CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS
AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

Unidad Zacatenco

Departamento de Computación

Herramienta para generación y ejecución automática de pruebas aleatorias de programas en lenguaje C

Tesis que presenta

Fermín Moreno Cabrera

para obtener el Grado de

Maestro en Ciencias

en Computación

Director de la Tesis

Dr. Pedro Mejía Álvarez

México, D.F. **Noviembre, 2014**

Resumen

Las pruebas de software son una parte esencial del ciclo de desarrollo del software. Las pruebas forman parte del proceso de verificación y validación del software sin las cuales no se podría alcanzar un nivel de fiabilidad y calidad aceptable. Es además, junto con la depuración, la actividad del desarrollo del software que requiere mayor tiempo en llevarse a cabo.

Una de las soluciones para tratar de disminuir el tiempo que ocupa el proceso de pruebas es su automatización. Es por eso que hoy en día, la compleja y pesada tarea de realizar pruebas de software se basa cada vez más en el uso de herramientas automáticas.

En este trabajo se desarrolló una herramienta, a la que llamamos *AutoTest4C*, que realiza un proceso automático de pruebas en programas escritos en lenguaje C. El código es probado con datos de prueba generados usando la estrategia de pruebas aleatorias. Además, combina las ventajas de las pruebas manuales y las pruebas automáticas. Se utilizan dos mecanismos para verificar el comportamiento adecuado del software bajo prueba: un mecanismo mediante el cual se verifica si el resultado arrojado por la función probada es correcto (ingresado por el usuario); un manejador de excepciones para detectar fallos en tiempo de ejecución. El usuario no tiene que preocuparse en elegir que casos de prueba ejecutar, sólo tiene que proveer el software a probar, el tiempo de prueba y otras especificaciones opcionales. Todo esto se hace mediante una *GUI* que facilita el uso de la herramienta.

Abstract

Software testing is an essential part of the software development cycle. Testing is part of the process of software verification and validation without which an acceptable level of reliability and quality could not be reach. It is also, along with debugging, the software development activity that requires more time to be performed.

One solution for trying to decrease the time it takes the testing process is automation. That's why today the complex task of software testing is increasingly based on the use of automatic tools.

In this thesis work we developed a tool we call AutoTest4C, which perform an automated testing process in programs written in C. The code is tested with test data generated using random testing strategy. It combines the advantages of manual testing and automated testing. Two mechanisms to verify proper behavior of the software under test are used: a mechanism by which it is checked whether the test result is correct (entered by the user); an exception handler to detect faults at runtime. Users do not have to worry about choosing which test cases to execute, they only have to provide software testing, testing time and other optional specifications. All this is done through a GUI that facilitates use of the tool.

Agradecimientos

Agradezco al Consejo de Ciencia y Tecnología (CONACYT) por la ayuda económica que me proporcionó y que permitió la realización de esta tesis.

Al Centro de Investigación y de Estudios Avanzados del I.P.N. (CINVESTAV-IPN), por permitirme forma parte de una institución tan grande e importante.

Al Dr. Pedro Mejía Alvarez, por su guía a lo largo de este trabajo. A todos mis compañeros y amigos del CINVESTAV, me han hecho sentir especial.

A mi familia, en especial a mis padres y hermanos, no se imaginan cuán orgulloso me siento de todos ustedes, cuán agradecido me siento de tenerlos.

¡Gracias Dios!, porque más que pedirte, tengo que agradecerte.

Quisiera expresar con las palabras perfectas, más humildes, más elegantes, ..., la gratitud que inunda mi alma. Pero, cada vez que intento expresar lo que siento y me veo acorralado por los límites de las palabras, y al no encontrar la manera, solo me queda expresar lo siguiente:

"Algún día encontraré las palabras adecuadas, y serán simples." - Jack Kerouac

Índice general

Índice de figuras

Índice de tablas

Lista de abreviaturas

ART	***A**daptive **R**andom **T**esting*
ARTOO	***A**daptive **R**andom **T**esting for **O**bject-**O**riented software*
DART	***D**irected **A**utomated **R**andom **T**esting*
DSSRS	***D**irt **S**pot **S**weeping **R**andom **S**trategy*
FDRT	***F**eedback-**D**irected **R**andom **T**esting*
FSCS-ART	***F**ixed **S**ize **C**andidate **S**et **A**daptive **R**andom **T**esting*
GUI	***G**raphical **U**ser **I**nterface*
IDE	***I**ntegrated **D**evelopment **E**nvironment*
MART	***M**irror **A**daptive **R**andom **T**esting*
QRT	***Q**uasi **R**andom **T**esting*
RANDOOP	***RAND**om tester for **O**bject **O**riented **P**rograms*
RT	***R**andom **T**esting*
RRT	***R**estricted **R**andom **T**esting*
SUT	***S**oftware **U**nder **T**est*
V & V	***V**erification and **V**alidation*

Capítulo 1

Introducción

El software, como muchos saben, es un ingrediente clave en muchos de los dispositivos y sistemas que se encuentran en nuestra sociedad. Se encuentra en relojes inteligentes, hornos, automóviles, reproductores de DVD, celulares y hasta en cosas más complejas como aviones, naves espaciales, sistemas de control aéreo, entre otros.

El origen de los errores en el software comenzó con el desarrollo del mismo. Este no es el caso en el que iniciamos con un producto perfecto e inventamos maneras de echarlo a perder. Sino que el único modo de que los errores entren en un programa es que sean introducidos por el autor. Hasta el día de hoy, no hay manera de desarrollar un programa libre de errores, incluso si no los descubrimos durante su desarrollo o en las pruebas no quiere decir que no estén ahí.

Una frase de Alan Perlis, el primer ganador del premio Turing, describe de mejor manera el mensaje del párrafo anterior:

"Hay dos formas de escribir programas libres de errores; sólo la tercera funciona."

La frase anterior nos dice que no hay manera de desarrollar software libre de errores. Entonces, ¿Cómo garantizamos que el software producido cumpla con sus requerimientos?, ¿Cómo garantizar que un error no va a ocasionar un fallo en el programa mientras está en medio de una operación muy importante como una transacción de millones de pesos en un banco? No podemos garantizar eso, pero, con el uso de las pruebas, podemos darle cierto nivel de confiabilidad. Las pruebas de software son una de las áreas de la ingeniería

de software que más interés tiene y de la que se han realizado una cantidad importante de investigaciones. Algunos definen las pruebas como una fuente que provee de información acerca de la calidad del producto o servicio. Pero estrictamente hablando, probar un programa es tratar de hacer que este falle [3]. Ese es el principal objetivo de las pruebas: encontrar el máximo número de fallos de ejecución posibles.

Existen diversas categorizaciones, que varios autores han expuesto, con respecto a los tipos de pruebas. Algunas dependen de la actividad del desarrollo de software donde se ocupen, del tamaño del software a probar, de la madurez con que se vea el mismo proceso de pruebas, de la estrategia de pruebas que se elija, entre otros. Y es que el número de diferentes tipos de pruebas varía tanto como los diferentes enfoques de desarrollo. Pero, independientemente de la categorización de las pruebas que se tome, cualquier estrategia de software debe incorporar una planeación de pruebas, que a su vez requiere de: el diseño de casos de prueba, la ejecución de pruebas, y una recolección y evaluación de datos resultantes. Para crear casos de prueba es necesario generar datos de prueba que lo conformen. Hay distintas metodologías para generar datos de prueba (algunas de las cuales se mencionarán en el capítulo 3 y otras en el capítulo 4). La metodología que se eligió para este trabajo fue la llamada "pruebas aleatorias", la cual consiste en generar datos de prueba aleatoriamente. Esta metodología es una de las más comúnmente usadas, entre otras razones, porque puede automatizarse.

Las pruebas (junto con la depuración) es la actividad del desarrollo del software que requiere mayor tiempo para llevarse a cabo. Una de las soluciones para tratar de disminuir el tiempo que ocupa el proceso de pruebas es su automatización (hablaremos más de este tema en el capítulo 3).

En el libro de Pressman [4], se menciona que Davis [5] sugiere una serie de principios de prueba, entre los cuales se encuentra el siguiente:

> *"Las pruebas deberían empezar en pequeño y progresar a probar en grande."*

Nuestro trabajo se enfoca, precisamente, en probar en pequeño, a lo que llamamos pruebas pequeñas o pruebas unitarias. Este tipo de pruebas verifica el comportamiento de una sola unidad de código en un ambiente de trabajo creado artificialmente. Al ser la herramienta

desarrollada destinada a probar código escrito en lenguaje C, para nosotros, esta unidad de código es una función en lenguaje C.

1.1. Motivación de la tesis

Aún hoy en día nos hacemos la pregunta: ¿Debemos confiar en el software? Los errores en el software han sido responsables de varios desastres, desde barcos encallados hasta explosiones de cohetes, o aun peor, pérdidas humanas. Y es que el software, como sus creadores, tiende a fallar.

Las pruebas nos ayudan a detectar estos errores para evitar los desastres que estos puedan ocasionar. Hay una amplia variedad de pruebas de software. En este trabajo se emplearán las pruebas aleatorias, las cuales constituyen una técnica de pruebas de software en donde los programas son probados por la generación de entradas independientes y aleatorias.

Las pruebas aleatorias son muy útiles cuando el tiempo para escribir y ejecutar pruebas es muy largo. Estas son rentables incluso si no se encuentran muchos defectos en un intervalo de tiempo específico, ya que pueden ser llevadas a cabo de forma automática. En ocasiones una hora de tiempo máquina es menos costosa que una hora de tiempo humano. Y con esto también evitamos el problema de que "el desarrollador es el peor probador" (al no incluir la intervención del hombre en la generación y aplicación de las pruebas, se reducen los posibles errores que pudiera ingresar en las mismas).

El lenguaje C sigue siendo hoy en día uno de los lenguajes más usados, se utiliza para desarrollar compiladores, sistemas operativos, y la mayoría de los motores de programas complejos como los de empresas de reservaciones en línea de aerolíneas, hoteles, etc. Sin embargo, a pesar de su uso, existen pocas herramientas que automatizan alguna parte del proceso de pruebas en programas escritos en C. Hay algunas herramientas como *DART* [6], pero que en realidad no están disponibles [1].

1.2. Planteamiento del problema

En las pruebas de software, uno se enfrenta a menudo con el problema de la selección de un conjunto de datos de prueba a partir de un dominio grande.

Un conjunto de datos de prueba es un subconjunto del dominio, cuidadosamente seleccionados para probar cierto software. Encontrar un adecuado conjunto de datos de prueba es un proceso crucial, ya que aspira a representar a todo el dominio, y esto es importante para evaluar las propiedades estructurales o funcionales del software bajo prueba.

La generación de estos datos de prueba de forma manual es una tarea muy laboriosa y consume mucho tiempo, por lo tanto, es preferible que esto se haga de una manera automatizada.

Las estrategias para generar datos de prueba se clasifican en: orientada a ruta (*Path-oriented*), orientada a objetivos (*Goal-oriented*) y aleatoria (*Random*) [7, 8]. La generación de datos de prueba, usando la estrategia aleatoria, se hace de forma aleatoria y de todo el dominio (aunque esto no garantiza que se genere un conjunto de datos adecuado). La generación aleatoria de datos es la estrategia más sencilla y la más usada. Esto es porque, por ejemplo, para generar automáticamente valores aleatorios para un dato tipo *int* solo es necesario un generador de números pseudoaleatorios (generan números lo suficientemente aleatorios para ser usados en las pruebas). Un generador de números pseudoaleatorios muy usado es el *rand* de C [1].

Sin embargo, la estrategia aleatoria, también ha sido fuertemente criticada por considerarla débil en proveer alta cobertura de código, i.e., no garantiza que todas las partes del código del programa bajo prueba se prueben. Por esta razón es que han surgido variaciones de la estrategia aleatoria que buscan dirigir, por medio de alguna metodología, la generación de los datos de prueba y así dar una mejor cobertura.

DSSRS (*Dirt Spot Sweeping Random Strategy*) [9] es una de estas variaciones creadas (esta técnica y algunas más se verán en el capítulo 4), basada en la estrategia aleatoria,

[1] http://www.cplusplus.com/reference/cstdlib/rand/

que surgió para atacar los problemas mencionados anteriormente, y que, en este trabajo, se considera usar para mejorar la efectividad de las pruebas.

1.3. Objetivos de la tesis

1.3.1. Objetivo general

El objetivo es el diseño y realización de un *framework* y herramientas para la generación y ejecución automática de pruebas aleatorias de programas secuenciales en lenguaje C.

1.3.2. Objetivos específicos

- Definir una estrategia de generación de casos de prueba a realizar.
- Definir un mecanismo de verificación de resultados de pruebas.
- Diseñar la arquitectura de la herramienta de pruebas.
- Implementar un modulo de generación aleatoria de casos de prueba.
- Implementar un modulo de estrategia de selección del mejor caso de prueba.
- Implementar un modulo de ejecución de casos de prueba.
- Implementar la interfaz gráfica de usuario.
- Definir un caso de estudio.

1.4. Justificación

Las herramientas nos permiten automatizar actividades y procesos que son complejos de realizar o que simplemente ocupa de mucho tiempo el llevarlos a cabo manualmente. El realizar pruebas de software es un proceso laborioso que puede automatizarse por medio de una herramienta que realice cada actividad de este proceso de una manera automática. La automatización se consigue reduciendo al mínimo la intervención del usuario en las

pruebas.

Nuestra propuesta es una herramienta que realiza pruebas automáticas en programas escritos en el lenguaje de programación C. Las actividades del proceso de pruebas que se automatizan son tres: (1) la generación de las pruebas, (2) la ejecución de las pruebas y (3) la verificación de los resultados de las pruebas y del comportamiento del programa durante la ejecución de las mismas. La arquitectura de la herramienta se diseño teniendo como objetivo implementar una herramienta que realice pruebas automáticas, sin tomar en cuenta el lenguaje de los programas a probar, el tipo de pruebas a realizar o las técnicas y metodologías a utilizar para realizar las tres actividades del proceso de pruebas que se automatizan. Esto nos permite, en un futuro, poder integrar otros módulos que añadan otras características, e.g., realizar otro tipo de pruebas o generar pruebas con una metodología distinta.

Los módulos de la herramienta implementados siguen el diseño de la arquitectura, pero su implementación está enfocada a realizar pruebas automáticas de caja negra en programas escritos en C. Se seleccionó a las pruebas aleatorias como la metodología para la generación de las pruebas porque dentro de las pruebas de caja negra es muy usada y porque la técnica de generación de datos aleatorios puede ser utilizada incluso en pruebas de caja blanca. Además, como se mencionaba en la sección 1.2, es la metodología más simple de implementar y su ejecución no requiere de muchos recursos de cómputo.

