secretaría de Comercio, Director de la OMB, o cualquier otro funcionario federal.

1.2 Alcance y Propósito del Documento

Esta publicación presenta las pruebas combinatorias y explica cómo usarlas de manera efectiva para la garantía de sistemas y software.

1.3 Audiencia y Suposiciones

Este documento asume que los lectores tienen experiencia con el desarrollo y las pruebas de software, cierta familiaridad con los lenguajes de secuencias de comandos y conocimientos básicos de programación, lógica y matemáticas discretas equivalentes a los que se adquirirían en un programa de licenciatura en informática o ingeniería. La mayor parte del material debe ser entendido fácilmente por un estudiante universitario con alguna experiencia en programación. Debido a la naturaleza en

constante cambio de la industria de la tecnología de la información, se recomienda encarecidamente a los lectores que aprovechen otros recursos (incluidos los que se enumeran en este documento) para obtener información más actualizada y detallada.

1.4 Organización: Cómo Utilizar este Documento

El documento está dividido en capítulos, con material de referencia cubierto en apéndices. Debido a que se pretende que sea independiente, cada capítulo proporciona material que se utilizará en temas posteriores. La mayoría de los evaluadores necesitarán los capítulos 2, 3 y 4, mientras que el material de los capítulos posteriores se especializa en varios temas. Los apéndices incluyen una revisión de la combinatoria básica y una discusión de datos empíricos sobre fallas de software.

Los lectores nuevos en las pruebas combinatorias pueden querer revisar los conceptos básicos de combinatoria en el Apéndice A y leer los capítulos 2, 3 y 4. Se pueden reservar otras secciones de la publicación para uso posterior según sea necesario.

2 MÉTODOS COMBINATORIOS EN PRUEBAS

Los desarrolladores de software con gran cantidad de datos a menudo notan un fenómeno interesante, aunque no sorprendente: cuando el uso de una aplicación aumenta drásticamente, los componentes que han funcionado durante meses sin problemas de repente desarrollan errores no detectados anteriormente. Por ejemplo, es posible que la aplicación se haya instalado en una plataforma de redes DBMS-hardware-OS diferente, o que los clientes agregados recientemente tengan registros de cuenta con una combinación extraña de valores que no se han producido antes. Algunas de estas raras combinaciones desencadenan fallas que han escapado a las pruebas anteriores y al uso extensivo. Tales fallas se conocen como fallas de interacción, porque solo se exponen cuando dos o más valores de entrada interactúan para hacer que el programa alcance un resultado incorrecto.

Las pruebas combinatorias pueden ayudar a detectar problemas como este en las primeras etapas del ciclo de vida de las pruebas. La idea clave que subyace a las pruebas combinatorias t-way es que no todos los parámetros contribuyen a todas las fallas y la mayoría de las fallas se desencadenan por un solo valor de parámetro o por interacciones entre una cantidad relativamente pequeña de parámetros (para obtener más información sobre la cantidad de parámetros que interactúan en las fallas, consulte Apéndice B). Para detectar fallas en la interacción, los desarrolladores de software a menudo usan "pruebas por pares", en las que todos los pares posibles de valores de parámetros están cubiertos por al menos una prueba. Su efectividad se basa en la observación de que las fallas del software a menudo involucran interacciones entre parámetros. Por ejemplo, se puede observar que un enrutador falla solo para un protocolo en particular cuando el volumen del paquete excede una determinada tasa, una interacción bidireccional entre el tipo de protocolo y la tasa de paquetes. La Figura 1 ilustra cómo puede ocurrir una interacción bidireccional de este tipo en el código. Tenga en cuenta que la falla solo se activará cuando tanto la *presión < 10 como el volumen > 300* sean verdaderos.

```
if (pressure < 10) {
      // do something
      if (volume > 300)  {
            faulty code!  BOOM!
      }
      else {
            good code, no problem
      }
}
else {
      // do something else
}
```

Figure 1. Falla de interacción bidireccional que se activa solo cuando se cumplen dos condiciones.

Las pruebas por pares pueden ser muy efectivas y hay buenas herramientas disponibles para generar matrices con todos los pares de combinaciones de valores de parámetros. Pero hasta hace poco, solo un puñado de herramientas podía generar combinaciones más allá de 2 vías, y la mayoría de las que lo hacían requerían tiempos demasiado largos para generar matrices de 3, 4 o 5 vías porque el proceso de generación es matemáticamente complejo. Las pruebas por pares, es decir, las

combinaciones de 2 vías se han aceptado como el enfoque común para las pruebas combinatorias porque son computacionalmente manejables y razonablemente efectivos.

Las fallas parecen ser causadas por interacciones de solo unas pocas variables, por lo que las pruebas que cubren todas esas interacciones de pocas variables pueden ser muy efectivas.

Pero ¿qué pasa si alguna falla se desencadena solo por una combinación muy inusual de 3, 4 o más valores de sensor? Es muy poco probable que las pruebas por pares detecten este caso inusual; necesitaríamos probar combinaciones de valores de 3 y 4 vías. Pero ¿es suficiente probar todas las combinaciones de 4 vías para detectar todos los errores? ¿Qué grado de interacción ocurre en fallas reales en sistemas reales? Sorprendentemente, esta pregunta no había sido estudiada cuando el NIST comenzó a investigar las fallas de interacción en 1999. Los resultados mostraron que, en una variedad de dominios, todas las fallas podrían desencadenarse por un máximo de interacciones de 4 a 6 vías [34, 35, 36, sesenta y cinco]. Como se muestra en la Figura 2, la tasa de detección aumentó rápidamente con la fuerza de la interacción (el nivel de interacción t en las combinaciones de vías t a menudo se denomina fuerza). Con la aplicación de la NASA, por ejemplo, el 67 % de las fallas fueron provocadas por un solo valor de parámetro, el 93 % por combinaciones de 2 vías y el 98 % por combinaciones de 3 vías. Las curvas de tasa de detección para las otras aplicaciones estudiadas son similares, alcanzando el 100% de detección con interacciones de 4 a 6 vías. Los estudios realizados por otros investigadores [6, 7, 26] han sido consistentes con estos resultados.

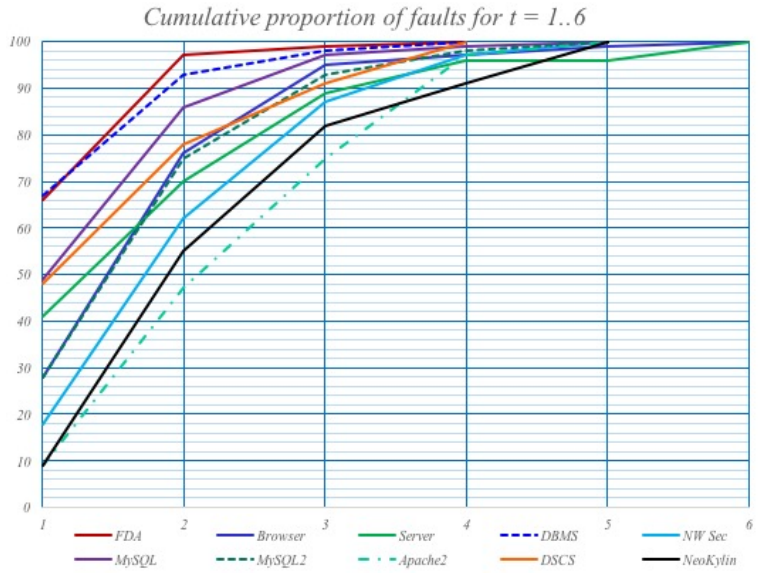


Figure 2. Tasas de detección de errores para las fuerzas de interacción 1 a 6

Si bien no son concluyentes, estos resultados son interesantes porque sugieren que, si bien las pruebas por pares no son suficientes, el grado de interacción involucrado en las fallas es relativamente bajo. Por lo tanto, probar todas las combinaciones de 4 a 6 vías puede proporcionar una seguridad razonablemente alta. Sin embargo, como ocurre con la mayoría de los problemas de software, la situación no es tan sencilla. La generación eficiente de conjuntos de pruebas para cubrir todas las combinaciones t-way es un problema matemático difícil que se ha estudiado durante casi un siglo. Además, la mayoría de los parámetros son variables continuas que tienen valores posibles en un rango muy amplio (+/- 232 o más). Estos valores deben ser discretizados a unos pocos valores distintos. Lo más evidente de todo es el problema de determinar el resultado correcto que debe esperarse del

sistema bajo prueba para cada conjunto de entradas de prueba. Generar 1000 entradas de datos de prueba es de poca ayuda si no podemos determinar qué debe producir el sistema bajo prueba (SUT) como salida para cada una de las 1000 pruebas.