Existen distintas herramientas que automatizan algunas actividades de las pruebas. Por ejemplo, hay algunas que solo automatizan la ejecución de las pruebas, como *JUnit* [10] (una biblioteca para realizar pruebas en programas en Java). Hay otras herramientas muy completas como *AutoTest* [11–15] y otras que se mencionan en el capítulo 5, pero realizan pruebas para programas escritos en lenguajes distintos de C. A pesar de que el lenguaje C es uno de los lenguajes más utilizados hoy en día, existen pocas herramientas que realizan pruebas automáticas en programas escritos en este lenguaje. La herramienta que soporta pruebas automáticas en lenguaje C más citada es *DART* [6], su mayor desventaja es que no está disponible. La existencia de una gran cantidad de herramientas que automatizan alguna parte del proceso de pruebas, pero tan pocas para el lenguaje C específicamente, nos

motiva y nos permite concluir que una herramienta como la que proponemos es posible.

1.5. Organización de la tesis

El contenido de la tesis está organizado en ocho capítulos. A continuación se presenta un resumen del contenido de cada capítulo:

- Capítulo 2: en este se describen algunos de los tipos de pruebas que existen, destacando aquel tipo que se va a desarrollar en nuestra herramienta. También se detallan los principales aspectos de la automatización de las pruebas, en lo que las herramientas existentes se enfocan más y en lo que se enfoca este trabajo.
- Capítulo 3: en este capítulo se menciona la importancia de la automatización del proceso de pruebas. Se presentan los principales aspectos que debe tener una herramienta que realice pruebas automáticas de software.
- Capítulo 4: esta parte de la tesis introduce la metodología elegida para generar casos de prueba. Se listan sus ventajas y sus desventajas, y se describen los métodos nacidos a partir de esta metodología.
- Capítulo 5: aquí se presentan algunas herramientas existentes que automatizan una parte del proceso de pruebas. Se exponen sus características, sus metodologías de trabajo y la solución que aportan. Sólo las herramientas más relevantes para esta tesis se describen.
- Capítulo 6: en este capítulo se expone a detalle el diseño y parte de la implementación de la herramienta desarrollada en este trabajo. Se describen sus características, su arquitectura, su metodología y la solución que se propone.
- Capítulo 7: se presentan los casos de estudio elegidos (el software probado), y se muestran los resultados obtenidos, tanto en el desarrollo propio de la herramienta como con los casos de estudio probados.
- Capítulo 8: aquí se presentan las conclusiones del presente trabajo de tesis, se resaltan los principales resultados obtenidos y se proponen una serie actividades a realizar como trabajo a futuro.

Capítulo 2

Pruebas de software

2.1. Introducción

Ya es bien sabida la importancia que tiene el software hoy en día. Prácticamente, se encuentra en todo lo que nos rodea, no sólo en nuestros dispositivos móviles o en nuestras computadoras, sino también en los cajeros automáticos de los bancos, en la red social que frecuentamos, en las consolas de videojuegos donde solemos perder la noción del tiempo, en los aparatos que utilizan los médicos para medir los ritmos cardíacos de un paciente, en los aviones, en los automóviles, en las plantas nucleares, etc.

También sabemos que es un desarrollador aquel que programa ese software, también conocido como programador y como cualquier persona, éste tiende a equivocarse. Ya lo dice una frase famosa de Marco Tulio Cicerón:

"Humano es errar,...."

Es obvio que al desarrollar software el programador introducirá errores (no intencionalmente) y que estos errores permanecerán en el programa mientras no se encuentren y no se corrijan. Estos errores son tolerables dependiendo de los daños que puedan ocasionar y definitivamente no lo son cuando hay grandes perdidas de dinero o incluso vidas. Algunos de los casos donde los errores han causado grandes perdidas son los siguientes:

- La explosión del Ariane 5 [16, 17].
- Falla en máquina de radioterapia Therac-25 [18].

- Falla del misil Patriot [17].

- Falla en biblioteca OpenSSL [17, 19].

Un reporte del 2002 de la *National Institute of Standards and Technology* [20] estima que el costo anual de las pruebas de software inadecuadas, en la economía de los EUA, está en un rango aproximado de veinte a sesenta mil millones de dolares. Debido a los costos que representan los errores en el software, es que este debe ser probado para descubrir y corregir tantos errores como sean posibles, antes de entregarse al cliente.

Con el uso de las pruebas, los desarrolladores pueden obtener información acerca de la calidad del software. Sin embargo, y pese a esta creencia popular, las pruebas no dan mucha información acerca de la calidad de un producto. Su único propósito es encontrar la mayor cantidad de fallos posibles.

La frase: “La calidad no puede ser probada”, es un cliché que es verdad. Y es que todo producto que se fabrica, sean automóviles o software, si no es construido de la manera correcta desde un principio, entonces nunca será el correcto y por consecuencia no tendrá calidad. Sin embargo, a pesar de que la calidad no puede ser probada, también es cierto que uno no puede construir algo de calidad sin las pruebas. Es por eso que en los trabajos de Beck [21] y Whittaker [22] se sugiere unir el desarrollo (codificación) y las pruebas en un sólo proceso de desarrollo del software, ya que consideran que las pruebas y el desarrollo van de la mano. La calidad no es igual a probar. Las pruebas deberían ser un aspecto inevitable del desarrollo; uniendo en matrimonio al desarrollo con las pruebas es como se alcanza la calidad [22].

En las siguientes secciones de este capítulo explicaremos más a fondo la definición de pruebas, su propósito, algunas categorizaciones de los tipos de pruebas que existen, las formas de ejecución que se le pueden dar a estas, los enfoques de pruebas que existen, la automatización de las mismas y su importancia en el desarrollo de software. Algunos de estos aspectos de las pruebas de software se resumen en la tabla 2.1. Por último, expondremos algunos de los tipos de defectos que se pueden encontrar en el software.

Perspectiva de las pruebas	Niveles de pruebas	Ejecución de pruebas
1. Caja negra a. Partición equivalente b. Análisis de valores límite c. Prueba de comparación d. Prueba de la tabla ortogonal 2. Caja blanca a. Pruebas de caminos b. Prueba de condición c. Prueba de bucles	1. Pruebas unitarias 2. Pruebas de integración 3. Pruebas de interfaces 4. Pruebas de validación 5. Pruebas del sistema	1. Estática 2. Dinámica

Tabla 2.1: Aspectos importantes de las pruebas de software

2.2. Definición de pruebas de software

Las pruebas de software forman parte de un tema muy amplio, frecuentemente conocido como verificación y validación (V & V) [23]. La verificación se refiere a un conjunto de actividades que aseguren que el software implementa correctamente una función específica. La validación se refiere a un conjunto diferente de actividades que aseguran que el software ha sido construido de acuerdo a los requerimientos del cliente. Boehm [24] lo explica de la siguiente manera:

- **Verificación**: ¿Estamos construyendo el producto correctamente?
- **Validación**: ¿Estamos construyendo el producto correcto?

Existen dos tipos de pruebas que se pueden utilizar:

- **Pruebas de validación**: intentan demostrar que el software es el que el cliente quiere, i.e., que cumple con sus requerimientos.
- **Pruebas de defectos**: intentan revelar defectos en el software. Su objetivo es detectar inconsistencias entre el software y su especificación.

Según el estándar de la IEEE 1059-1993 “Guía para los planes de verificación y validación de software” [25], se define a las pruebas de la siguiente manera:

"Es el proceso de analizar un elemento de software para detectar las diferencias entre las condiciones existentes y las requeridas (es decir, bugs), y evaluar las características del elemento de software."

En el estándar IEEE 610.12-1990 "Glosario de términos de Ingeniería de Software" [26], definen el término "prueba" de la siguiente manera:

"Actividad en la cual un sistema o componente es ejecutado bajo condiciones específicas, los resultados son observados o grabados, y se realiza una evaluación de algún aspecto del sistema o componente."

Otras definiciones, dadas por Myers [27] y Meyer [12], nos dicen que las pruebas de software es el proceso de ejecutar programas de software tratando de hacer que falle con el objetivo de intentar encontrar el máximo número de defectos.

A continuación, vamos a definir ciertos términos cuyos significados suelen confundirse:

- **Error**: se refiere a un error humano, una equivocación, i.e., algo que una persona piensa o hace y que no debería de hacer [28].
- **Defecto**: es la manifestación de un error [28]. Más precisamente, el defecto es una representación de un error, una carencia o algo que está mal ya sea en los diagramas de flujo, en casos de uso, en el código fuente, etc. Se menciona "carencia" porque los defectos pueden ser elusivos. Cuando se comenten errores de omisión, el defecto resultante es algo que falta y que debería estar presente. Por lo tanto, como Jorgensen menciona [28], debemos hablar de defectos de comisión y defectos de omisión. Los primeros ocurren cuando se introduce algo en una representación que es incorrecta. Los defectos de omisión ocurren cuando no introducimos alguna información que es correcta. Como es de imaginarse, este último tipo de defectos es el más difícil de detectar y de resolver. El término *bug*, muy comúnmente usado en el ámbito de la computación, es usado para referirse a un defecto.
- **Falla**: es un comportamiento incorrecto del software debido a la activación de un defecto. Hay dos cosas que debemos remarcar aquí: la primera es que las fallas sólo ocurren en una representación ejecutable del código fuente; la segunda, la definición de fallas, sólo corresponde a los defectos de comisión [28].

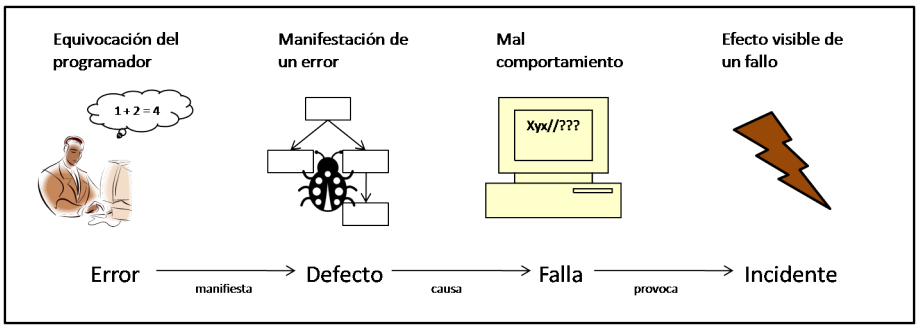

Figura 2.1: Diferencias entre error, defecto, falla e incidente

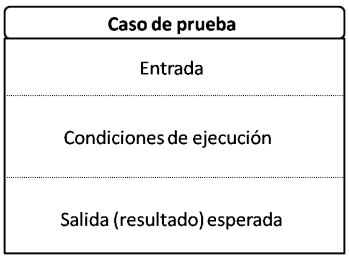

Figura 2.2: Definición de caso de prueba

- **Incidente**: cuando una falla sucede, puede o no ser evidente para el usuario (o cliente o probador). Un incidente es el síntoma, asociado con la falla, que alerta al usuario de la ocurrencia de una falla [28]. Ver figura 2.1.

- **Dato de prueba**: es un dato que ha sido seleccionado específicamente para ser usado en la prueba de un programa de computadora.

- **Caso de prueba**: un conjunto de entradas (datos) de prueba, condiciones de ejecución y resultados esperados desarrollados para un objetivo en particular, como ejecutar una ruta de programa en particular o verificar el cumplimiento de un requerimiento específico [26]. Ver figura 2.2.

- **Oráculo de prueba**: establece el comportamiento aceptable para las ejecuciones de prueba, e.g., los resultados esperados para determinadas entradas son un oráculo de prueba.

- **Dominio de entrada**: llamado también espacio de entrada, es el conjunto de todos los posibles datos de entrada que puede recibir un programa. Esto incluye las

variables globales, los parámetros recibidos por una función o las entradas que puedan ser introducidas externamente (e.g., entrada introducida por teclado). Por ejemplo, consideremos que una función acepta como entrada el vector de entradas $X = x_1, x_2, x_3, ..., x_n$; el dominio de cada entrada es $D_1, D_2, D_3, ..., D_n$, tal que, $x_1 \in D_1$, $x_2 \in D_2$ y así sucesivamente. Entonces, el dominio de entrada de la función puede ser expresado como el producto cruzado de los dominios de cada entrada: $D = D_1 \times D_2 \times ... \times D_n$. Cada entrada añade una dimensión al dominio. Un dominio de entrada es n-dimensional. Normalmente, para ejemplificar, se usa un dominio de dos dimensiones (representado por un cuadrado); donde las líneas del cuadrado representan los límites del dominio y el área del mismo representa a todas las entradas posibles. En las pruebas unitarias, consideraremos como dominio de entrada de una función a todas las posibles entradas que pueda recibir esta función como parámetro.

Myers [27] también nos dice que:

- Un buen caso de prueba es aquel que tiene una alta probabilidad de encontrar un error no descubierto.
- Una prueba exitosa es aquella que descubre un error no descubierto.

De aquí en adelante nos referiremos a las pruebas como a pruebas de defectos, a menos que se especifique explícitamente lo contrario.

2.3. Propósito de las pruebas de software

¿Por qué probamos? Hay dos razones principales por las que realizamos pruebas de software:

1. Para descubrir defectos.
2. Para emitir un juicio acerca de la calidad o fiabilidad del software.

La primer razón conduce a las pruebas de defectos, en las cuales se tratan de revelar defectos. La segunda razón conduce a las pruebas de validación, en las que se espera que el programa funcione correctamente usando un conjunto de casos de prueba que reflejan el

funcionamiento esperado del programa. Probamos porque sabemos que podemos equivocarnos [28].

El propósito de una prueba es tratar de hacer que el software falle, con el fin de encontrar la mayor cantidad defectos para que puedan ser corregidos; de manera que se pueda lograr un mayor nivel de fiabilidad y mejorar la calidad. Idealmente, desde una perspectiva de caja negra (ver sección 2.6.1), un máximo número de defectos pueden ser identificados si el software bajo prueba (*SUT*, por sus siglas en inglés) es probado exhaustivamente, i.e., probar el *SUT* con todas las posibles combinaciones de datos de entrada y comparar los resultados obtenidos con los esperados (obtenidos con el oráculo). Desafortunadamente, las pruebas exhaustivas no siempre son posibles, debido a los recursos limitados de los que se disponen y al tamaño infinito del domino de entradas, i.e., el infinito número de valores de entrada que un programa puede tomar. Las pruebas, por lo tanto, tienen que basarse en un subconjunto de posibles casos de prueba [23].

En este punto debemos mencionar que: "las pruebas no son una ciencia exacta"; no podemos asegurar que al encontrar un número determinado de *bugs*, después de una serie de pruebas, esos sean todos los *bugs* que el *SUT* tiene o que, al no encontrar ninguno, el *SUT* esté libre de errores. Como lo dice Dijkstra [29]:

"Las pruebas sólo pueden mostrar la presencia de errores, no su ausencia."

Lo anterior quiere decir que no podemos probar completamente un programa. "Probar completamente un programa" se refiere a que al final de las pruebas no queda ningún defecto por descubrir.

2.4. Proceso de pruebas de software

Un modelo general del proceso de pruebas se muestra en la figura 2.3. El proceso comienza con el diseño de los casos de prueba (ver sección 2.5), después se eligen los datos para cada caso de prueba. Se ejercita el programa bajo prueba con los datos de prueba elegidos, se comparan los resultados y se realiza un informe de la prueba. Los datos de prueba a veces pueden generarse automáticamente. Sin embargo, la generación

automática de casos de prueba es imposible, ya que las salidas esperadas de las pruebas sólo pueden predecirse por personas que comprenden lo que debería hacer el sistema.