Con la excepción de la combinación que cubre la generación de pruebas, estos desafíos son comunes a todos los tipos de pruebas de software y se han desarrollado una variedad de buenas técnicas para enfrentarlos. Lo que ha hecho que las pruebas combinatorias sean prácticas hoy en día es el desarrollo de algoritmos eficientes para generar pruebas que cubran combinaciones de *t*-way y métodos efectivos para integrar las pruebas producidas en el proceso de prueba. Se puede utilizar una variedad de enfoques presentados en esta publicación para hacer que las pruebas combinatorias sean una adición práctica y efectiva a la caja de herramientas del probador de software.

Los avances en los algoritmos han hecho que las pruebas combinatorias más allá de los pares finalmente sean prácticas.

Una nota sobre la terminología: usamos las definiciones a continuación, siguiendo el Instituto de Ingenieros Eléctricos y Electrónicos [30]. El término "error" también se puede utilizar cuando su significado es claro.

- *error: un error cometido por un desarrollador. Esto podría ser un error de codificación o un malentendido de los requisitos o especificaciones.*
- *defecto: una diferencia entre un programa incorrecto y uno que implementa correctamente una especificación. Un error puede resultar en uno o más defectos.*
- *falla: un resultado que difiere del resultado correcto como se especifica. Una falla en el código puede resultar en cero o más fallas, según las entradas y la ruta de ejecución.*

2.1 Dos Formas de Pruebas Combinatorias

Básicamente, existen dos enfoques para las pruebas combinatorias: usar combinaciones de valores de parámetros de *configuración* o combinaciones de valores de parámetros de *entrada*. En el primer caso, seleccionamos combinaciones de valores de parámetros configurables. Por ejemplo, se puede probar un servidor configurando todas las combinaciones de 4 vías de parámetros de configuración, como la cantidad de conexiones simultáneas permitidas, la memoria, el sistema operativo, el tamaño de la base de datos, etc., con el mismo conjunto de pruebas ejecutándose en cada configuración. Las pruebas pueden haber sido construidas utilizando cualquier metodología, no necesariamente de cobertura combinatoria. El aspecto combinatorio de este enfoque consiste en lograr una cobertura combinatoria de los valores de los parámetros de configuración. (Tenga en cuenta que el término *variable* a menudo se usa indistintamente con *parámetro* para referirse a las entradas de una función).

Las pruebas combinatorias se pueden aplicar a configuraciones, datos de entrada o ambos.

En el segundo enfoque, seleccionamos combinaciones de valores de datos de entrada, que luego pasan a formar parte de casos de prueba completos, creando un conjunto de pruebas para la aplicación. En este caso, se requiere una cobertura combinatoria de los valores de los datos de entrada para las pruebas construidas. Un enfoque ad hoc típico para las pruebas implica que los expertos en la materia configuren escenarios de uso y luego seleccionen valores de entrada para ejercitar la aplicación en cada escenario, posiblemente complementando estas pruebas con casos de problemas inusuales o sospechosos. En el enfoque combinatorio para la selección de datos de entrada, se utiliza una herramienta de generación de datos de prueba para cubrir todas las combinaciones de valores de

entrada hasta un límite específico. Una de esas herramientas es ACTS (descrita en el Apéndice C), que está disponible gratuitamente en NIST.

2.1.1 Pruebas de Configuración

Muchos sistemas de software, si no la mayoría, tienen una gran cantidad de parámetros de configuración. Muchas de las primeras aplicaciones de las pruebas combinatorias fueron para probar todos los pares de configuraciones del sistema. Por ejemplo, el software de telecomunicaciones puede configurarse para trabajar con diferentes tipos de llamadas (locales, de larga distancia, internacionales), facturación (persona que llama, tarjeta telefónica, 800), acceso (ISDN, VOIP, PBX) y servidor para facturación (Windows Server), Linux/MySQL, Oracle). El software debe funcionar correctamente con todas las combinaciones de estos, por lo que se podría aplicar un solo conjunto de pruebas a todas las combinaciones por pares de estos cuatro elementos principales de configuración. Cualquier sistema con una variedad de opciones de configuración es un candidato adecuado para este tipo de prueba.

La cobertura de configuración es quizás la forma más desarrollada de pruebas combinatorias. Se ha utilizado durante años con cobertura por pares, particularmente para aplicaciones que deben demostrar que funcionan en una variedad de combinaciones de sistemas operativos, bases de datos y características de red.

Por ejemplo, supongamos que tenemos una aplicación diseñada para ejecutarse en una variedad de plataformas compuesta por cinco componentes: un sistema operativo (Windows XP, Apple OS X, Red Hat Enterprise Linux), un navegador (Internet Explorer, Firefox), protocolo pila (IPv4, IPv6), un procesador (Intel, AMD) y una base de datos (MySQL, Sybase, Oracle), un total de $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ plataformas posibles. Con solo 10 pruebas, que se muestran en la Tabla 1, es posible probar cada componente interactuando con todos los demás componentes al menos una vez, es decir, se cubren todos los pares posibles de componentes de la plataforma.

Prueba	OS	Browser	Protocolo	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

Table 1. Pruebas de Configuraciones

2.1.2 Pruebas de Parámetros de Entrada

Incluso si una aplicación no tiene opciones de configuración, se procesará alguna forma de entrada. Por ejemplo, una aplicación de procesamiento de texto puede permitir al usuario seleccionar

10 formas de modificar algún texto resaltado: *subíndice, superíndice, subrayado, negrita, cursiva, tachado, relieve, sombreado, versalitas* o *todo en mayúsculas*. La función de procesamiento de fuentes dentro de la aplicación que recibe esta configuración como entrada debe procesar la entrada y modificar el texto en la pantalla correctamente. La mayoría de las opciones se pueden combinar, como negrita y versalitas, pero algunas son incompatibles, como subíndice y superíndice.

Las pruebas exhaustivas requieren que la función de procesamiento de fuentes funcione correctamente para todas las combinaciones válidas de estas configuraciones de entrada. Pero con 10 entradas binarias, hay 2^{10} = 1,024 combinaciones posibles. Pero el análisis empírico informado anteriormente muestra que las fallas parecen involucrar una pequeña cantidad de parámetros y que probar todas las combinaciones de 3 vías puede detectar el 90% o más de los errores. Para una aplicación de procesamiento de textos, las pruebas que detectan más del 90 % de los errores pueden ser una opción rentable, pero debemos asegurarnos de que se prueben todas las combinaciones de valores de 3 vías. Para hacer esto, creamos un conjunto de pruebas para cubrir todas las combinaciones de 3 vías (conocidas como *matriz de cobertura*) [12, 14, 23, 26, 30, 43, 63].

El componente clave es una matriz de cobertura, que incluye todas las combinaciones de t-way. Cada columna es un parámetro. Cada fila es una prueba.

En la Figura 3 se da un ejemplo, que muestra una matriz de cobertura de 3 vías para 10 variables con dos valores cada una. La propiedad interesante de esta matriz es que tres columnas cualesquiera contienen los ocho valores posibles para tres variables binarias. Por ejemplo, tomando las columnas F, G y H, podemos ver que las ocho posibles combinaciones de 3 vías (000, 001, 010, 011, 100, 101, 110, 111) ocurren en algún lugar de las tres columnas juntas. De hecho, cualquier combinación de tres columnas elegidas en cualquier orden también contendrá los ocho valores posibles. Colectivamente, por lo tanto, este conjunto de pruebas ejercitará todas las combinaciones de 3 vías de valores de entrada en solo 13 pruebas, en comparación con 1024 para una cobertura exhaustiva.

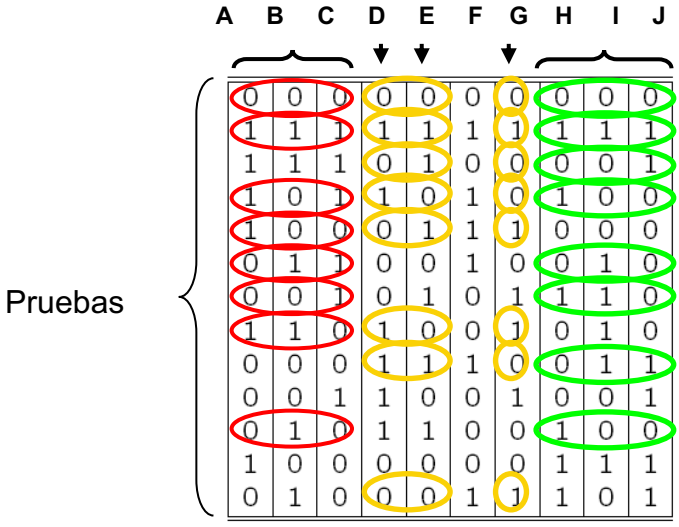


Figure 3. Matriz de cobertura de 3 vías

Se pueden generar matrices similares para cubrir hasta todas las combinaciones de 6 vías. En general, el número de pruebas combinatorias de t vías que se requerirán es proporcional a $v^t \log n$, para n parámetros con v valores posibles cada uno.

La figura 4 contrasta estos dos enfoques. Con el primer enfoque, podemos ejecutar el mismo conjunto de pruebas contra todas las combinaciones de 3 vías de opciones de configuración, mientras que, para el segundo enfoque, construiríamos un conjunto de pruebas que cubra todas las combinaciones de 3 vías de campos de transacciones de entrada. Por supuesto, estos enfoques podrían combinarse, con las pruebas combinatorias (enfoque 2) ejecutadas contra todas las combinaciones de configuración (enfoque 1).

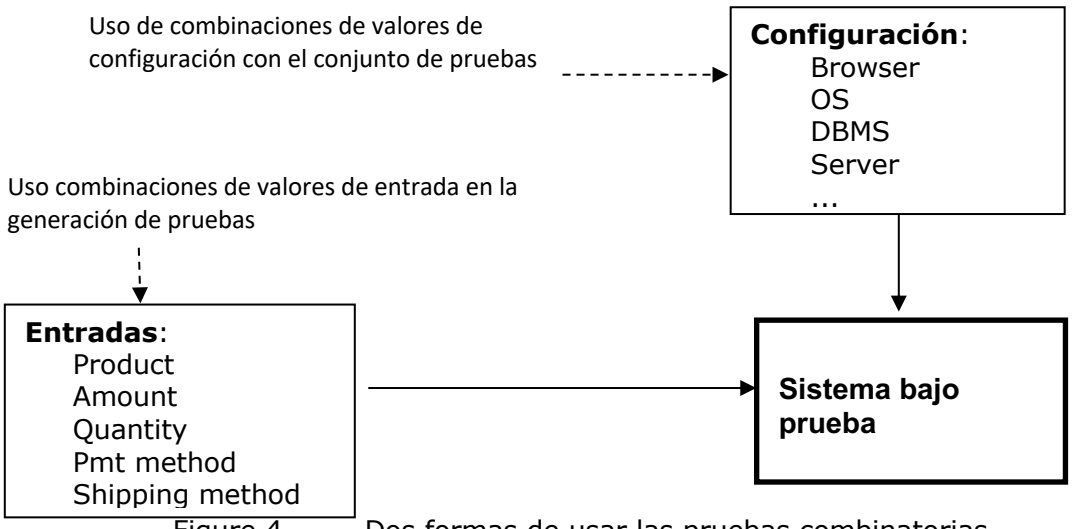


Figure 4. Dos formas de usar las pruebas combinatorias

2.2 El Principal Problema en las Pruebas

Incluso con algoritmos eficientes para producir matrices de cobertura, el problema del principal persiste: las pruebas requieren tanto datos de prueba como resultados que se deben esperar para cada entrada de datos. Las pruebas combinatorias de alta fuerza de interacción pueden requerir una gran cantidad de pruebas en algunos casos, aunque no siempre. Los enfoques para resolver el problema principal para las pruebas combinatorias incluyen:

Prueba de bloqueo: el enfoque más fácil y menos costoso es simplemente ejecutar pruebas contra el sistema bajo prueba (SUT) para verificar si alguna combinación inusual de valores de entrada provoca un bloqueo u otra falla fácilmente detectable. Este es esencialmente el mismo procedimiento utilizado en la "prueba de fuzz", que envía valores aleatorios contra el SUT. Esta forma de prueba combinatoria podría considerarse como una forma disciplinada de prueba fuzz [59]. Debe tenerse en cuenta que, aunque las pruebas aleatorias puras generalmente cubrirán un alto porcentaje de combinaciones de t vías, la cobertura del 100% de las combinaciones requiere un conjunto de pruebas aleatorias mucho más grande que una matriz de cobertura. Por ejemplo, todas las combinaciones de 3 vías de 10 parámetros con 4 valores cada una pueden cubrirse con 151 pruebas. La generación puramente aleatoria requiere más de 900 pruebas para proporcionar una cobertura total de 3 vías.

Aserciones incrustadas: una técnica cada vez más popular de "métodos formales ligeros" consiste en incrustar aserciones en el código para garantizar las relaciones adecuadas entre los datos, por ejemplo, como condiciones previas, condiciones posteriores o comprobaciones de valores de entrada. Se pueden usar herramientas como el lenguaje de modelado Java (JML) para introducir aserciones muy complejas, incorporando efectivamente una especificación formal dentro del código. Las aserciones

incrustadas sirven como una forma ejecutable de la especificación, proporcionando así un oráculo para la fase de prueba. Con aserciones incrustadas, ejercitar la aplicación con todas las combinaciones de t-way puede proporcionar una seguridad razonable de que el código funciona correctamente en una amplia gama de entradas. Este enfoque se ha utilizado con éxito para probar tarjetas inteligentes, con aserciones JML integradas que actúan como un oráculo para pruebas combinatorias [25]. Los resultados mostraron que el 80% - 90% de los errores se podían encontrar de esta manera.

Se pueden utilizar varios tipos de oráculo de prueba, según los recursos y el sistema que se esté probando.

La generación de *pruebas basada en modelos* utiliza un modelo matemático del SUT y un simulador o verificador de modelos para generar los resultados esperados para cada entrada [1,8,9,52,55]. Si se puede usar un simulador, los resultados esperados se pueden generar directamente a partir de la simulación, pero los verificadores de modelos están ampliamente disponibles y también se pueden usar para probar propiedades como la vivacidad en procesos paralelos, además de generar pruebas. Conceptualmente, se puede considerar que un verificador de modelos explora todos los estados de un modelo de sistema para determinar si una propiedad reivindicada en una declaración de especificación es verdadera. Lo que hace que un verificador de modelos sea particularmente valioso es que si la afirmación es falsa, el verificador de modelos no solo lo informa, sino que también proporciona un "contraejemplo" que muestra cómo se puede demostrar que la afirmación es falsa. Si la afirmación es falsa, el verificador de modelos lo indica y proporciona un seguimiento de los valores de entrada de los parámetros y los estados que probarán que es falsa. En efecto, este es un caso de prueba completo, es decir, un conjunto de valores de parámetros y resultados esperados. Entonces es simple mapear estos valores en casos de prueba completos en la sintaxis necesaria para el sistema bajo prueba. Los capítulos posteriores desarrollan procedimientos detallados para aplicar cada uno de estos enfoques de prueba.

2.3 Resumen del Capítulo

1. Los datos empíricos sugieren que las fallas del software son causadas por la interacción de relativamente pocos valores de parámetros, y que la proporción de fallas atribuibles a las interacciones *t*-way disminuye muy rápidamente con el aumento de *t*. Es decir, normalmente los valores de un solo parámetro o un par de valores son la causa de una falla, pero las proporciones cada vez más pequeñas son causadas por interacciones de 3 vías, 4 vías y de orden superior.

2. Debido a que una pequeña cantidad de parámetros están involucrados en las fallas, podemos lograr un alto grado de seguridad probando todas las interacciones de *t*-way, para una fuerza de interacción t apropiada (2 a 6 por lo general). El número de pruebas *t*-way que se requerirán es proporcional a v^t log n, para n parámetros con valores v cada uno.