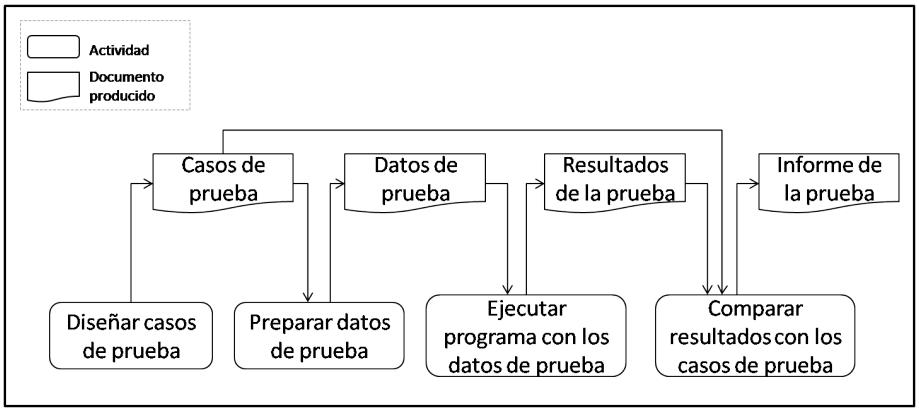

Figura 2.3: Proceso de pruebas de software

2.5. Diseño de casos de prueba

Se refiere a diseñar casos de prueba (entradas y salidas esperadas) para probar un programa. Recordando el objetivo de las pruebas, debemos diseñar un conjunto de casos de pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible. Esta es la consideración más importante de las pruebas de software, diseñar y crear casos de prueba efectivos.

La razón de la importancia de diseñar casos de prueba es precisamente que una prueba "completa" es imposible (i.e., una prueba no puede garantizar la ausencia de errores), y en consecuencia, una prueba de cualquier programa es, por definición, incompleta. La estrategia obvia que Myers [27] nos sugiere es intentar reducir esta carencia tanto como sea posible. Entonces, dadas las restricciones de tiempo, costo, tiempo de computo, grupo de trabajo, etc., la cuestión clave se convierte en: "¿Qué subconjunto, de todos los casos de prueba posibles, tiene la más alta probabilidad de detectar la mayor cantidad de errores?"

En la sección 2.6 se muestran algunas metodologías para el diseño de casos de prueba (orientadas a caja negra y a caja blanca).

Cualquier producto de ingeniería (y de muchos otros campos) puede probarse de una de las siguientes dos formas [4]:

1. Conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo tiempo, se buscan errores en cada función. Es decir, se realizan pruebas, conociendo solamente las funciones que realiza el producto, que verifiquen que el producto cumpla con los requerimientos para los que fue construido.

2. Conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. Es decir, se realizan pruebas, conociendo solamente la forma interna en que opera el producto (e.g., código fuente, componentes mecánicos, etc.), que verifiquen que el comportamiento de los componentes internos del producto cumplan con los requerimientos para los que fue construido.

El primer enfoque de prueba se denomina prueba de caja negra (ver sección 2.6.1) y el segundo, prueba de caja blanca (ver sección 2.6.2).

Para diseñar un caso de prueba, se selecciona una característica del sistema o componente que se está probando. A continuación, se selecciona un conjunto de entradas que ejecutan dicha característica, se documentan las salidas esperadas o rangos de salida y, donde sea posible, se diseña una prueba automatizada que pruebe que las salidas reales y esperadas sean las mismas.

2.6. Perspectiva de las pruebas de software

Las pruebas de software pueden dividirse en “caja negra” y “caja blanca” con base en la perspectiva tomada. En las siguientes secciones se describe el enfoque de pruebas de caja negra (sección 2.6.1) y caja blanca (sección 2.6.2). Además, se mencionan algunas metodologías de prueba de cada enfoque (las cuales se resumen en la tabla 2.2).

Perspectiva de pruebas	**Metodología**	**Principio**
Caja negra	Partición equivalente	Se divide el dominio de entrada del programa bajo prueba en particiones de datos con características comunes (números positivos, números negativos, etc.).
	Análisis de valores límite	Se eligen los valores límite (mínimo y máximo), de cada dato de entrada al programa, como datos de prueba.
	Prueba de comparación	Se desarrollan versiones independientes, del programa a probar, usando las mismas especificaciones. Cada versión se ejercita con los mismos casos de prueba (seleccionados mediante otra técnica de prueba). Si los resultados de las pruebas son diferentes, se analizan las diferencias entre las versiones en busca del defecto responsable de la falla.
	Prueba de la tabla ortogonal	Se crea una tabla ortogonal de casos de prueba. En esta tabla, los casos de prueba están uniformemente dispersos en el dominio de entrada.
Caja blanca	Pruebas de caminos	Se prueba, al menos una vez, cada camino de ejecución independiente en un programa. Un camino es una vía por la cual procede la ejecución a través de una función desde su inicio hasta el fin.
	Prueba de condición	Se prueban todas las condiciones lógicas del programa bajo prueba. Cada clausula de cada condición del programa bajo prueba debe ser ejercitada con ambos valores (`true` y `false`).
	Prueba de bucles	Se prueba la correcta construcción de los bucles (bucles simples, anidados, concatenados y no estructurados).

Tabla 2.2: Metodologías de prueba de caja negra y caja blanca

2.6.1. Pruebas de caja negra

En las pruebas de caja negra, no se necesita saber de la estructura interna del código del *SUT* (ver figura 2.4). Los casos de prueba se obtienen de especificaciones y el programa pasa la prueba si la salida obtenida corresponde con la salida esperada [4]. Este tipo de pruebas es también conocido como pruebas funcionales o pruebas de comportamiento, se centran en obtener conjuntos de condiciones de entrada que ejerciten completamente los requisitos funcionales de un programa [4].

Las pruebas de caja negra intenta encontrar las siguientes categorías de errores [4]:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en acceso a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y de terminación.

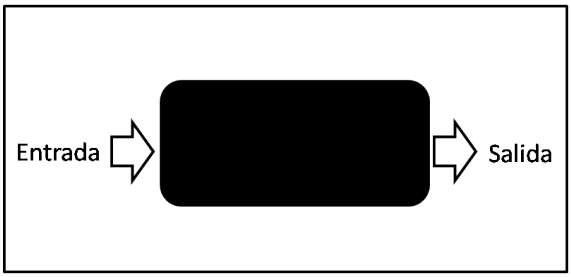

Figura 2.4: Pruebas de caja negra

El diseño de casos de prueba de caja negra puede utilizar distintas metodologías con el fin de diseñar los casos más efectivos u óptimos. Entre las más comunes se encuentran las descritas a continuación.

Partición equivalente

Normalmente, los datos y los resultados de salida de un programa pueden ser agrupados en distintas clases diferentes, cada una con características comunes como números

positivos, números negativos, etc. Usualmente, los programas tienen un comportamiento similar para todos los miembros de una clase. Es decir, si un programa realiza alguna operación que requiere dos números positivos, se esperaría que el programa se comporte de la misma manera con todos los números positivos. Debido a este comportamiento equivalente, las clases formadas se denominan particiones de equivalencia o dominios [23].

En este método de prueba de caja negra, se divide el dominio de entrada de un programa en clases de datos de los que se pueden obtener casos de prueba. La partición equivalente busca obtener casos de prueba ideales que descubran de forma inmediata clases de errores, reduciendo el número total de casos de prueba a desarrollar [4].

Una clase de equivalencia representa un conjunto de estados válidos o no válidos para condiciones de entrada. Una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica [4].

Las clases de equivalencia se pueden definir de acuerdo a las siguientes directrices [4]:

- Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos clases de equivalencia no válidas.
- Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos clases de equivalencia no válidas.
- Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una no válida.
- Si una condición de entrada es lógica, se define una clase de equivalencia válida y una no válida.

En la figura 2.5, cada partición de equivalencia se muestra como una elipse.

Análisis de valores límite

El análisis de valores límites es una técnica de prueba en la cual se eligen casos de prueba que ejercitan los valores límites aceptables del programa [4]. Los valores límite son los valores iniciales y finales de un tipo de dato en particular [9].

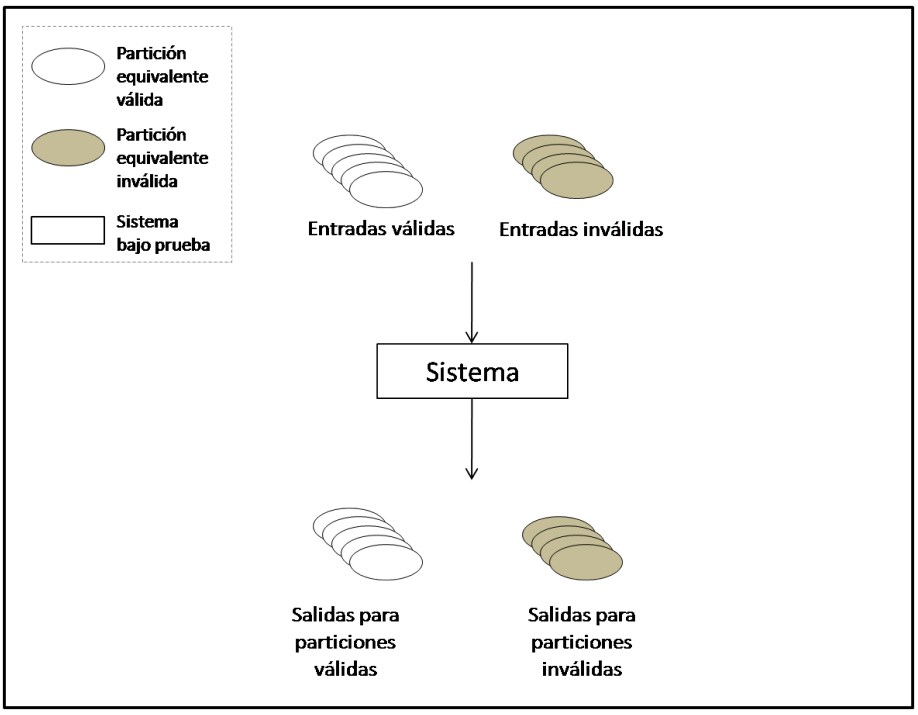

Figura 2.5: Particiones de equivalencia

Esta prueba de caja negra complementa a la partición equivalente; en lugar de seleccionar cualquier elemento de una clase de equivalencia, se seleccionan casos de prueba en los "extremos" de la clase [4].

La idea básica de un análisis de valores límite es usar variables de entrada en su valor mínimo, justo arriba de su mínimo, un valor nominal, justo abajo de su máximo y su máximo [28]. El análisis de valores límite está basado en dos razones subyacentes:

- Los diseñadores y programadores tienden a considerar valores típicos de entrada cuando desarrollan un sistema. Los valores límites son a menudo atípicos (por ejemplo, el cero pudiera necesitar un comportamiento diferente del resto de los números) por lo que los diseñadores y programadores pudieran pasarlo por alto. Los fallos de ejecución de los programas a menudo ocurren cuando se procesan estos valores atípicos. Además, el cambio de comportamiento alrededor de los valores límites tiende a hacer las condiciones de frontera puntos débiles del programa. Los casos de

prueba que ejercitan estos valores límites tienen mucha probabilidad de encontrar defectos.

- El supuesto de que es muy probable que una rutina que maneje de forma adecuada los casos de frontera se comporte también de forma adecuada con los casos típicos.

Las directrices para el análisis de valores límite son las siguientes [4]:

- Si una condición de entrada especifica un rango acotado por los valores a y b, se deben diseñar casos de prueba para estos valores y para los valores justo por debajo y justo por encima de a y b, respectivamente.
- Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo. También se deben probar los valores justo por encima y justo por debajo del máximo y del mínimo.
- Las dos directrices anteriores aplican para las condiciones de salida.
- Si las estructuras de datos internas tienen límites preestablecidos (por ejemplo, un arreglo limitado a 100 elementos), debe diseñarse un caso de prueba que ejercite la estructura de datos en sus límites.

Prueba de comparación

Diversos investigadores [30] han sugerido que para las aplicaciones críticas, se deben desarrollar versiones de software independientes (normalmente desarrolladas por otros programadores), usando las mismas especificaciones. Esas versiones independientes son la base de una técnica de prueba de caja negra llamada "prueba de comparación" o "prueba mano a mano" [4].

A cada versión del software se le proporciona como entrada los casos de prueba diseñados mediante alguna otra técnica de caja negra. Si las salidas producidas por las diferentes versiones son iguales, se asume que todas las implementaciones son correctas. Si las salidas son diferentes, se analizan todas las aplicaciones para determinar el defecto responsable de la diferencia en una o más versiones [4].

Prueba de la tabla ortogonal

La prueba de la tabla ortogonal permite diseñar casos de prueba que proporcionen una cobertura máxima con un número razonable de casos de prueba. Puede aplicarse a problemas en los que el dominio de entrada es relativamente pequeño pero, a la vez, demasiado grande para hacer de las pruebas exhaustivas algo impráctico [4].

Este método es particularmente útil para encontrar errores asociados con fallos localizados (una categoría de error asociada con defectos de la lógica dentro un componente de software) [4].

Pressman [4] propone la siguiente manera de ver esta técnica de prueba. Consideremos un sistema que tiene tres elementos de entrada: x, y y z. Cada uno de estos elementos puede tomar tres valores distintos. De manera que hay $3^3 = 27$ posibles casos de prueba. La figura 2.6 ilustra una visión geométrica de los posibles casos de prueba (representados por puntos negros) asociados con x, y y z.

En el cubo de la izquierda de la figura 2.6, se puede observar que cada elemento de entrada puede modificarse secuencialmente en cada eje de entrada. Lo cual da como resultado un alcance relativamente limitado al dominio de entrada.

Cuando se realiza la prueba de la tabla ortogonal, se crea una tabla ortogonal $L9$ de casos de prueba. Esta tabla tiene una propiedad de equilibrio, i.e., los casos de prueba están uniformemente dispersos en el dominio de entrada (ver cubo de la derecha de la figura 2.6). De esta manera, el alcance de la prueba por todo el dominio de entrada es más completo.

2.6.2. Pruebas de caja blanca

En las pruebas de caja blanca, el probador necesita saber de la estructura interna del software y la puede modificar (en caso de que se requiera), ver figura 2.7. En este enfoque de pruebas, llamado también pruebas estructurales, los casos de prueba se obtienen de la estructura interna del código del software bajo prueba y la prueba pasa sólo si los resultados son correctos [4].

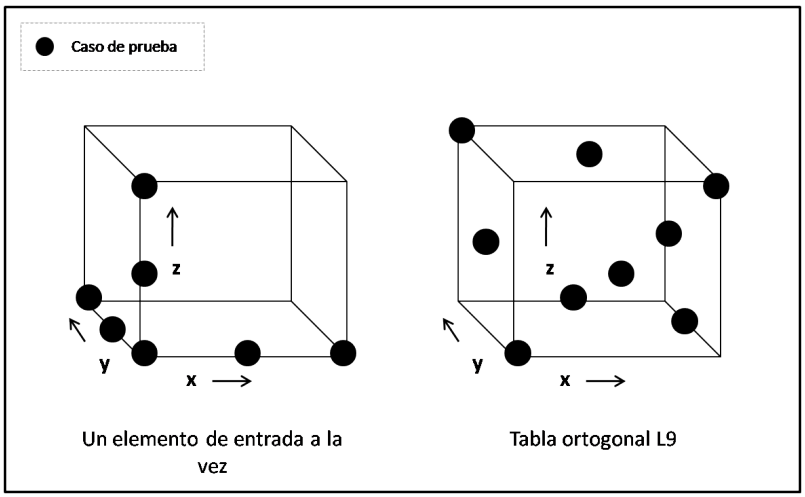

Figura 2.6: Vista geométrica de los casos de prueba en prueba de la tabla ortogonal

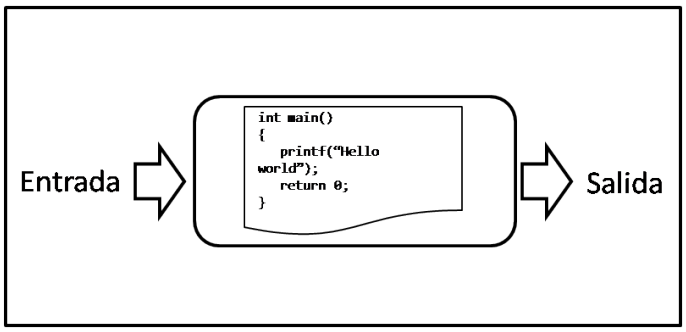

Figura 2.7: Pruebas de caja blanca

Mediante una prueba de caja blanca, se pueden obtener casos de prueba que cumplan alguna de las siguientes funciones [4]:

- Ejerciten por lo menos una vez todos los caminos independientes de cada módulo.
- Ejerciten todas las decisiones lógicas en sus caminos verdadero y falso.
- Ejecuten todos los bucles en sus límites y con sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.

El diseño de casos de prueba de caja blanca puede utilizar distintas metodologías con el fin de diseñar los casos más efectivos u óptimos. Entre las más comunes se encuentran las

descritas a continuación.

Pruebas de caminos

Las pruebas de caminos son una estrategia de pruebas estructurales cuyo objetivo es probar cada camino de ejecución independiente en un componente o programa [23]. Un camino o ruta es una vía por la cual procede la ejecución a través de una función desde su inicio hasta el fin.

El número de caminos en un programa es normalmente proporcional a su tamaño. Puesto que los módulos se integran en sistemas, no es factible utilizar técnicas de pruebas estructurales. Por lo tanto, las técnicas de pruebas de caminos son principalmente utilizadas durante las pruebas de componentes [23].

Las pruebas de caminos no prueban todas las posibles combinaciones de todos los caminos en el programa (es un objetivo imposible en un programa donde el número de posibles combinaciones de caminos es infinito), pero se asegura de que cada camino independiente en el programa se ejecuta al menos una vez. Un camino independiente del programa es aquel que recorre al menos una nueva arista en el grafo de flujo, i.e., que ejecuta una o más condiciones nuevas.

Prueba de condición

La prueba de condición es un método de prueba de caja blanca en la que se ejercitan las condiciones lógicas contenidas en el módulo de un programa [4].

Los tipos posibles de componentes en una condición pueden ser: un operador lógico, una variable lógica, un par de paréntesis lógicos, un operador relacional o una expresión aritmética. Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición. Los tipos de errores de una condición pueden ser los siguientes [4]:

- Error en operador lógico (operador lógico incorrecto, desaparecido o sobrante).
- Error en variable lógica.

- Error en paréntesis lógico.
- Error en operador relacional.
- Error en expresión aritmética.

Existen una serie de estrategias de prueba de condiciones, entre las cuales se encuentran las siguientes:

- **Prueba de ramificaciones:** para una condición compuesta C, es necesario ejecutar al menos una vez las ramas verdadera y falsa de C y cada condición simple de la misma [4].
- **Prueba del dominio:** para una expresión relacional de la forma: $E_1 < operador - relacional > E_2$, se requieren tres pruebas para comprobar que el valor de E_1 es mayor, igual o menor que el valor de E_2. Si el operador relacional es incorrecto y E_1 y E_2 son correctos, estas tres pruebas garantizan la detección de un error del operador relacional [4].

Prueba de bucles

La prueba de bucles es una técnica de prueba de caja blanca que se centra en la validez de las construcciones de bucles [4]. Se pueden definir cuatro tipos de bucles [4]:

- **Bucles simples:** a estos bucles se les aplica el siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucle:
  1. Pasar por alto totalmente al bucle.
  2. Pasar una sola vez por el bucle.
  3. Pasar dos veces por el bucle.
  4. Hacer m pasos por el bucle, donde $m < n$.
  5. Hacer $n - 1$, n y $n + 1$ pasos del bucle.
- **Bucles anidados:** Si aplicáramos el enfoque de pruebas de los bucles simples a los bucles anidado, el número de posibles pruebas aumentaría a medida que aumenta el nivel de anidamiento. Los siguientes pasos, sugeridos por Beizer [31], ayudan a reducir el número de pruebas:

1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos.
2. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras que las variables de iteración de los bucles externos se mantienen en sus mínimos. Añadir otras pruebas para valores fuera de rango.
3. Progresar hacia afuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores típicos.
4. Continuar hasta que se hayan probado todos los bucles.

- **Bucles concatenados:** si cada uno de los bucles concatenados son independientes del resto, se pueden probar cuando el enfoque para bucles simples. Si hay bucles que no sean independientes (e.g., el contador del bucle 1 se use como valor inicial del contador del bucle 2), se recomienda probarlos usando el enfoque para bucles anidados.
- **Bucles no estructurados:** siempre que sea posible, este tipo de bucles se deben rediseñar para que se ajusten a las construcciones de programación estructurada.

2.7. Niveles de pruebas de software

A continuación, se muestra la categorización de niveles de pruebas más comúnmente usada. Esta categorización se basa en las etapas del desarrollo de software tradicional.

2.7.1. Niveles de pruebas basados en la actividad del software

El proceso de pruebas está involucrado en cada etapa del ciclo de vida del software, pero las pruebas realizadas en cada nivel del desarrollo del mismo son diferentes, ya que tienen diferentes objetivos. En esta categorización, un diferente nivel de pruebas se aplica para cada actividad del desarrollo del software [32]:

- **Pruebas de aceptación**: evalúan al software con respecto a los requerimientos.
- **Pruebas de sistema**: evalúan al software con respecto al diseño de la arquitectura.

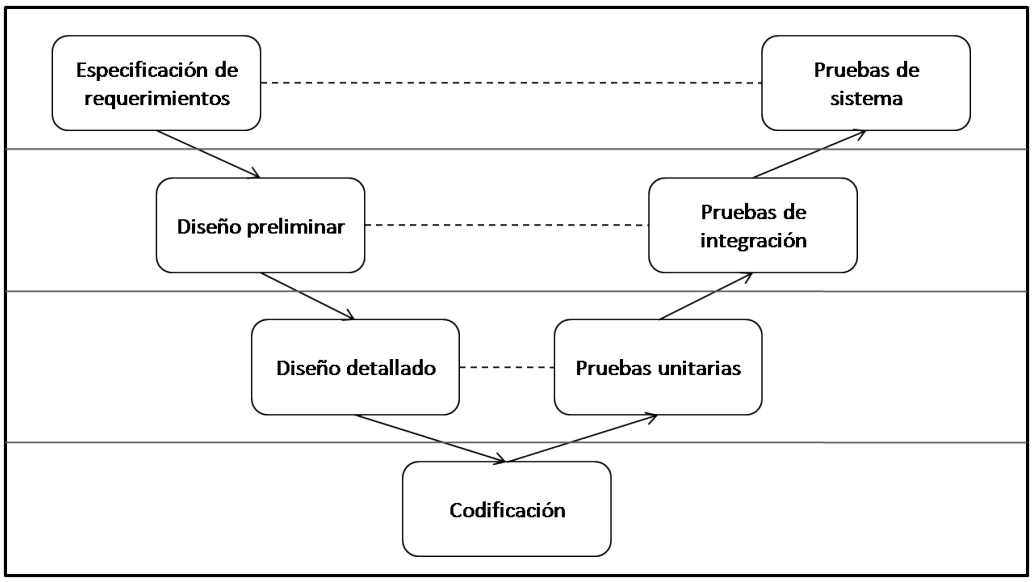

Figura 2.8: Niveles de abstracción y pruebas en un modelo en cascada de Jorgensen

- **Pruebas de integración**: evalúan al software con respecto al diseño de los subsistemas.

- **Pruebas modulares**: evalúan al software con respecto al diseño detallado.

- **Pruebas unitarias**: evalúan al software con respecto a la implementación.

Jorgensen [28] tiene una versión de esta categorización a la que le llama "niveles de abstracción" mostrada en la figura 2.8. En esta tesis, nos centramos en las pruebas unitarias, las cuales ponen a prueba la unidad básica de software, la pieza más pequeña, que con frecuencia se le llama "unidad", "modulo" o "componente". Expondremos más de las pruebas unitarias en la sección 2.7.2.

Según el tipo del sistema a probar, se puede tomar alguna categorización de niveles de pruebas u otra; también depende del enfoque de las pruebas que se tome. Las dos actividades fundamentales de pruebas son la prueba de unidades (probar las partes del sistema) y la prueba del sistema (probar el sistema como un todo) [23], ver figura 2.9. Algunos niveles de pruebas se describen en las siguientes secciones (2.7.2 a 2.7.6).

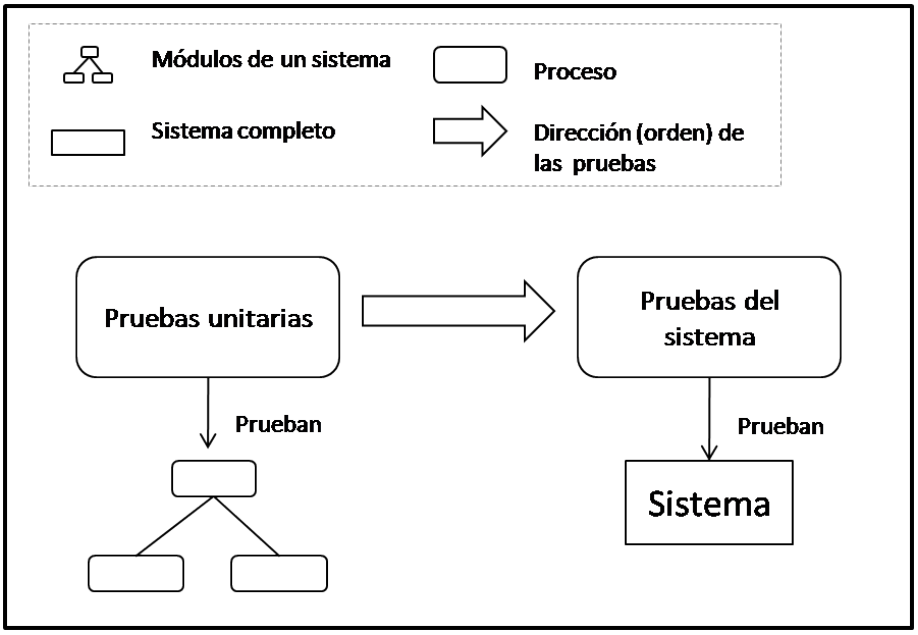

Figura 2.9: Actividades fundamentales en la fase de pruebas de software

2.7.2. Pruebas unitarias

Las pruebas unitarias, también llamadas pruebas de componentes, se encargan de probar, individualmente, subprogramas, subrutinas o procedimientos en un programa [27]. Es decir, en lugar de probar al programa como un todo, se prueban pequeños bloques del programa. El propósito de este tipo de pruebas es comparar la función de una unidad con cierta especificación funcional que define la unidad.

Existen distintos tipos de unidades que pueden probarse [23]:

- Funciones individuales o métodos dentro de un objeto.
- Clases de objetos que tienen varios atributos y métodos.
- Componentes compuestos formados por diferentes objetos o funciones. Estos componentes tienen una interfaz definida que sirve para acceder a su funcionalidad.

Las funciones o métodos individuales son el tipo más simple de unidad y sus pruebas son un conjunto de llamadas a estas rutinas con diferentes parámetros de entrada.

Algunas de las razones que motivan a realizar pruebas unitarias son [27]:

1. Facilitan la tarea de depuración de *bugs*, ya que, al encontrarse un error, se sabe en que unidad se encuentra.

2. Introducen la opción de poder paralelizar el proceso de pruebas por medio de probar múltiples unidades simultáneamente.

Diseño de casos de pruebas unitarias

Se necesitan dos tipos de información cuando se diseñan casos de prueba para una prueba unitaria: una especificación de la unidad y el código fuente de la misma [27]. La especificación define los parámetros de entrada y salida de la unidad.

Los principales casos de prueba para una prueba unitaria (ilustradas en la figura 2.10) incluyen [23]:

- **Probar la interfaz del módulo**: para asegurar que los datos fluyen de forma adecuada hacia y desde la unidad probada.

- **Probar las estructuras de datos locales**: para asegurar que los datos que se almacenan temporalmente conservan su integridad durante la ejecución de la unidad.

- **Probar las condiciones límite**: para asegurar que la unidad funciona correctamente en los límites establecidos como restricciones de procesamiento. Por ejemplo, es frecuente que aparezca un error cuando se procesa el n-ésimo elemento de un arreglo, cuando se ejecuta la i-ésima repetición de un bucle o cuando se encuentran los valores máximo o mínimo permitidos.

- **Probar los caminos independientes**: se ejercitan todos los caminos independientes de la estructura de control con el fin de asegurar que todas sentencias de la unidad se ejecuten por lo menos una vez.

- **Probar los caminos de manejo de errores**: se deben diseñar casos de prueba para detectar errores debidos a cálculos incorrectos, comparaciones incorrectas o flujos de control inapropiados.