3. Los métodos combinatorios se pueden aplicar a configuraciones o parámetros de entrada, o en algunos casos a ambos.

4. Al igual que con todos los demás tipos de pruebas, el problema principal debe resolverse, es decir, para cada entrada de prueba, se debe determinar la salida esperada para verificar si la aplicación produce el resultado correcto para cada conjunto de entradas. Hay una variedad de métodos disponibles para resolver el problema principal.

3 PRUEBAS DE CONFIGURACIÓN

Este capítulo presenta ejemplos resueltos que ilustran el desarrollo de configuraciones de prueba. Como se verá, las ventajas de las pruebas combinatorias aumentan con el tamaño del problema.

3.1 Ejemplo de Plataforma de Aplicación Simple

Volviendo al ejemplo simple presentado en el Capítulo 2, ilustramos el desarrollo de configuraciones de prueba y comparamos el tamaño de los conjuntos de prueba para varias fuerzas de interacción versus probar todas las configuraciones posibles. Para los cinco parámetros de configuración, tenemos $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ configuraciones. La convención para describir las variables y los valores en las pruebas combinatorias es $v_1^{n1} v_2^{n2}$ ... donde v_i es el número de valores de las variables y n_i es el número de ocurrencias. Por lo tanto, esta configuración se designa como $2^3 3^2$. Tenga en cuenta que en $t = 5$, el número de pruebas es el mismo que el de las pruebas exhaustivas para este ejemplo, porque solo hay cinco parámetros. Los ahorros como porcentaje de las pruebas exhaustivas son buenos, pero no tan impresionantes para este pequeño ejemplo. Con sistemas más grandes, los ahorros pueden ser enormes, como se verá en la siguiente sección.

Parámetro	Valores
Operating system	XP, OS X, RHL
Browser	IE, Firefox
Protocol	IPv4, IPv6
CPU	Intel, AMD
DBMS	MySQL, Sybase, Oracle

Table 2. Opciones de configuración de ejemplo simple.

Ahora podemos generar configuraciones de prueba utilizando la herramienta ACTS. Para simplificar la presentación, ilustramos el uso de la versión de línea de comandos de ACTS, pero hay disponible una versión GUI intuitiva que puede ser más conveniente. Esta herramienta se resume en el Apéndice C y se incluye un manual de usuario completo con la descarga de ACTS.

El primer paso para crear configuraciones de prueba es especificar los parámetros y valores posibles en un archivo para ingresar a ACTS, como se muestra en la Figura 5:

```
[System]

[Parameter]
OS (enum): XP,OS_X,RHL
Browser (enum): IE, Firefox
Protocol(enum): IPv4,IPv6
CPU (enum):  Intel,AMD
DBMS (enum): MySQL,Sybase,Oracle

[Relation]
[Constraint]
[Misc]
```

Figure 5. Archivo de entrada de ejemplo simple para ACTS.

Tenga en cuenta que la mayoría de las etiquetas entre corchetes en el archivo de entrada son opcionales y no se completaron para este ejemplo. La parte esencial del archivo es la especificación

[Parámetro], en el formato <nombre del parámetro> (<tipo>): <valores>, donde se enumeran uno o más valores separados por comas. La herramienta se puede ejecutar en la línea de comando:

java -Ddoi=2 –jar acts_cmd.jar ActsConsoleManager entrada.txt salida.txt

Se puede especificar una variedad de opciones, pero para este ejemplo solo usamos la opción "grado de interacción" para especificar cobertura de 2 vías, 3 vías, etc. La salida se puede crear en la forma conveniente que se muestra a continuación, o como una matriz de números, valores separados por comas o una hoja de cálculo de Excel. Si la salida será utilizada por probadores humanos en lugar de como entrada para el procesamiento posterior de la máquina, el formato de la Figura 6 es útil:

```
Degree of interaction coverage: 2
Number of parameters: 5
Maximum number of values per parameter: 3
Number of configurations: 10
------------------------------------
Configuration #1:

1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv4
4 = CPU=Intel
5 = DBMS=MySQL
------------------------------------
Configuration #2:

1 = OS=XP
2 = Browser=Firefox
3 = Protocol=IPv6
4 = CPU=AMD
5 = DBMS=Sybase
------------------------------------
Configuration #3:

1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv6
4 = CPU=Intel
5 = DBMS=Oracle
------------------------------------
Configuration #4:

1 = OS=OS_X
2 = Browser=Firefox
3 = Protocol=IPv4
4 = CPU=AMD
5 = DBMS=MySQL
. . .
```

Figure 6. Extracto de la salida de configuración de prueba que cubre todas las combinaciones de 2 vías.

El conjunto de prueba completo para combinaciones de 2 vías se muestra en la Tabla 1 en la Sección 2.1.1. Solo se necesitan 10 pruebas. Pasar a fuerzas de interacción de 3 vías o superiores requiere más pruebas, como se muestra en la Tabla 3.

t	# Tests	% of Exhaustive
2	10	14
3	18	25
4	36	50
5	72	100

Table 3. Número de pruebas combinatorias para un ejemplo simple.

En este ejemplo, se podrían lograr ahorros sustanciales probando configuraciones de t-way en lugar de todas las configuraciones posibles, aunque para algunas aplicaciones (como un módulo pequeño pero muy crítico) puede justificarse una prueba completa y exhaustiva. Como veremos en el siguiente ejemplo, en muchos casos es imposible probar todas las configuraciones, por lo que necesitamos desarrollar alternativas razonables.

3.2 Ejemplo de Aplicación de teléfono Inteligente

Los teléfonos inteligentes se han vuelto enormemente populares porque combinan la capacidad de comunicación con potentes pantallas gráficas y capacidad de procesamiento. Literalmente, decenas de miles de aplicaciones para teléfonos inteligentes, o 'apps', se desarrollan anualmente. Entre las plataformas para aplicaciones de teléfonos inteligentes se encuentra Android, que incluye un entorno de desarrollo de código abierto y un sistema operativo especializado. Las unidades Android contienen una gran cantidad de opciones de configuración que controlan el comportamiento del dispositivo. Las aplicaciones de Android deben funcionar en una variedad de plataformas de hardware y software, ya que no todos los productos admiten las mismas opciones. Por ejemplo, algunos teléfonos inteligentes pueden tener un teclado físico y otros pueden presentar un teclado en pantalla usando la pantalla sensible al tacto. Los teclados también pueden ser solo numéricos con algunas teclas especiales o un teclado de máquina de escribir completo. Según el estado de la aplicación y las elecciones del usuario, el teclado puede estar visible u oculto. Asegurarse de que una aplicación en particular funcione en la enorme cantidad de opciones es un desafío importante para los desarrolladores. El amplio conjunto de opciones hace que sea difícil probar todas las configuraciones posibles, por lo que las pruebas combinatorias son una alternativa práctica.

La figura 7 muestra un archivo de configuración de recursos para aplicaciones de Android. Se pueden configurar un total de 35 opciones. Nuestra tarea es desarrollar un conjunto de configuraciones de prueba que permitan probar todas las combinaciones de 4 vías de estas opciones. El primer paso es determinar el conjunto de parámetros y valores posibles para cada uno que se probará. Aunque las opciones se enumeran individualmente para permitir que se asocie un valor entero específico con cada una, representan claramente conjuntos de valores de opciones con elecciones mutuamente excluyentes. Por ejemplo, "Teclado oculto" puede ser "sí", "no" o "indefinido". Estos valores serán las configuraciones posibles para los nombres de los parámetros que usaremos para generar una matriz de cobertura. La Tabla 4 muestra los nombres de los parámetros y la cantidad de valores posibles que usaremos como entrada para el generador de matriz de cobertura. Para obtener una especificación completa de estos parámetros, consulte:

http://developer.android.com/reference/android/content/res/Configuration.html

```
int     HARDKEYBOARDHIDDEN_NO;
int     HARDKEYBOARDHIDDEN_UNDEFINED;
int     HARDKEYBOARDHIDDEN_YES;
int     KEYBOARDHIDDEN_NO;
int     KEYBOARDHIDDEN_UNDEFINED;
int     KEYBOARDHIDDEN_YES;
int     KEYBOARD_12KEY;
int     KEYBOARD_NOKEYS;
int     KEYBOARD_QWERTY;
int     KEYBOARD_UNDEFINED;
int     NAVIGATIONHIDDEN_NO;
int     NAVIGATIONHIDDEN_UNDEFINED;
int     NAVIGATIONHIDDEN_YES;
int     NAVIGATION_DPAD;
int     NAVIGATION_NONAV;
int     NAVIGATION_TRACKBALL;
int     NAVIGATION_UNDEFINED;
int     NAVIGATION_WHEEL;
int     ORIENTATION_LANDSCAPE;
int     ORIENTATION_PORTRAIT;
int     ORIENTATION_SQUARE;
int     ORIENTATION_UNDEFINED;
int     SCREENLAYOUT_LONG_MASK;
int     SCREENLAYOUT_LONG_NO;
int     SCREENLAYOUT_LONG_UNDEFINED;
int     SCREENLAYOUT_LONG_YES;
int     SCREENLAYOUT_SIZE_LARGE;
int     SCREENLAYOUT_SIZE_MASK;
int     SCREENLAYOUT_SIZE_NORMAL;
int     SCREENLAYOUT_SIZE_SMALL;
int     SCREENLAYOUT_SIZE_UNDEFINED;
int     TOUCHSCREEN_FINGER;
int     TOUCHSCREEN_NOTOUCH;
int     TOUCHSCREEN_STYLUS;
int     TOUCHSCREEN_UNDEFINED;
```

Figure 7. Archivo de configuración de recursos de Android.

Nombre del parámetro	Valores	# Valores
HARDKEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARD	12KEY, NOKEYS, QWERTY, UNDEFINED	4
NAVIGATIONHIDDEN	NO, UNDEFINED, YES	3
NAVIGATION	DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL	5
ORIENTATION	LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED	4
SCREENLAYOUT_LONG	MASK, NO, UNDEFINED, YES	4
SCREENLAYOUT_SIZE	LARGE, MASK, NORMAL, SMALL, UNDEFINED	5
TOUCHSCREEN	FINGER, NOTOUCH, STYLUS, UNDEFINED	4

Table 4. Opciones de configuración de Android.

Usando la Tabla 4, ahora podemos calcular el número total de configuraciones: $3 \cdot 3 \cdot 4 \cdot 3 \cdot 5 \cdot 4 \cdot 4 \cdot 5 \cdot 4 = 172{,}800$ configuraciones (es decir, un $3^3 4^4 5^2$ sistema). Al igual que muchas aplicaciones, las pruebas exhaustivas requerirán alguna intervención humana para ejecutar las pruebas y verificar los resultados, y un conjunto de pruebas generalmente incluirá muchas

pruebas. Si cada conjunto de pruebas se puede ejecutar en 15 minutos, se necesitarán aproximadamente 24 años de personal para completar las pruebas de una aplicación. Con costos de salario y beneficios para cada probador de $150,000, el costo de probar una aplicación será de más de $3 millones, lo que hace que sea prácticamente imposible obtener ganancias para la mayoría de las aplicaciones. ¿Cómo podemos proporcionar pruebas efectivas para aplicaciones a un costo razonable?

Usando el generador de matriz de cobertura, podemos producir pruebas que cubren combinaciones de valores de t-way. La Tabla 5 muestra el número de pruebas requeridas en varios niveles de t. Para muchas aplicaciones, las pruebas de 2 ó 3 vías pueden ser apropiadas y cualquiera de estas requerirá menos del 1% del tiempo requerido para cubrir todas las configuraciones de prueba posibles.

t	# Tests	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3

Table 5. Ejemplo de número de pruebas combinatorias para Android.

3.3 Costo y Consideraciones Prácticas

3.3.1 Combinaciones y restricciones no válidas

El sistema descrito en la Sección 3.1 ilustra una situación común en todos los tipos de pruebas: algunas combinaciones no se pueden probar porque no existen para los sistemas bajo prueba. En este caso, si el sistema operativo es OS X o Linux, Internet Explorer no está disponible como navegador. Tenga en cuenta que no podemos simplemente eliminar pruebas con estas combinaciones no comprobables, porque eso daría como resultado la pérdida de otras combinaciones que son esenciales para probar pero que no están cubiertas por otras pruebas. Por ejemplo, eliminar las pruebas 5 y 7 en la Sección 2.1.1 significaría que también perderíamos la prueba para Linux con el protocolo IPv6.

Algunas combinaciones nunca ocurren en la práctica.

Una forma de evitar este problema es eliminar las pruebas y complementar el conjunto de pruebas con configuraciones de prueba construidas manualmente para cubrir las combinaciones eliminadas, pero las herramientas de matriz de cobertura ofrecen una mejor solución. Con ACTS podemos especificar restricciones, que le indican a la herramienta que no incluya combinaciones específicas en las configuraciones de prueba generadas. ACTS admite un conjunto de operadores lógicos y aritméticos de uso común para especificar restricciones. En este caso, se puede utilizar la siguiente restricción para garantizar que no se generen combinaciones no válidas:

(SO != “XP”) => (Navegador = “Firefox”)

La herramienta de matriz de cobertura generará un conjunto de configuraciones de prueba que no incluye las combinaciones no válidas, pero cubre todas las que son esenciales. La matriz de configuración de prueba revisada se muestra en la Figura 8 a continuación. Los valores de los parámetros que han cambiado con respecto a las configuraciones originales están subrayados. Tenga en cuenta que agregar la restricción también resultó en la reducción de la cantidad de configuraciones de prueba en una. Este no siempre será el caso, dependiendo de las restricciones utilizadas, pero ilustra cómo las restricciones pueden ayudar a reducir el problema. Incluso si se pueden probar

combinaciones particulares, el equipo de prueba puede considerar que algunas combinaciones son innecesarias y se pueden usar restricciones para evitar estas combinaciones, lo que posiblemente reduzca el número de configuraciones de prueba.

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	Firefox	IPv4	Intel	Sybase
6	OS X	Firefox	IPv6	AMD	Oracle
7	RHL	Firefox	IPv6	Intel	MySQL
8	RHL	Firefox	IPv4	Intel	Oracle
9	XP	IE	IPv4	AMD	Sybase

Figure 8. Configuraciones de prueba para un ejemplo simple con restricción.

3.3.2 Factores de Costo

El uso de métodos combinatorios para diseñar configuraciones de prueba es probablemente el enfoque combinatorio más utilizado porque es rápido y fácil de hacer y, por lo general, ofrece mejoras significativas en las pruebas. Las pruebas combinatorias para parámetros de entrada pueden proporcionar una mejor cobertura de prueba a un costo menor que las pruebas convencionales, y pueden extenderse a una cobertura de alta resistencia para brindar una garantía mucho mayor.

3.4 Resumen del Capítulo

1. La prueba de configuración es probablemente la aplicación más utilizada de los métodos combinatorios en las pruebas de software. Siempre que una aplicación tenga aproximadamente cinco o más atributos configurables, es probable que una matriz de cobertura haga que las pruebas sean más eficientes. Los atributos configurables suelen tener un pequeño número de valores posibles cada uno, lo que es una situación ideal para los métodos combinatorios. Debido a que el número de pruebas t-way es proporcional a $v^t \log n$, para n parámetros con valores v cada uno, a menos que los atributos configurables tengan más de 8 ó
10 valores posibles cada uno, el número de pruebas generadas probablemente sea razonable. El problema de prueba del mundo real presentado en la Sección 3.2 tiene un tamaño bastante típico, donde las interacciones de 4 vías se pueden probar con unos pocos cientos de pruebas.

2. Debido a que muchos sistemas tienen ciertas configuraciones que pueden no ser de interés (como el navegador Internet Explorer en un sistema Linux), las restricciones son una consideración importante en cualquier tipo de prueba. Con los métodos combinatorios, es importante que el generador de matriz de cobertura permita la inclusión de restricciones para que se prueben todas las interacciones relevantes y no se pierda información importante porque una prueba contiene una combinación imposible.