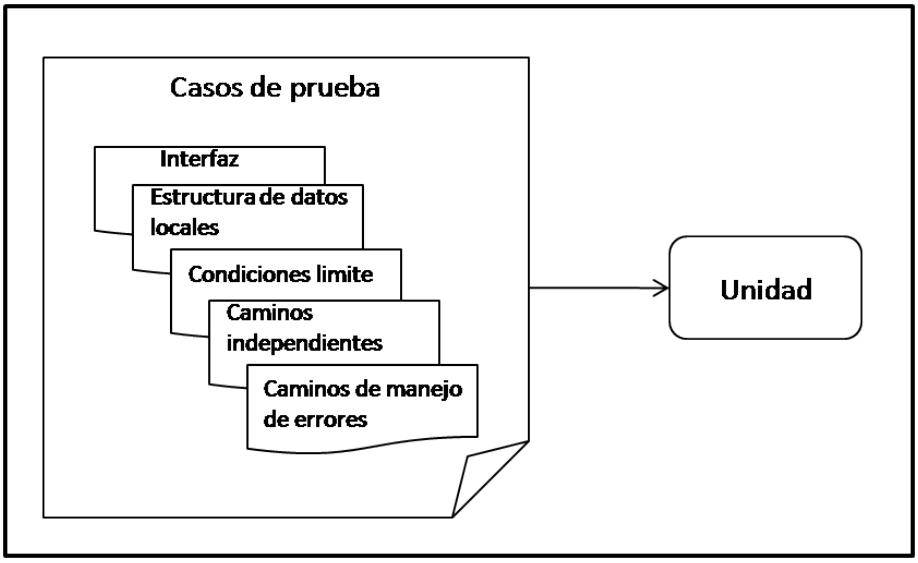

Figura 2.10: Prueba unitaria

Debido a que una unidad de software no es un programa independiente, se debe desarrollar, para cada prueba unitaria, un software que controle y/o resguarde la prueba [23]. La figura 2.11 muestra un entorno para una prueba unitaria. En este entorno, el controlador es el "programa principal" que acepta los datos de prueba, pasa estos datos a la unidad bajo prueba y registra o imprime los resultados. Un resguardo es un subprograma que simula una unidad (i.e., usa la interfaz de la unidad simulada) que es llamada por la unidad bajo prueba. El resguardo realiza una mínima manipulación de datos, imprime una verificación de entrada y devuelve el control a la unidad que lo invocó.

2.7.3. Pruebas de integración

Una vez que se tienen los componentes o unidades y se han probado individualmente, se tienen que integrar para formar al sistema. Las pruebas de integración son una técnica que nos permite, al mismo tiempo que construimos el sistema con los componentes, llevar a cabo pruebas para detectar errores que surgen debido a la integración e interacción de los mismos [4, 23].

La integración del sistema implica identificar grupos de componentes que proporcionen alguna funcionalidad del sistema e integrarlos añadiendo código para hacer que funcionen

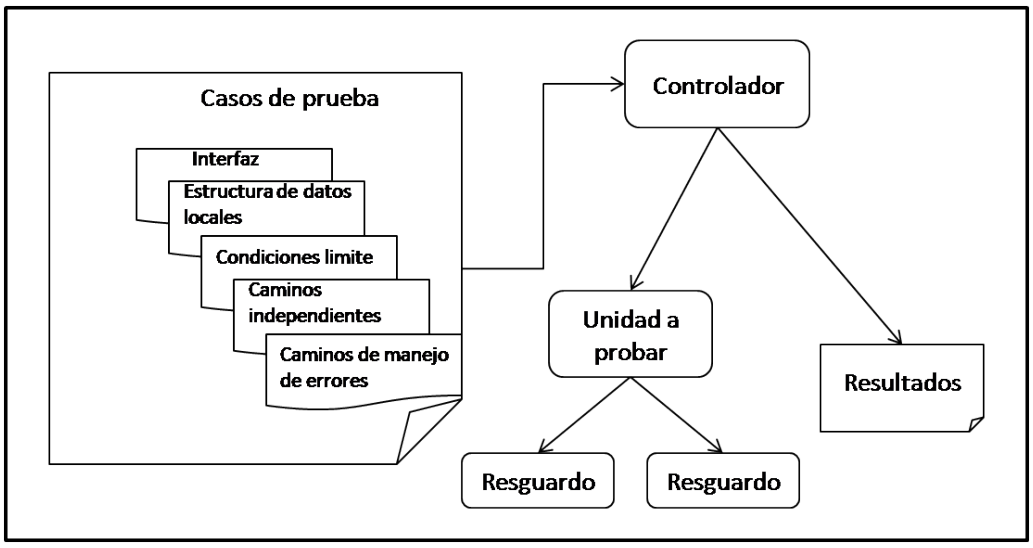

Figura 2.11: Entorno para una prueba unitaria

conjuntamente. Algunos de los posibles problemas que pueden surgir de la integración de los componentes o módulos son: perdida de datos en una interfaz, un componente puede tener efecto adverso e inadvertido sobre otro, las estructuras de datos globales pueden presentar problemas, entre otros. Las pruebas de integración verifican que los componentes funcionen correctamente juntos, que sean llamados correctamente y que transfieran los datos correctos en el tiempo preciso a través de sus interfaces [23].

La integración de los componentes puede ser incremental o no incremental. En el enfoque no incremental, llamado *big bang*, se combinan todos los módulos por anticipado y se prueba todo el programa en conjunto. Este tipo de integración tiene la desventaja de que cuando se encuentran errores, se encuentra un gran número de ellos; lo que supone un gran reto a la hora de corregirlos, puesto que es complicado aislar las causas y localizar donde se encuentran dichos errores al tener el programa completo bajo prueba [4]. En la integración incremental, al contrario de la no incremental, el programa se construye mediante pequeños incrementos o segmentos en los que los errores son más fáciles de localizar y corregir [4]. Existen dos tipos de integración incremental: integración descendente e integración ascendente.

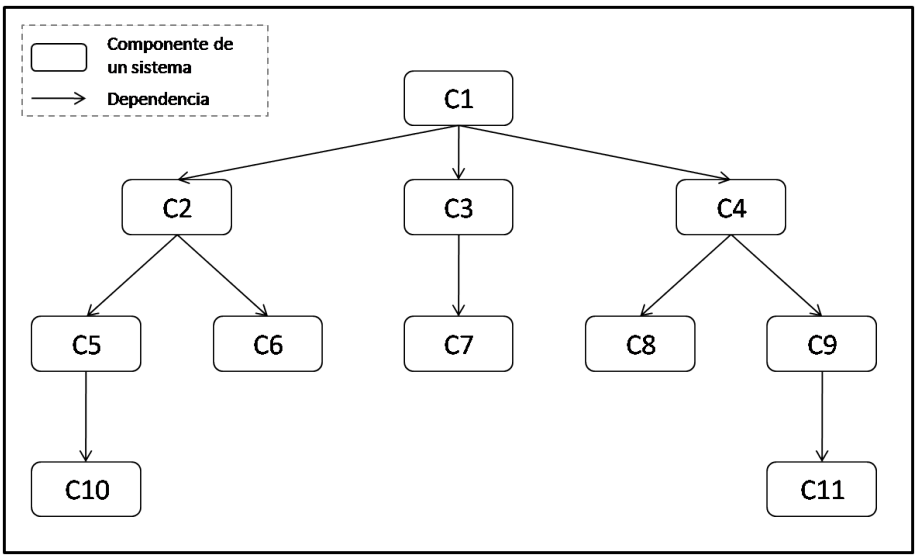

Figura 2.12: Ejemplo de estructura de un programa a nivel de componentes

Integración descendente

En este tipo de integración incremental, primero se desarrolla el esqueleto del sistema completo y después se van añadiendo los componentes. En otras palabras, se integran los componentes avanzando hacia abajo por la jerarquía de control, comenzando por el componente principal. Después, se van agregando los componentes subordinados al componente principal. La incorporación de los componentes subordinados puede ser de dos formas: primero en profundidad o primero en anchura.

La integración descendente primero en profundidad, integra todos los componentes de un camino de control principal de la estructura del sistema. La selección del camino principal se hace de manera arbitraria y depende de las características del sistema [4]. Por ejemplo, supongamos que la estructura de un programa bajo prueba es el mostrado en la figura 2.12. Vamos a integrar los componentes de la forma primero en profundidad, así que, si elegimos el camino de la izquierda, integraremos primero los componentes $C1$, $C2$ y $C5$. A continuación integraremos $C6$ (si es necesario para un funcionamiento adecuado de $C2$) y $C10$. Seguido de esto, construimos el camino central y el derecho, ver figura 2.13.

La integración descendente primero en anchura, integra todos los componentes direc-

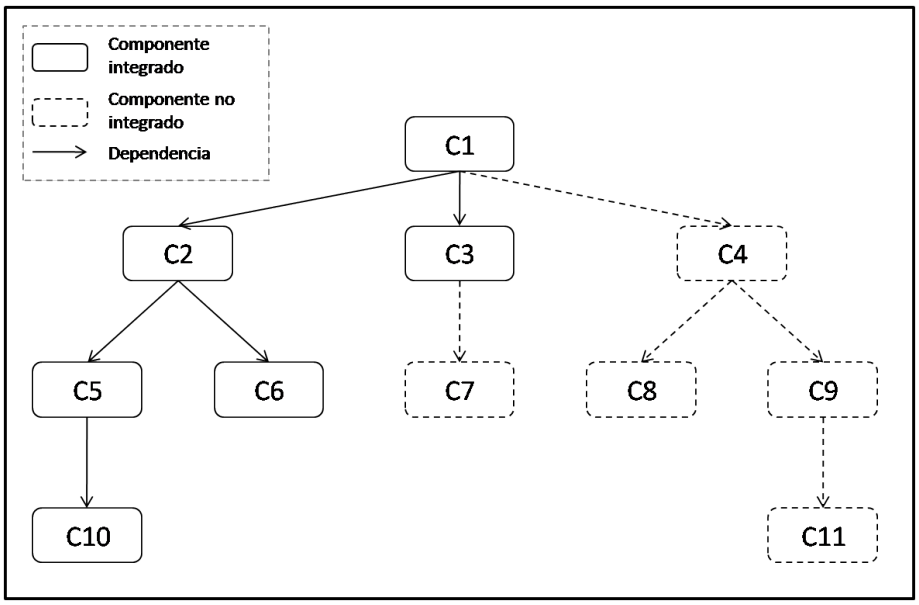

Figura 2.13: Integración descendente primero en profundidad

tamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal [4]. Por ejemplo, siguiendo con el programa con la estructura mostrada en la figura 2.12, los primeros componentes que se integran son $C1$ (puesto que es el componente principal) y sus subordinados directamente: $C2$, $C3$ y $C4$. Después, en el siguiente nivel de control, se incorporan $C5$ y $C6$, para $C2$; $C7$, para $C3$; etc (ver figura 2.14).

Según Pressman [4], el proceso de integración descendente se realiza siguiendo los pasos siguientes:

1. Se usa el componente de control principal como controlador de la prueba, disponiendo de resguardos (programas que simulan la funcionalidad de algún componente real) para todos los componentes directamente subordinados al componente principal.

2. Según el enfoque de integración elegido (primero en profundidad o primero en anchura), se van sustituyendo uno a uno los resguardos subordinados por los componentes reales.

3. Se llevan a cabo pruebas cada vez que se integra un nuevo componente.

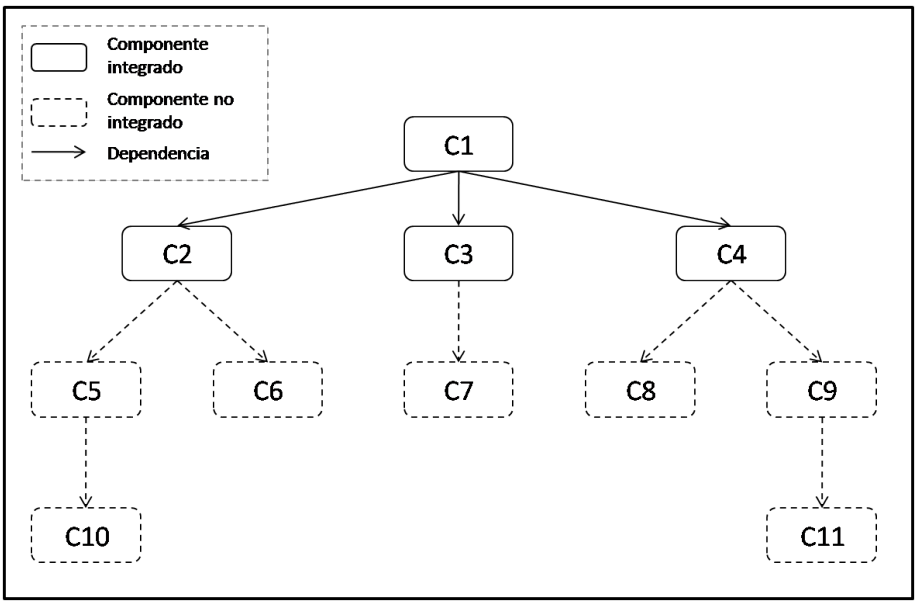

Figura 2.14: Integración descendente primero en anchura

4. Una vez terminado cada conjunto de pruebas, se reemplaza otro resguardo con el correspondiente componente real.

5. Se realizan las pruebas de regresión (las cuales veremos en esta misma sección) para verificar que no se han introducido errores nuevos.

Integración ascendente

En la integración ascendente, el sistema se construye y se prueba comenzando con los componentes de los niveles más bajos de la estructura del programa, i.e., los componentes se integran de abajo hacia arriba siguiendo la jerarquía de control [4]. Debido a la forma en como son integrados los componentes, la funcionalidad requerida de los componentes subordinados siempre está disponible, por lo que no es necesario ocupar de resguardos. Pressman [4] define una estrategia de integración ascendente mediante los siguientes pasos:

1. Se agrupan los componentes de bajo nivel que realicen una subfunción específica del programa.

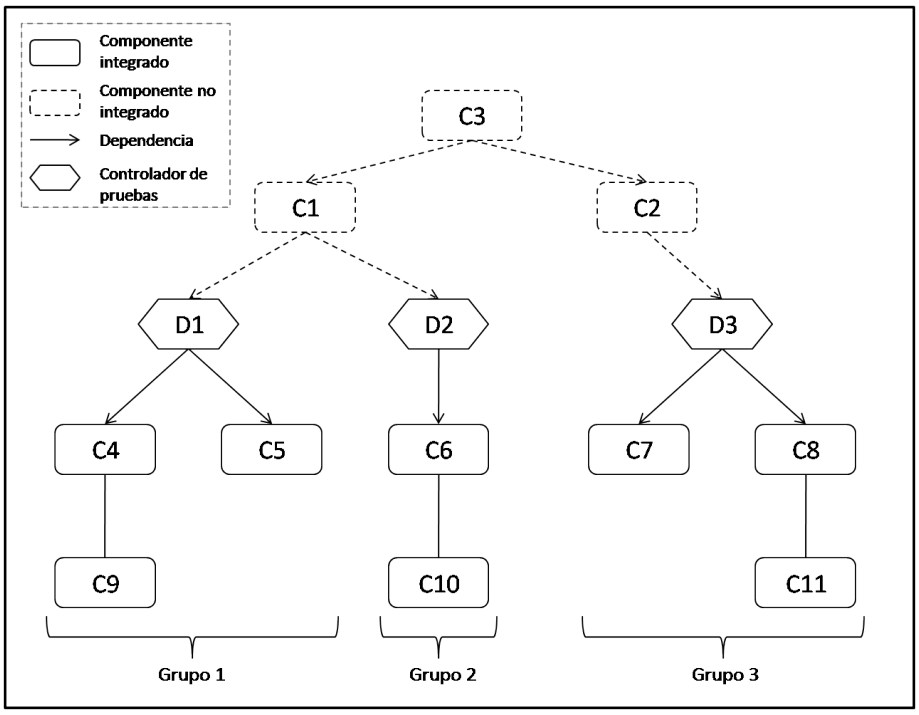

Figura 2.15: Integración ascendente

2. Se escribe un controlador (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba.

3. Se prueba el grupo de componentes.

4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

Un ejemplo de este tipo de integración se muestra en la figura 2.15. Primero, se combinan los componentes para formar los grupos 1, 2 y 3. Cada grupo es sometido a pruebas que son dirigidas por un controlador ($D1$, $D2$ y $D3$). Los componentes de los grupos 1 y 2 son subordinados de $C1$. Al terminar las pruebas de los grupos 1 y 2, los controladores $D1$ y $D2$ se eliminan y los grupos interaccionan directamente con $C1$. De igual forma, se elimina el controlador $D3$ después de terminadas las pruebas del grupo 3, y se integra el grupo 3 con el componente $C2$. A continuación, los componentes $C1$ y $C2$ se integran con el componente $C3$.

Pruebas de regresión

Cada vez que agregamos un componente nuevo como parte de una prueba de integración, el software cambia. Se establecen nuevos caminos de flujo de datos, pueden ocurrir nuevas entradas y salidas y se invoca una nueva lógica de control. En otras palabras, la integración y prueba de un nuevo componente puede cambiar las interacciones de componentes ya probados y, estos cambios, pueden causar problemas con componentes que antes trabajaban perfectamente. Estos problemas significan que, al integrar un nuevo componente, es necesario volver a ejecutar las pruebas para verificar incrementos previos, así como las nuevas pruebas requeridas para verificar el nuevo componente añadido. Se conoce como pruebas de regresión al proceso de volver a ejecutar un subconjunto de pruebas ya existentes con el fin de asegurarse de que los cambios no han ingresado errores o algún otro efecto no deseado. No sólo hay que realizar pruebas de regresión al integrar un nuevo componente, sino cada vez que se realiza un cambio importante en el software, e.g., al corregir errores que han sido descubiertos por una prueba [4, 23]. Según Pressman [4], el conjunto de pruebas de regresión contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que ejercite todas las funciones del software.
- Pruebas adicionales que se centran en las funciones del software que, probablemente, se verán afectadas por el cambio.
- Pruebas que se centran en los componentes del software que han cambiado.

A medida que avanza la prueba de integración, el número de pruebas de regresión puede crecer demasiado. Por esta razón, el conjunto de pruebas de regresión debe diseñarse para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio [4].

2.7.4. Pruebas de interfaces

La mayoría de los componentes de un sistema no son sólo funciones u objetos, sino que son componentes formados por varias funciones y objetos que se comunican entre si. Es a través de la interfaz de cada componente como se accede a su funcionalidad. Las

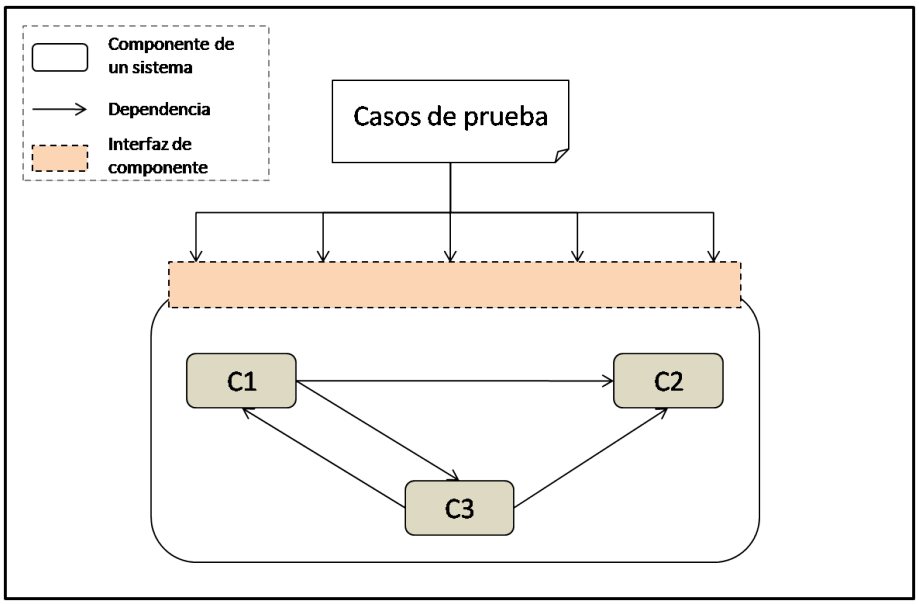

Figura 2.16: Pruebas de interfaces

pruebas de interfaces se encargan de probar que la interfaz del componente bajo prueba se comporta según su especificación. Este tipo de pruebas es importante sobre todo para el desarrollo orientado a objetos y basado en componentes [23].

En la figura 2.16 se muestra un ejemplo de un proceso de pruebas de interfaces. Consideremos que los componentes $C1$, $C2$ y $C3$ se han integrado para formar un componente compuesto o un subsistema. Los casos de prueba, destinados a pruebas de interfaces, no se aplican a los componentes individuales, sino a la interfaz del componente compuesto.

Entre los distintos tipos de interfaces entre componentes que existen, nombramos los siguientes [23]:

- **Interfaces de parámetros**: en este tipo de interfaces, los datos o referencias a funciones se pasan de un componente a otro en forma de parámetros.

- **Interfaces de memoria compartida**: en estas interfaces, los componentes comparten un bloque de memoria. Los datos, que ocupan entre ellos, son colocados en la memoria por un componente y son recuperados por otro.

- **Interfaces procedurales**: son interfaces en las que un componente encapsula un conjunto de procedimientos que pueden ser llamados por otros componentes.
- **Interfaces de paso de mensajes**: son interfaces en las que un componente solicita el servicio de otro componente por medio del paso de un mensaje. El mensaje de retorno, enviado por el componente que proporciona el servicio, incluye los resultados de la ejecución del servicio.

Estos distintos tipos de interfaces, consecuentemente, indican que hay distintos tipos de errores de interfaces que pueden ocurrir. Estos errores se clasifican en tres clases [23]:

- **Mal uso de la interfaz**: un componente trata de llamar a otro componente pero comete un error al usar su interfaz. Por ejemplo, se puede pasar un parámetro de tipo incorrecto, el orden o el número de parámetros puede ser equivocado, etc.
- **No comprensión de la interfaz**: un componente llama a otro componente pero no comprende la especificación de la interfaz del componente al que llama, por lo tanto, hace suposiciones sobre el posible comportamiento del componente al que intenta llamar. El componente invocado no se comporta como se esperaba, lo que provoca un comportamiento inesperado en el componente que lo llamó.
- **Errores temporales**: esta clase de errores se producen en sistemas de tiempo real que utilizan una memoria compartida o una interfaz de paso de mensajes. El componente que produce los datos y el componente que los consume pueden trabajar a diferentes velocidades. Si no se tiene cuidado en el diseño de la interfaz, el componente consumidor puede acceder a información, de la interfaz compartida, que no ha sido actualizada por el componente productor.

2.7.5. Pruebas de validación

Una vez terminada la prueba de integración y la prueba de interfaz, el sistema esta completamente integrado y se puede comenzar a realizar la prueba de validación. La validación se refiere a verificar que el software funciona de acuerdo con los requerimientos del mismo (ordenados por el cliente).

La validación del software se consigue por medio de una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Se necesita un plan de prueba que establezca la clase de pruebas que se han de llevar a cabo, y un procedimiento de prueba que defina los casos de prueba específicos para tratar de descubrir errores relacionados con los requisitos [4]. Una vez que se prueba cada caso de prueba de validación, puede darse uno de los dos resultados siguientes [4]:

- Las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables.
- Se descubre una desviación de las especificaciones y se crea una lista de deficiencias.

Las deficiencias o errores descubiertos en esta etapa del proyecto muy difícilmente se pueden corregir antes de la terminación planificada del mismo. En ocasiones, es necesario negociar con el cliente un procedimiento para realizar las correcciones y solventar las deficiencias.

Pruebas alfa y beta

Cuando se desarrolla un software a la medida para un cliente, es necesario realizar pruebas de aceptación con el fin de que el cliente valide todos los requerimientos. Las pruebas de aceptación son realizadas por el usuario final. Si el software se desarrolla como un producto que será utilizado por varios clientes, no es práctico realizar pruebas de aceptación para cada cliente. Para este caso, los desarrolladores suelen llevar a cabo pruebas para descubrir errores que parezca que sólo el usuario final puede descubrir. Estas pruebas se dividen en dos tipos: prueba alfa y prueba beta.

La prueba alfa es realizada por un cliente en el lugar de desarrollo del programa, en un entorno controlado. El desarrollador actúa como observador y, al mismo tiempo, registra los errores y los problemas de uso que se presenten.

La prueba beta es realizada por los usuarios finales del software en los lugares de trabajo de los clientes. En este tipo de pruebas, el desarrollador no está presente. De esta manera, la prueba beta es una demostración “en vivo” del software, en un ambiente no controlado por el desarrollador. En este caso, es el cliente quien registra los problemas

(reales o imaginarios) que se le presenten e informa al desarrollador de los mismos. El desarrollador del programa recibe los informes, realiza las modificaciones necesarias y prepara una versión del software para todos los clientes [4].

2.7.6. Pruebas del sistema

Según Sommervile [23], las pruebas del sistema implican integrar dos o más componentes del sistema y probarlos como un sistema integrado. En un proceso de desarrollo iterativo, este tipo de pruebas se encarga de probar un incremento del sistema. En un proceso en cascada, las pruebas del sistema se encargan de probar el sistema completo.

Pressman [4] menciona que, la prueba del sistema (refiriéndose como sistema a un software que es incorporado a otros elementos como hardware, información, etc.) está constituida por una serie de pruebas cuyo propósito primordial es ejercitar profundamente el sistema. Cada prueba tiene un propósito distinto, pero todas trabajan para verificar que todos los elementos del sistema se han integrado adecuadamente y que realizan las funciones apropiadas de manera correcta. Algunos tipos de pruebas del sistema son mencionados brevemente a continuación.

Prueba de recuperación

La prueba de recuperación es una prueba del sistema que fuerza al fallo del software de muchas maneras y verifica que la recuperación se lleve a cabo apropiadamente. Si la recuperación es automática, se evalúa la corrección de la inicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación de datos y del proceso de rearranque. Si la recuperación necesita de intervención humana, se evalúan los tiempos medios de reparación para determinar si están dentro de unos límites aceptables [4].

Prueba de seguridad

La prueba de seguridad verifica que los mecanismos de protección del sistema lo protegerán de accesos ilegales [4].

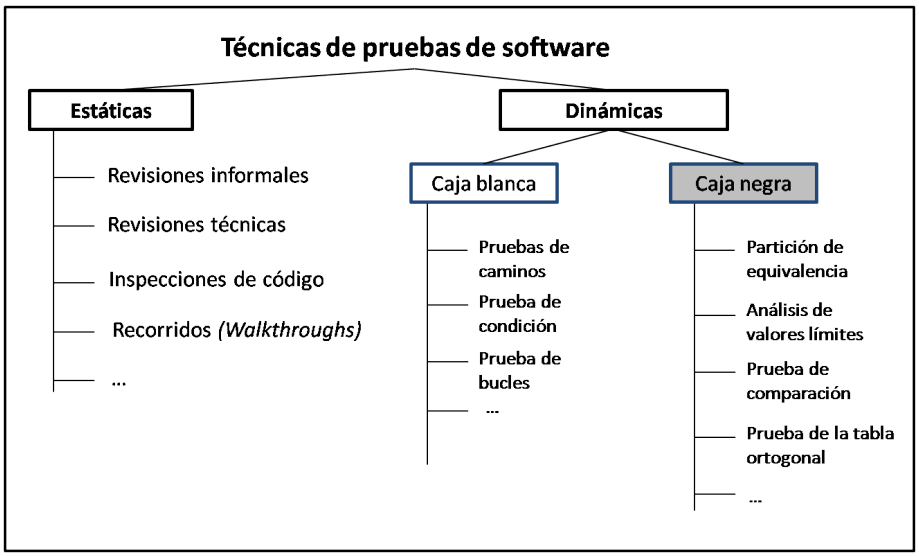

Figura 2.17: Pruebas de software estáticas y dinámicas

Prueba de resistencia

Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales. La prueba de resistencia ejecuta un sistema de manera que demande recursos en cantidad, frecuencia o volúmenes anormales. Una variante de la prueba de resistencia es la llamada prueba de sensibilidad. La prueba de sensibilidad trata de descubrir combinaciones de datos dentro de una entrada válida que pueda producir un proceso incorrecto [4].

Prueba de rendimiento

La prueba de rendimiento verifica el rendimiento del software en términos de tiempo de ejecución dentro del contexto de un sistema integrado [4].

2.8. Ejecución de las pruebas de software

Las técnicas de pruebas de software pueden ser clasificadas como estáticas o dinámicas, dependiendo de si requieren ejecución o no [27] (ver figura 2.17).

2.8.1. Estáticas

Estas actividades son consideradas como "procesos humanos", puesto que no ocupan de la ejecución del *SUT*. Se consideran estáticos porque envuelven la lectura o inspección visual de un programa (código, diagramas de flujo, casos de uso, etc.) por un grupo de personas. Entre estas actividades están: inspecciones de código, revisiones del programa, recorridos (en inglés, *walkthroughs*), etc.

2.8.2. Dinámicas

Estas actividades ocupan de la ejecución del programa. Las entradas son convertidas en salidas, las cuales son analizadas con respecto a los resultados esperados (obtenidos de un oráculo). Pueden ser de caja blanca o caja negra. Las actividades dinámicas más comunes son las pruebas unitarias, pruebas de integración, pruebas de sistema, etc.

2.9. Creación de las pruebas de software

Las pruebas pueden ser creadas manual o automáticamente. Ambas técnicas tienen sus ventajas: las pruebas manuales nos proveen de profundidad; las pruebas automáticas nos proveen de amplitud.

2.9.1. Pruebas manuales

Esta es una técnica de pruebas de software en la que el probador debe escribir, manualmente, los casos de prueba y el oráculo de prueba, o bien el código que los genere. Aunque las pruebas manuales tienen sus ventajas (el probador puede saber o tener una idea de los mejores casos de prueba que tengan mayor probabilidad de encontrar un *bug*), en general, son laboriosas, ocupan mucho tiempo y son propensas a que el probador se equivoque y haga malos casos de prueba.

2.9.2. Pruebas automáticas

El proceso de pruebas es una labor muy ardua y requiere de mucho tiempo. Beizer [31] dice que la mitad de la labor que se ocupa en el desarrollo de un programa es gastada

en las actividades de pruebas. Hailpern y Santhanam [33] nos dicen que las actividades de depuración, pruebas y verificación pueden ir del 50 % al 75 % del costo total de desarrollo. Es por eso que uno de los objetivos de las pruebas de software se centra en automatizar tanto como sea posible las actividades del proceso de pruebas.

En las pruebas automáticas, la generación (y ejecución) de los casos y oráculos de prueba se hacen automáticamente por medio de alguna herramienta de pruebas. Las herramientas de prueba pueden automatizar parte de las actividades del proceso de pruebas, i.e., generación y ejecución de casos de prueba, y evaluación de los resultados. En el capítulo 3 explicaremos con más detalle la automatización de estas actividades.