4 PRUEBA DE PARÁMETROS DE ENTRADA

Como se señaló en la introducción, la ventaja clave de las pruebas combinatorias se deriva del hecho de que todas, o casi todas, las fallas de software parecen involucrar interacciones de solo unos pocos parámetros. El uso de pruebas combinatorias para seleccionar configuraciones puede hacer que las pruebas sean más eficientes, pero puede ser aún más efectivo cuando se usa para seleccionar valores de parámetros de entrada. Los probadores tradicionalmente desarrollan escenarios de cómo se usará una aplicación, luego seleccionan entradas que ejercitarán cada una de las características de la aplicación utilizando valores representativos, normalmente complementados con valores extremos para probar el rendimiento y la confiabilidad. El problema con este enfoque, a menudo ad hoc, es que generalmente se perderán las combinaciones inusuales, por lo que un sistema puede pasar todas las pruebas y funcionar bien en circunstancias normales, pero finalmente encuentra una combinación de entradas que no puede procesar correctamente.

Al probar todas las combinaciones t-way, para algún nivel específico de t, las pruebas combinatorias pueden ayudar a evitar este tipo de situación. En este capítulo trabajamos a través de un pequeño ejemplo para ilustrar el uso de estos métodos.

4.1 Ejemplo de Módulo de Control de Acceso

El sistema bajo prueba es un módulo de control de acceso que implementa la siguiente política:

Acceso permitido si y solo si:

- El sujeto es un empleado
  Y (AND) la hora actual está entre las 9 am y las 5 pm
  Y (AND) no es fin de semana
- O (OR) el sujeto es un empleado con un código de autorización especial
- O (OR) el sujeto es un auditor
  Y (AND) la hora es entre las 9 am y las 6 pm
  (los fines de semana no están limitados).

Los parámetros de entrada para este módulo se muestran en la Figura 9:

```
emp:  boolean;
time:  0..1440;  // time in minutes
day:   {m,tu,w,th,f,sa,su};
auth: boolean;
aud:  boolean;
```

Figure 9. Parámetros de entrada del módulo de control de acceso.

Nuestra tarea es desarrollar una serie de pruebas que cubran estas entradas. El primer paso será desarrollar una tabla de parámetros y valores posibles, similar a la de la Sección 3.1 del capítulo anterior. La única diferencia es que en este caso estamos tratando con parámetros de entrada en lugar de opciones de configuración. En su mayor parte, la tarea es simple: simplemente tomamos los valores directamente de las especificaciones o el código, como se muestra en la Figura 10. Varios parámetros son booleanos, y usaremos 0 y 1 para valores falsos y verdaderos respectivamente. Para el día de la semana, solo hay siete valores, por lo que se pueden usar todos. Sin embargo, la hora del día presenta un problema. Recuerde que el número de pruebas generadas para n parámetros es

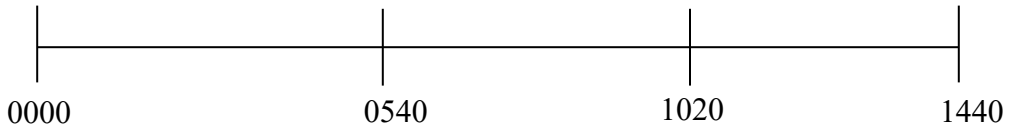
proporcional a v^t, donde v es el número de valores y t es el nivel de interacción (de 2 a 6 vías). Para todos los valores booleanos y pruebas de 4 vías, por lo tanto, el número de pruebas será un múltiplo de 24. Pero considere lo que sucede con una gran cantidad de valores posibles, como 24 horas. El número de pruebas será proporcional a $24^4 = 331.736$. Para este ejemplo, el tiempo se da en minutos, lo que obviamente sería completamente intratable. Por lo tanto, debemos seleccionar valores representativos para el parámetro de la hora. Este problema ocurre en todos los tipos de pruebas, no solo con métodos combinatorios, y se han desarrollado buenos métodos para solucionarlo. La mayoría de los probadores ya están familiarizados con dos de estos: *partición de equivalencia* y *análisis de valor límite*. Se pueden encontrar antecedentes adicionales sobre estos métodos en textos de prueba de software como Ammann y Offutt [2], Beizer [4], Copeland [21], Mathur [45] y Myers [52].

Parámetro	Valores
emp	0,1
time	??
day	`m,tu,w,th,f,sa,su`
auth	0, 1
aud	0, 1

Figure 10. Parámetros y valores para el control de acceso.

Ambos métodos intuitivamente obvios producirán un conjunto más pequeño de valores que deberían ser adecuados para fines de prueba, al dividir los valores posibles en particiones que sean significativas para el programa que se está probando. Se selecciona un valor para cada partición. El objetivo es particionar el espacio de entrada de modo que cualquier valor seleccionado de la partición afecte al programa bajo prueba de la misma manera que cualquier otro valor en la partición. Por lo tanto, idealmente, si un caso de prueba contiene un parámetro x que tiene un valor y, reemplazar y con cualquier otro valor de la partición no afectará el resultado del caso de prueba. Es posible que este ideal no siempre se logre en la práctica.

¿Cómo deben determinarse las particiones? Un enfoque obvio, pero no necesariamente bueno, es simplemente seleccionar valores de varios puntos en el rango de una variable. Por ejemplo, si la capacidad puede oscilar entre 0 y 20 000, puede parecer sensato seleccionar 0, 10 000 y 20 000 como valores posibles. Pero es probable que este enfoque pase por alto casos importantes que dependen de los requisitos específicos del sistema bajo prueba. Se requiere algo de juicio, pero las particiones generalmente se determinan mejor a partir de la especificación. En este ejemplo, las 9 am y las 5 pm son significativas, por lo que 0540 (9 horas después de la medianoche) y 1020 (17 horas después de la medianoche) determinan las particiones apropiadas:



Idealmente, el programa debería comportarse igual para cualquiera de los tiempos dentro de las particiones; no debería importar si la hora es las 4:00 am o las 7:03 am, por ejemplo, porque la especificación trata ambas horas de la misma manera. Del mismo modo, no debe importar qué horario se elija entre las 9 am y las 5 pm; el programa debe comportarse igual para las 10:20 am y las

Utilice un máximo de 8 a 10 valores por parámetro para mantener las pruebas manejables.

2:33 pm. Una estrategia común, el análisis de valores límite, es seleccionar valores de prueba en cada límite y en la unidad más pequeña posible a cada lado del límite, para tres valores por límite. La intuición, respaldada por la investigación empírica, es que los errores son más probables en las condiciones de contorno porque en estos puntos se pueden cometer errores de programación. Por ejemplo, si los requisitos para el software del cajero automático indican que no se debe permitir un retiro que exceda los $300, podría ocurrir un error de programación como el siguiente:

```
if (amount > 0 && amount < 300) {
      //process withdrawal
} else {
      // error message
}
```

Aquí, la segunda condición debería haber sido "cantidad <= 300", por lo que un caso de prueba que incluye el valor cantidad = 300 puede detectar el error, pero una prueba con cantidad = 305 no lo haría.

En general, también es deseable probar los extremos de los rangos. Una posible selección de valores para el parámetro de tiempo sería: 0000, 0539, 0540, 0541, 1019, 1020, 1021 y 1440. Más valores serían mejores, pero el probador puede creer que este es el conjunto más efectivo para el presupuesto de tiempo disponible. Con esta selección, el número total de combinaciones es = 448.

La generación de arreglos de cobertura para t = 2 a 6, como se detalla en la Sección 3.1, da como resultado el siguiente número de pruebas:

t	# Tests
2	56
3	112
4	224

Figure 11. Número de pruebas para control de acceso.

4.2 Sistemas del Mundo Real

Cuanto más grande sea el sistema, mayor será el beneficio de las pruebas combinatorias.

Como en el ejemplo anterior, la ventaja sobre las pruebas exhaustivas no es grande, debido al pequeño número de parámetros. Con problemas más grandes, las ventajas de las pruebas combinatorias pueden ser espectaculares. Por ejemplo, considere el problema de probar el software que procesa la configuración de los interruptores para el panel que se muestra en la Figura 12. Hay 34 interruptores, cada uno de los cuales puede estar activado o desactivado, para un total de $2^{34} = 1{,}7 \times 10^{10}$ configuraciones posibles. Claramente, no podemos probar 17 mil millones de configuraciones posibles, pero todas las interacciones de 3 vías pueden probarse con solo 33 pruebas, y todas las interacciones de 4 vías con solo 85. Esto puede parecer sorprendente al principio, pero resulta del hecho de que cada prueba de 34 parámetros contiene $\binom{34}{3} = 5984$ combinaciones de 3 vías y $= \binom{34}{4} 46\,376$ combinaciones de 4 vías.