2.10. Defectos en el software

La introducción de defectos en el software es una práctica común. Éstos no aparecen por generación espontánea, sino que el desarrollador es el que comete un error al codificar. El defecto es la manifestación de este error. El defecto, comúnmente llamado *bug*, es un paso, instrucción o dato incorrecto en un programa, que puede llegar a provocar que el programa tenga un comportamiento no definido o arroje una salida no esperada. A esto se le conoce como fallo.

A continuación mencionamos algunos casos donde los errores en el software han causado grandes perdidas:

- **La explosión del Ariane 5 [16, 17]**: el 4 de junio de 1996, el cohete no tripulado Ariane 5 fue lanzado por la agencia espacial europea y, sólo 37 segundos después del despegue, explotó. El cohete estaba en su primer viaje después de una década de desarrollo que costó cerca de siete mil millones de dolares. Las pérdidas por el cohete destruido y su carga fueron valuadas en quinientos millones de dolares. La causa del fallo fue un error en el sistema de referencia o guía. Específicamente, un error de conversión de un número de punto flotante de 64 bits a un entero de 16 bits. El número de punto flotante era mayor que el valor que puede ser representado por un entero con signo de 16 bits.
- **Fallo en un misil Patriot [17]**: el 25 de febrero de 1991, durante la guerra del Golfo,

un error en el software del sistema de misiles MIM-104 Patriot ocasionó el fallo en la interceptación de un misil iraquí que se dirigía a una cuartel militar estadounidense en Dharan, Arabia Saudita. El resultado fue la muerte de 28 soldados. La causa fue un cálculo inexacto del tiempo debido a un error aritmético.

- **Falla en máquina de radioterapia Therac-25 [18]**: entre junio de 1985 y enero de 1987, seis accidentes conocidos, debido a sobredosis de rayos X administradas por la máquina de radioterapia llamada Therac-25, resultaron en lesiones graves e incluso muertes.

- **Falla en biblioteca OpenSSL [17, 19]**: no es necesario remontarnos a épocas lejanas para nombrar errores en el software que han causado gran impacto. El *bug* conocido como *Heartbleed*, una vulnerabilidad en el protocolo de seguridad de la biblioteca OpenSSL, fue dado a conocer en abril del 2014. Este *bug* permitía obtener información crucial de otros usuarios. Una de las cosas que provocó fue el cierre del acceso público al sitio web de la agencia de ingresos de Canadá tras el robo de números de seguridad social [34].

Tipos de defectos de software

En esta sección vamos a presentar los tipos de defectos más comunes en la programación. Hay distintas maneras de clasificar los *bugs*.

Primero, vamos a presentar una clasificación según su grado de severidad. La gravedad de un error puede determinarse con base en el daño causado al funcionamiento del sistema. Es importante mencionar que el grado de severidad de un error varía dependiendo del tipo software donde se presente. Por ejemplo, un defecto catastrófico para un sistema nuclear significa que el fallo puede resultar en alguna muerte o en daño ambiental; mientras que un defecto catastrófico para un sistema de base de datos significa que el fallo puede causar pérdida de datos valiosos.

Es por eso que no hay un estándar que defina una clasificación de este tipo, sino que en cada sistema se determina el grado de severidad de los defectos basándose en el contexto en el que se aplica. Un ejemplo de clasificación de defectos de acuerdo a su grado

severidad se muestra a continuación [35]:

- **Catastrófico**: defectos que pueden causar varios daños serios. Ejemplo: el sistema puede perder funcionamiento, problemas de seguridad.
- **Mayor**: defectos que pueden causar serias consecuencias como la pérdida de datos importantes.
- **Menor**: defectos que pueden causar consecuencias pequeñas o insignificantes. Ejemplo: mostrar los resultados en un formato diferente al esperado.
- **Sin efecto**: los defectos de este tipo pueden no ocasionar impedimento en la ejecución del sistema, pero pueden llevar a una interpretación diferente y, generalmente, evitable. Ejemplo: simples errores tipográficos en la documentación.

Otro tipo de clasificación que tiene que ver con la etapa en la que se muestran los fallos es la siguiente:

- ***Bugs* de sintaxis**: aquellos en los que existe un error en la sintaxis de una instrucción en código de C. Este tipo de errores se detectan en tiempo de compilación. Un error común de este tipo es olvidar poner ';' al final de una sentencia.
- ***Bugs* de enlace**: esto significa que el enlazador no ha podido encontrar una o más funciones que ha llamado. Se detectan en tiempo de compilación/enlace.
- ***Bugs* en tiempo de ejecución**: este tipo de defectos ocurren cuando el programa está en ejecución. Normalmente, este tipo de *bugs* son detectados por algún *Segmentation Error*, *Floating Point Overflow* o, en el peor de los casos, porque el programa parece no hacer nada. Otros *bugs* de este tipo se explicarán en el capitulo 6, ya que en estos *bugs* nos enfocamos en este trabajo.

Otro tipo de defectos, que también se presentan en tiempo de ejecución, pero que no producen que el programa termine su ejecución, son los llamados:

- **Defectos que causan resultados erróneos**: este tipo de errores son los más difíciles de corregir, ya que no hay indicio de donde pudo haber ocurrido el error. Es decir, el programa parece funcionar bien, pero el resultado que arroja no es el que debería de arrojar. Estos resultados también pueden dividirse en dos:

- **Basura**: es cuando el resultado es cualquier cosa que no tiene que ver con lo que debería ser.
- **Los resultados casi correctos**: es cuando los resultados son correctos para algunos datos, pero no para otros. Normalmente esto lo causa algún error en la lógica del código, e.g., alguna sentencia condicional puesta en un mal lugar.

Hay un peor escenario que nos lleva a resultados erróneos y es cuando se intenta resolver algo de una manera errónea. Entonces, no importa que tanto se juegue con el código, si uno no se da cuenta que la manera de resolverlo está mal, siempre se van a obtener resultados erróneos.

2.11. Resumen

En este capítulo vimos el propósito de las pruebas, la importancia de estas y por qué es un área que genera un gran interés, no sólo para el desarrollador del software, sino también para el usuario final. Las pruebas forman parte de un proceso conocido como verificación y validación (V & V) [23]. Las pruebas pueden ser usadas para validar que el software cumple con los requerimientos del cliente: pruebas de validación; y para revelar defectos en el software: pruebas de defectos. En nuestro trabajo, utilizamos las pruebas para revelar defectos. Por consiguiente, probar un software es tratar de hacer que falle [12, 27]. Una falla en un software es un comportamiento incorrecto del mismo debido a la activación de un defecto. El defecto es una manifestación de un error o equivocación cometida por el programador [28].

El enfoque de las pruebas puede ser de caja blanca o de caja negra. Las pruebas de caja negra no necesitan conocer la estructura interna del programa a probar, sino que los casos de prueba se obtienen de especificaciones [4]. Las pruebas de caja blanca si necesitan conocer la estructura interna del programa bajo prueba para poder obtener los casos de prueba [4]. Las pruebas se clasifican también por niveles, los cuales corresponden con las etapas del desarrollo de software. Los niveles de pruebas más comunes son: pruebas unitarias, pruebas de integración, pruebas de interfaces, pruebas de validación y pruebas del sistema [4, 23, 32].

Puesto que no es una ciencia exacta, no podemos asegurar que una prueba encuentra todos los fallos o que al no encontrar ninguno, el programa bajo prueba, no los tiene. Por este tipo de razones y por el trabajo tan arduo y complejo que representa todo el proceso de pruebas (desde la generación de casos de prueba, su ejecución, la evaluación y el reporte de los resultados) es por lo que varios programadores no le dan la importancia debida a este proceso de verificación del software. Es por eso que varias investigaciones recientes basan sus esfuerzos en la automatización de, al menos, una parte del proceso de pruebas.

Capítulo 3

Pruebas automáticas de software

3.1. Introducción

Ya mencionamos que el proceso de pruebas es una tarea laboriosa que ocupa de al menos el 50 % del costo total de desarrollo [33]. Por esta razón es que parte de la investigación en el área de pruebas se centra en automatizar, tanto como sea posible, las actividades de las mismas. En las siguientes secciones de este capítulo se mencionan los principales aspectos de las pruebas que son automatizados con el fin de reducir el tiempo y la intervención humana en el proceso de pruebas.

3.2. Automatización de las pruebas de software

Sommerville describe un banco de pruebas que podría ayudar en la automatización de las pruebas. Un banco de pruebas del software es un conjunto integrado de herramientas para soportar el proceso de pruebas [23]. Algunas de las herramientas que podría tener un banco de pruebas de este tipo son (ver figura 3.1):

- **Gestor de pruebas**: gestiona la ejecución de las pruebas del programa. El gestor de pruebas mantiene un registro de los datos probados, los resultados esperados y las unidades probadas. Los marcos de trabajo automatizados tales como *JUnit* [10] son ejemplos de gestores de pruebas.

- **Generador de datos de prueba**: genera datos de prueba para el programa a pro-

bar. Esto puede conseguirse seleccionando datos de una base de datos o utilizando patrones para generar datos aleatorios de forma correcta.

- **Oráculo**: genera predicciones de resultados esperados de pruebas.

- **Comparador de ficheros**: compara los resultados de las pruebas del programa con los resultados de pruebas previos e informa de las diferencias entre ellos. Cuando se utilizan pruebas automatizadas, los comparadores pueden ser llamados desde las mismas pruebas.

- **Generador de informes**: proporciona la definición de informes y facilidades de generación para los resultados de las pruebas.

- **Analizador dinámico**: añade código a un programa para contar el número de veces que se ha ejecutado cada sentencia. Después de las pruebas, se genera un perfil de ejecución que muestra cuántas veces se ha ejecutado cada sentencia del programa.

- **Simulador**: se pueden utilizar diferentes tipos de simuladores. Los simuladores de la máquina objetivo simulan la máquina sobre la que se ejecuta el programa. Los simuladores de interfaces de usuario son programas conducidos por *scripts* que simulan múltiples interacciones de usuarios simultáneas.

Meyer [12] menciona que algunos de los aspectos que debería proporcionar una herramienta que automatice el proceso de pruebas son los siguientes:

- **Ejecución de pruebas**: se refiere al hecho de que cuando se tienen miles de casos de prueba, se requiere de algún mecanismo que los ejecute automáticamente, ya que hacerlo manualmente sería algo tedioso, especialmente cuando se tienen que hacer cada vez que se libere una versión del software. Tradicionalmente, los probadores hacen uso de *scripts* que ejecutan las pruebas. Este aspecto es el que más automatizado está hoy en día.

- **Pruebas de regresión**: con estas pruebas se comprueba que la nueva versión del *SUT* tenga al menos las mismas características que la versión anterior, i.e., tiene que pasar todas las pruebas que su versión anterior ya pasaba.

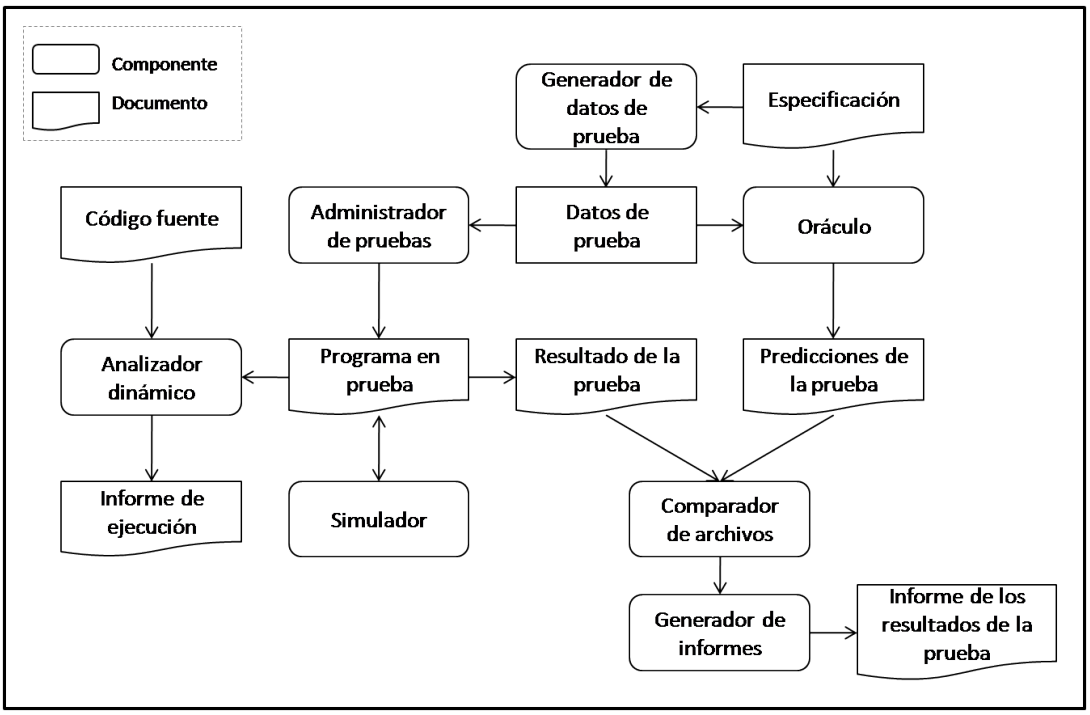

Figura 3.1: Banco de pruebas de software de Sommerville

- **Capacidad de recuperación**: se refiere a la capacidad de una herramienta de prueba que permite que, a pesar que algún caso de prueba haga que el *SUT* finalice de forma incorrecta, pueda recuperarse del fallo y continuar probando los demás casos de prueba.

- **Generación de casos de prueba**: es uno de los aspectos más interesantes y que pocas herramientas lo trabajan (una es *AutoTest* [12]). Abarca la generación de datos de prueba, la especificación o condiciones de ejecución y una salida esperada. En la sección 3.3 expondremos las metodologías existentes para la generación de datos de prueba.

- **Oráculos de prueba**: este aspecto representa otro desafío. Entra dentro de la automatización de la evaluación de los resultados de las pruebas. Una prueba que ha sido ejecutada sólo es útil si sabemos si pasó o falló; un oráculo es un mecanismo para determinar esto.

- **Minimización**: significa simplificar lo más posible un caso de prueba que ya produ-

ce un fallo, de manera que el fallo que produzca el caso de prueba minimizado sea evidencia del mismo defecto.

En este trabajo nos enfocamos en la gestión y ejecución de las pruebas, en la generación de datos de prueba (ver sección 3.3) y una parte de los oráculos de prueba (que veremos como evaluación de resultados en la sección 3.4).

3.3. Generación de datos de prueba

La mayor parte del software que se desarrolla actualmente tiene un nivel de complejidad tal que el número de valores de entrada e incluso el número de posibles escenarios, que permite, es extremadamente grande. Para cualquier caso de prueba, los datos de prueba son requeridos para poder probar el *SUT*. Una generación manual de los datos de prueba tiende a perder ciertos escenarios que podrían ser importantes; puesto que, usualmente, no se tienen los recursos humanos (más importante aún, el tiempo) necesarios para cubrir todos los casos posibles.