Figure 12. Panel con 34 interruptores.

4.3 Costo y Consideraciones Prácticas

Los métodos combinatorios pueden ser altamente efectivos y reducir sustancialmente el costo de las pruebas. Por ejemplo, Justin Hunter aplicó estos métodos a una amplia variedad de problemas de prueba y encontró consistentemente tanto un menor costo como una detección de errores más rápida [30]. Pero como con la mayoría de los aspectos de la ingeniería, se deben considerar las compensaciones. Entre los más importantes está la cuestión de cuándo dejar de realizar pruebas, equilibrando el costo de las pruebas con el riesgo de no poder descubrir fallas adicionales. Se ha dedicado una gran cantidad de investigación a este tema, y existen modelos sofisticados disponibles para determinar cuándo el costo de las pruebas adicionales excederá los beneficios esperados [10, 45]. Los modelos existentes sobre cuándo detener la prueba también se pueden aplicar al enfoque de prueba combinatoria, pero hay una consideración adicional: ¿Cuál es la fuerza de interacción adecuada para usar en este tipo de prueba?

Para abordar estas preguntas, considere la cantidad de pruebas con diferentes intensidades de interacción para un ejemplo de software de aviónica [34] que se muestra en la Figura 13. Si bien la cantidad de pruebas será diferente (probablemente mucho menor que en la Figura 13) según el sistema bajo prueba, la magnitud de la diferencia entre los niveles de t será similar a la Figura 13, porque el número de pruebas crece con v^t, para parámetros con valores de v. Es decir, el número de pruebas crece con el exponente t, por lo que queremos usar la fuerza de interacción más pequeña que sea apropiada para el problema. Intuitivamente, parece que si no se detectan fallas mediante las pruebas t-way, entonces puede ser razonable realizar pruebas adicionales solo para las interacciones t+1, pero no mayores si no se encuentran fallas adicionales en t+1. En los estudios empíricos de fallas de software, el número de fallas detectadas en $t > 2$ disminuyó monótonamente con t, por lo que esta heurística parece tener sentido: comience a probar usando combinaciones de 2 vías (por pares), continúe aumentando la fuerza de interacción t hasta que no haya errores detectados por las pruebas t-way, luego (opcionalmente) intente t+1 y asegúrese de que no se detecten errores adicionales. Al igual que con otros aspectos del desarrollo de software, esta guía también depende de los recursos, las limitaciones de tiempo y las consideraciones de costo-beneficio.

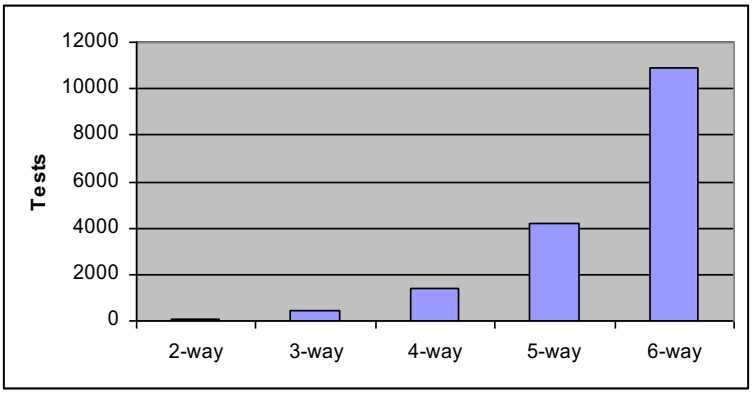


Figure 13. Número de pruebas para aviónica.

Al aplicar métodos combinatorios a los parámetros de entrada, los factores clave de costo son el número de valores por parámetro, la fuerza de interacción y el número de parámetros.

Como se muestra arriba, la cantidad de pruebas aumenta rápidamente a medida que aumenta el valor de t, pero la tasa de aumento depende de la cantidad de valores por parámetro. Las variables binarias, con solo dos valores cada una, dan como resultado muchas menos pruebas que los parámetros con muchos valores cada uno. Como cuestión práctica, al dividir el espacio de entrada (sección 4.1), es mejor mantener el número de valores por parámetro por debajo de 8 ó 10 si es posible.

Debido a que el número de pruebas aumenta solo logarítmicamente con el número de parámetros, el tamaño del conjunto de pruebas para un problema grande puede ser solo algo mayor que para un problema mucho más pequeño. Por ejemplo, si un proyecto usa pruebas combinatorias para un sistema que tiene 20 parámetros y genera varios cientos de pruebas, un sistema mucho más grande con 40 a 50 parámetros puede requerir solo unas cuantas docenas de pruebas más. Los métodos combinatorios pueden generar la mejor relación costo-beneficio para sistemas grandes

4.4 Resumen del Capítulo

1. La ventaja clave de las pruebas combinatorias se deriva del hecho de que todas, o casi todas, las fallas de software parecen involucrar interacciones de solo unos pocos parámetros. La generación de una matriz de cobertura de valores de parámetros de entrada nos permite probar todas estas interacciones, hasta un nivel de combinaciones de 5 ó 6 vías, según los recursos.

2. Las pruebas prácticas a menudo requieren abstraer los valores posibles de una variable en un pequeño conjunto de clases de equivalencia. Por ejemplo, si una variable es un número entero de 32 bits, claramente no es posible probar el rango completo de valores en +/- 2^{31}. Este problema no es exclusivo de las pruebas combinatorias, pero ocurre en la mayoría de las metodologías de prueba. Se requieren heurísticas simples y juicios de ingeniería para determinar la porción apropiada de valores en clases de equivalencia, pero una vez que esto se logra, es posible generar matrices de cobertura de unos pocos cientos a unos pocos miles de pruebas para muchas aplicaciones. La exhaustividad de la cobertura dependerá de los recursos y la criticidad de la aplicación.

5 MATRICES DE COBERTURA DE SECUENCIAS

Al probar el software basado en eventos, la condición crítica para desencadenar fallas a menudo es si un evento en particular ha ocurrido o no antes de otro, no necesariamente si son consecutivos. Esta situación refleja el hecho de que, en muchos casos, se debe alcanzar un estado particular antes de que se pueda desencadenar una falla en particular. Por ejemplo, puede ocurrir una falla al conectar el dispositivo A solo si el dispositivo B ya está conectado. Los métodos descritos en este capítulo se desarrollaron para resolver un problema real en la prueba y evaluación de interoperabilidad, utilizando métodos combinatorios para proporcionar pruebas eficientes. Las matrices de cobertura de secuencias, tal como se definen aquí, garantizan que cualquier evento t se probará en cada orden posible de t-way.

En muchos sistemas, el orden de las entradas es importante.

Para este problema podemos definir una matriz que cubra secuencias [39, 40], que es un conjunto de pruebas que aseguran que todas las secuencias de eventos de t-way han sido probadas. Los eventos t en la secuencia se pueden intercalar con otros, pero se probarán todas las permutaciones. Por ejemplo, podemos tener un componente de un sistema de automatización de fábrica que utiliza ciertos dispositivos que interactúan con un programa de control. Queremos probar los eventos definidos en la Tabla 6.

¡Hay 6! = 720 secuencias posibles para estos seis eventos, y el sistema debería responder de manera correcta y segura sin importar el orden en que ocurran. Se puede instruir a los operadores para que usen un orden en particular, pero los errores son inevitables y no deben causar lesiones a los usuarios ni comprometer a la empresa. Debido a que la configuración, las conexiones y el funcionamiento de este componente son manuales, cada prueba puede llevar una cantidad considerable de tiempo. No es raro que las pruebas a nivel del sistema como esta, tarden horas en ejecutarse, monitorearse y completarse. Queremos probar este sistema lo más a fondo posible, pero las limitaciones de tiempo y presupuesto no permiten probar todas las secuencias posibles, por lo que probaremos todas las secuencias de 3 eventos.

Con seis eventos, *a, b, c, d, e* y *f*, un subconjunto de tres es {*b, d, e*}, que se puede organizar en seis permutaciones: [*b d e*], [*b e d*], [*d b e*], [*d e b*], [*e b d*], [*e b d*]. Una prueba que cubre la permutación [*d b e*] es: [*a d c f b e*]; otro es [*a d c b e f*]. Un sistema de ejemplo más grande puede tener 10 dispositivos para conectar, ¡en cuyo caso el número de permutaciones es 10!, o 3,628,800 pruebas para pruebas exhaustivas. En ese caso, una matriz de cobertura de secuencia de 3 vías con 14 pruebas que cubren todas las secuencias de 3 vías es una mejora espectacular, al igual que 72 pruebas para todas las secuencias de 4 vías (consulte la Tabla 8).

Evento	Descripción
a	connect air flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	engage drive motor
f	engage steering control

Table 6. Eventos del sistema

Definición. Definimos una matriz de cobertura de secuencias, SCA(N, S, t) como una matriz N x S donde las entradas son de un conjunto finito S de s símbolos, tal que cada permutación de símbolos de S ocurre en al menos una fila; no se requiere que los símbolos t en la permutación sean adyacentes. Es decir, para cada disposición en t de los símbolos x_1, x_2, ..., x_t, la expresión regular $.*x_1.*x_2.*x_t.*$coincide con al menos una fila de la matriz. Las matrices de cobertura de secuencias, como su nombre lo indica, son análogas a las matrices de cobertura estándar, que incluyen al menos una de cada combinación de t vías de cualquier n variable, donde $t < n$. Hay una variedad de algoritmos disponibles para construir arreglos de cobertura, pero estos no se pueden usar para generar secuencias de dos vías porque están diseñados para cubrir combinaciones en cualquier orden.

Ejemplo 1. Considere el problema de probar cuatro eventos, *a, b, c* y *d*. Por conveniencia, una permutación de símbolos de t vías se denomina secuencia de t vías. ¡Hay 4! = 24 permutaciones posibles de estos cuatro eventos, pero podemos probar todas las secuencias de 3 vías de estos eventos con solo seis pruebas (ver Tabla 7).

Test				
1	*a*	*d*	*b*	*c*
2	*b*	*a*	*c*	*d*
3	*b*	*d*	*c*	*a*
4	*c*	*a*	*b*	*d*
5	*c*	*d*	*b*	*a*
6	*d*	*a*	*c*	*b*

Table 7. Pruebas para cuatro eventos.

5.1 Construcción de Matrices de Cobertura de Secuencias

Se puede construir una matriz de cobertura de secuencia bidireccional enumerando los eventos en algún orden para una prueba y en orden inverso para la segunda prueba:

1	*a*	*b*	*c*	*d*
2	*d*	*c*	*b*	*a*

Para ver que el procedimiento del Ejemplo 2 genera pruebas que cubren todas las secuencias de 2 vías, tenga en cuenta que para la cobertura de secuencias de 2 vías, cada par de variables x e y, $x..y$ e $y..x$ deben estar ambas en alguna prueba. (donde $a..b$ significa que a es eventualmente seguido por b). Todas las variables están incluidas en cada prueba, por lo tanto, cualquier secuencia $x..y$ debe estar en la prueba 1 o en la prueba 2 y su inversa $y..x$ en la otra prueba.

Para la generación de pruebas de secuencias t-way, donde $t > 2$, usamos un algoritmo voraz que genera una gran cantidad de pruebas, califica cada una por la cantidad de secuencias previamente descubiertas que cubre y luego elige la prueba con la puntuación más alta. Este enfoque simple produce resultados sorprendentemente buenos.

5.2 Uso de Matrices de Cobertura de Secuencias

Los arreglos de cobertura de secuencia se han incorporado a las pruebas operativas para un sistema de misión crítica que utiliza múltiples dispositivos con entradas y salidas a una computadora portátil. El procedimiento de prueba tiene 8 pasos: arrancar el sistema, abrir la aplicación, ejecutar el escaneo, conectar los periféricos P-1 a P-5. Se espera que, para algunas secuencias, el sistema no funcione correctamente, por lo que el orden de conexión de los periféricos es un aspecto crítico de la prueba. Además, existen restricciones en la secuencia de eventos: no se puede escanear hasta que la aplicación

esté abierta; no se puede abrir la aplicación hasta que se inicia el sistema. Hay 40 320 permutaciones de 8 pasos, pero algunas son redundantes (p. ej., cambiar el orden de los periféricos conectados antes del arranque) y otras no son válidas (violan una restricción). Alrededor de 7000 son válidos y no redundantes, pero esto es demasiado para probar un sistema que requiere conexiones físicas manuales de dispositivos.

El sistema se probó utilizando una matriz de cobertura de secuencia de siete pasos, incorporando la suposición de que no hay necesidad de examinar secuencias de fuerza 3 que impliquen arranque. La configuración de prueba inicial (Figura 14) se extrajo de la biblioteca de pruebas de secuencia calculadas previamente. Se realizaron algunos cambios en las secuencias precalculadas en función de los requisitos únicos de la prueba del sistema. Si 6 = 'Abrir aplicación' y 5 = 'Ejecutar análisis', los casos 1, 4, 6, 8, 10 y 12 no son válidos porque el análisis no se puede ejecutar antes de iniciar la aplicación. Esto se manejó 'intercambiando 0 y 1' cuando están adyacentes (1 y 4), fuera de servicio. Para los demás casos, se generaron varios casos a partir de cada uno de los cuales eran mutaciones válidas del caso no válido. También se incorporó una prueba para ver si importaba dónde se colocaron cada una de las tres conexiones USB. El último caso de prueba asegura al menos fuerza 2 (secuencia de longitud 2) para todas las conexiones de periféricos y 'arranque', es decir, que cada conexión de periférico se produzca antes del arranque. La matriz de prueba final se muestra en la Tabla 9.

Test 1	0	1	2	3	4	5	6
Test 2	6	5	4	3	2	1	0
Test 3	2	1	0	6	5	4	3
Test 4	3	4	5	6	0	1	2
Test 5	4	1	6	0	3	2	5
Test 6	5	2	3	0	6	1	4
Test 7	0	6	4	5	2	1	3
Test 8	3	1	2	5	4	6	0
Test 9	6	2	5	0	3	4	1
Test 10	1	4	3	0	5	2	6
Test 11	2	0	3	4	6	1	5
Test 12	5	1	6	4	3	0	2

Figure 14. Prueba de siete eventos de la biblioteca de pruebas precalculadas.

5.3 Costo y Consideraciones Prácticas

Al igual que con otras formas de prueba combinatoria, algunas combinaciones pueden ser imposibles o no existir en el sistema bajo prueba. Por ejemplo, 'recibir mensaje' debe ocurrir antes de 'procesar mensaje'. El algoritmo que hemos desarrollado hace posible especificar pares x,y, donde la secuencia x..y debe ser excluida de la matriz de cobertura generada. Por lo general, esto conducirá a pruebas adicionales, pero no aumenta significativamente la matriz de prueba.

5.4 Resumen del Capítulo

1. Las matrices de cobertura de secuencias son una nueva aplicación de métodos combinatorios, desarrollada por NIST para resolver problemas con pruebas de interoperabilidad. Una matriz que cubre secuencias es un conjunto de pruebas que garantizan que se hayan probado todas las secuencias de eventos de *t*-way. Los eventos t en la secuencia se pueden intercalar con otros, pero se probarán todas las permutaciones.

2. Todas las secuencias bidireccionales se pueden probar simplemente enumerando los eventos que se van a probar en cualquier orden y luego invirtiendo el orden para crear una segunda prueba. Se han desarrollado algoritmos para crear matrices de cobertura de secuencias para niveles de interacción de mayor fuerza.

3. Al igual que con otros tipos de pruebas combinatorias, las restricciones pueden ser importantes, ya que es muy común que ciertos eventos dependan de que otros ocurran primero. Las herramientas que NIST ha desarrollado para este problema permiten al usuario especificar restricciones en forma de secuencias excluidas que no aparecerán en la matriz de prueba generada.

Events	3-seq Tests	4-seq Tests
5	8	29
6	10	38
7	12	50
8	12	56
9	14	68
10	14	72
11	14	78
12	16	86
13	16	92
14	16	100
15	18	108
16	18	112
17	20	118
18	20	122
19	22	128
20	22	134
21	22	134
22	22	140
23	24	146
24	24	146
25	24	152
26	24	158
27	26	160
28	26	162
29	26	166
30	26	166
40	32	198
50	34	214
60	38	238
70	40	250
80	42	264
90	44	
100	44	

Table 8. Número de pruebas para secuencias combinatorias de 3 y 4 vías.