Las técnicas que permiten automatizar esta actividad, ayudan a descubrir estos casos que escapan a los creados manualmente por el probador. Además, el costo para generar los datos de prueba disminuye y se incrementa la calidad del *SUT*.

Las pruebas manuales nos dan profundidad; mientras que las automáticas nos dan amplitud [11].

La generación de datos de pruebas es el proceso en el que se identifica un conjunto de datos de prueba que cumple con un criterio de prueba seleccionado [36]. Esta generación es una actividad compleja que abarca varios pasos donde, cada paso, se ve envuelto en distintas dificultades.

Un sistema generador de datos de prueba típico, consiste de tres partes: un programa analizador, un selector de ruta y un generador de datos de prueba [8] (ver figura 3.2). El código fuente es pasado al programa analizador, el cual produce los datos necesarios para el selector de ruta y el generador de datos de prueba. El selector de ruta define el criterio de

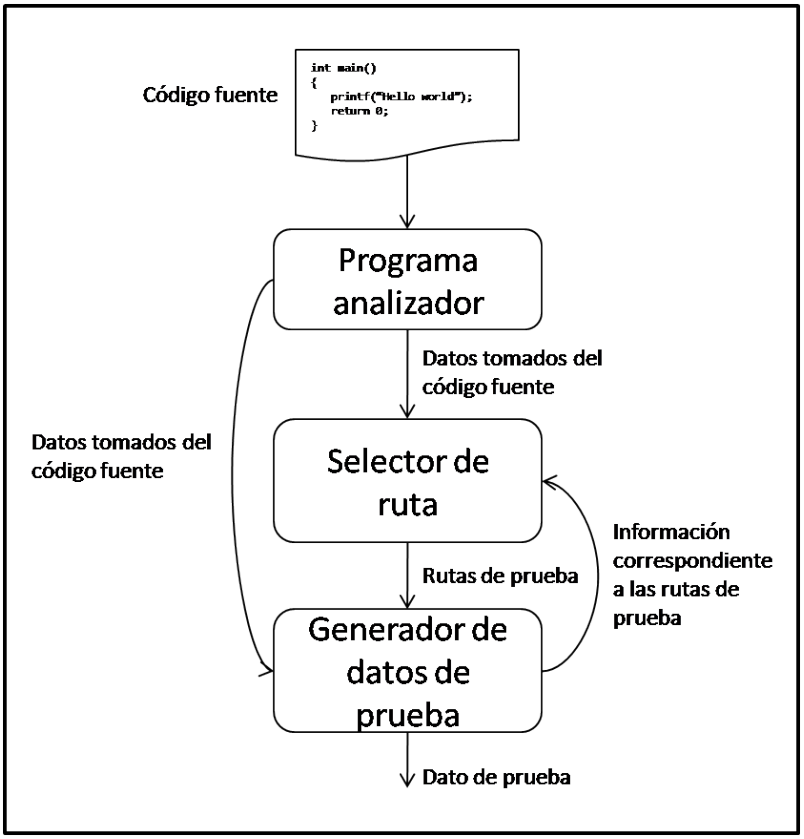

Figura 3.2: Arquitectura general de un sistema generador de datos de prueba

selección de datos de prueba. Este criterio puede ser alta cobertura de código, selección de caminos, etc. El generador produce datos de prueba de acuerdo al criterio de selección definido por el selector de ruta.

3.3.1. Programa analizador

El programa analizador provee toda la información necesaria, concerniente al programa a probar, como grafos de dependencia de datos, grafos de flujo de control, etc., con el fin de que el selector de ruta determine las rutas para las cuales el generador de datos de prueba producirá valores de entrada [8].

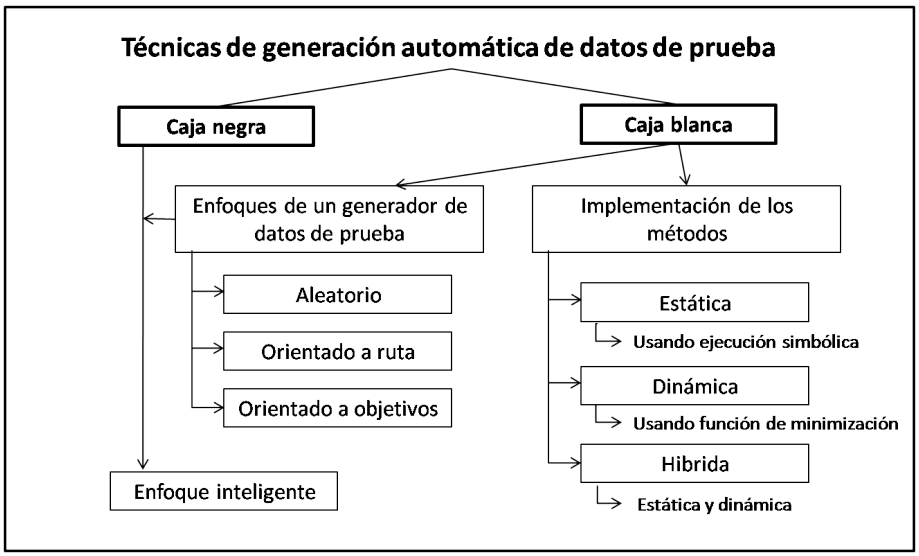

Figura 3.3: Técnicas de generación automática de datos de prueba

3.3.2. Generador de datos de prueba

Existen tres enfoques para desarrollar un generador de datos de prueba:

- Generar datos de prueba para una ruta específica.
- Generar datos de prueba para una ruta no específica.
- Generar datos de prueba aleatoriamente.

Estos enfoques caen en tres estrategias de generación de datos de prueba: orientada a ruta (*Path-oriented*), orientada a objetivos (*Goal-oriented*) y aleatoria (*Random*) [7, 8]. Cada una de estas estrategias puede implementarse estática o dinámicamente, ver figura 3.3.

Generación de datos de prueba estática y dinámica

Un generador puede usar ejecución simbólica o ejecución real, i.e., la generación puede ser estática o dinámica.

Ejecutar un programa simbólicamente significa que en lugar de usar valores reales se usan variables. Es decir, la idea es crear expresiones en términos de variables. Por ejemplo, hagamos de x y y dos variables de entrada. Además, tenemos las siguientes expresiones en

código:

$$t = x + y;$$

$$w = x - y;$$

$$z = t * w;$$

Entonces, si expresamos la operación que contiene z en términos de variables obtenemos:

$$z = x * x - y * y;$$

Por otro lado, la ejecución real se refiere a que, en lugar de usar variables, se usan valores. Valores que el programa selecciona como entradas con algún algoritmo, quizás con cierta aleatoriedad.

Estrategia de generación de datos de prueba orientada a ruta

Este enfoque de generación de datos de prueba no da la posibilidad al generador de seleccionar un conjunto de caminos, sino sólo uno [8]. En otras palabras, genera datos de prueba que ejerciten un sólo camino especifico. Esta estrategia alcanza una mejor predicción de cobertura, sin embargo, su precisión incrementa la dificultad de seleccionar datos de prueba.

Estrategia de generación de datos de prueba orientada a objetivos

Este enfoque genera datos de prueba para ejercitar cualquier camino, no uno especifico [8]. Es decir, puede seleccionar el camino, de todos los posibles, para el cual generará datos de prueba. Dos de los métodos que utilizan este enfoque son:

- Enfoque de encadenamiento (*chaining approach*)
- Enfoque orientado a aseveración (*assertion-oriented approach*)

Estrategia de generación de datos de prueba aleatoria

Es el enfoque más simple para la generación de datos de prueba. La ventaja es que puede ser usado para generar entradas de cualquier tipo de un programa. Sin embargo, ha

sido criticado puesto que no tiene un buen desempeño en términos de cobertura. Esto es debido a que tiene una baja probabilidad de encontrar fallos semánticamente pequeños, de manera que no alcanza una cobertura alta. Un fallo semánticamente pequeño es aquel fallo que solamente es revelado por un pequeño porcentaje de la entrada del programa. El mayor problema del enfoque aleatorio es la selección de datos de prueba adecuados [36].

Debido a que nuestro trabajo de tesis consiste en desarrollar una herramienta que realice pruebas aleatorias, usamos la estrategia aleatoria en una técnica llamada "pruebas aleatorias", la cual explicaremos en el capítulo 4.

3.3.3. Selector de ruta

La efectividad del sistema generador de datos de prueba depende mucho en como son seleccionadas las rutas.

Con respecto a la selección de las rutas, podemos definir al problema de la generación automática de datos de prueba como: dado un programa P encontrar conjunto de rutas mínimo en P tal que cumpla con un criterio de cobertura especificado [8], que básicamente es un criterio de selección de los datos. El objetivo de un criterio de cobertura es abarcar la máxima cantidad de fallos en un programa.

Seleccionando cuidadosamente las rutas podemos obtener un conjunto de datos de prueba que ejercite la mayor parte del programa. Mientras más fuerte sea el criterio de cobertura más rutas son seleccionadas. A continuación se listan algunos de los criterios de cobertura más conocidos [8]:

- **Cobertura de sentencia (*statement coverage*)**: ejecuta todas las sentencias del programa.
- **Cobertura de rama (*branch coverage*)**: ejecuta todas las ramas del programa, e.g., en una sentencia condicional `if`, ambas rutas (`true` y `false`) deben ser ejecutadas.
- **Cobertura de condición (*condition coverage*)**: cada clausula de cada condición del programa debe ser ejecutada con ambos valores (`true` y `false`). Esta cobertura

no es igual que la cobertura de rama. La cobertura de condición puede implicar una cobertura de rama, pero no necesariamente. Por ejemplo, supongamos que tenemos el siguiente fragmento de código:

```
if a and b then
```

La cobertura de condición se satisface con las siguientes dos pruebas:

1. `a = true`, `b = false`
2. `a = false`, `b = true`

Sin embargo, estas dos pruebas no satisfacen la cobertura de rama, ya que nunca cumplen la condición del `if` y, por lo tanto, no ejercitan la rama `then`.

- **Cobertura de múltiple condición (*multiple-condition coverage*)**: cada combinación de los valores de cada clausula de cada condición deben ser ejecutados. Por ejemplo, en la siguiente sentencia:

```
if (a or b) and c then
```

Se requerirían de ocho pruebas para satisfacer esta cobertura:

- `a=false`, `b=false`, `c=false`
- `a=false`, `b=false`, `c=true`
- `a=false`, `b=true`, `c=false`
- `a=false`, `b=true`, `c=true`
- `a=true`, `b=false`, `c=false`
- `a=true`, `b=false`, `c=true`
- `a=true`, `b=true`, `c=false`
- `a=true`, `b=true`, `c=true`

- **Cobertura de ruta (*path coverage*)**: se ejecuta cada ruta del programa.

Todos los criterios de cobertura mencionados son orientados a caja blanca, ya que es necesario realizar un análisis del código para poder lograr la cobertura deseada. Cuando se trata de pruebas de caja negra, el criterio de selección de los datos no es una cobertura de código como los anteriores, sino que el criterio es seleccionar un subconjunto de todos los casos de prueba posibles que tenga la más alta probabilidad de detectar la mayor cantidad de errores. Explicaremos más del criterio de selección de datos en pruebas de caja negra, más específicamente en pruebas aleatorias, en el capítulo 4.

3.4. Evaluación de resultados

La automatización de la evaluación o interpretación de los resultados obtenidos con las pruebas, es una de las actividades más laboriosas, y uno de los desafíos más grandes e importantes que se tiene en la investigación del área de pruebas de software. Son necesarios un conjunto de resultados esperados para cada caso de prueba con el fin de verificar los resultados de las pruebas. La generación de estos resultados esperados es a menudo realizada por un mecanismo conocido como "oráculo de prueba" [37].

Oráculos de prueba

Los oráculos de prueba establecen el comportamiento aceptable para las ejecuciones de prueba. Todas las técnicas de pruebas de software dependen de la disponibilidad de un oráculo, i.e., algún método con el cual se verifique si el *SUT* ha tenido, o no, un comportamiento adecuado en una ejecución particular [38].

El crear oráculos de prueba para programas simples puede ser sencillo, sin embargo, para software relativamente complejo, puede ser difícil saber si el programa devolvió un dato correcto o uno incorrecto. Existen diferentes enfoques para generar, capturar y comparar resultados de prueba, lo cual puede involucrar una actividad humana o ser automático. La verificación automática de resultados de prueba ha tenido una implicación importante tanto en el diseño de casos de prueba como en el diseño de oráculos.

Los oráculos varían enormemente en sus características. Hoffman [37] identificó cinco clases de oráculos que corresponden con enfoques de pruebas automáticas (la tabla 3.1

Clase de oráculo	Definición	Ejemplo de uso	Ventajas	Desventajas
Oráculo verdadero	Generación independiente de resultados esperados	Validación de algoritmos	Posibilidad de hacer pruebas exhaustivas	Implementación costosa y probables tiempos de ejecución largos
Oráculo estocástico	Verifica un ejemplo seleccionado aleatoriamente	Verificación operacional	Automatiza pruebas con un oráculo sencillo	Podría perder errores sistemáticos y específicos y puede consumir tiempo para verificar
Oráculo heurístico	Verifica puntos seleccionados y usa una heurística para los restantes	Verificación de algoritmos	Más fácil que un verdadero oráculo	Puede perder errores sistemáticos y algoritmos incorrectos
Oráculo de muestreo	Verifica un ejemplo especialmente seleccionado	Pruebas de valores límites	Posible verificación muy rápida con un oráculo sencillo	Podría perder errores sistemáticos o específicos
Oráculo consistente	Compara los resultados de la ejecución n con la ejecución $n-1$	Pruebas de regresión	El más rápido: puede generar y verificar grandes cantidades de datos	La ejecución original podría incluir errores desconocidos

Tabla 3.1: Clases de oráculos

resume las características, ventajas y desventajas de estas cinco clases de oráculos):

- **Oráculo verdadero**: reproduce fielmente todos los resultados relevantes para un software bajo prueba usando recursos independientes que van desde la plataforma, algoritmos, procesos, compiladores, código, etc.
- **Oráculo estocástico**: se enfoca en la verificación de un conjunto de valores de ejemplo seleccionados estadísticamente.
- **Oráculo heurístico**: reproduce los resultados seleccionados para el *SUT* y los valores restantes se puede comprobar usando algoritmos simples o verificaciones de consistencia basadas en una heurística.
- **Oráculo de muestreo**: utiliza un conjunto seleccionado de valores. Los valores son seleccionados por medio de algún criterio más que por alguna aleatoriedad estadística. Por ejemplo, valores límite, enteros específicos, puntos medios, valores mínimos y máximos, etc.
- **Oráculo consistente**: usa los resultados obtenidos de una prueba ya ejecutada como oráculo para posteriores pruebas. Esto es muy útil para evaluar efectos de cambios de una versión a otra.

En la sección 5.2.4 expondremos una metodología que usa oráculos para la implementación y posterior verificación de un programa. Esta metodología es llamada diseño por contrato [11, 12].

3.5. Resumen

Las actividades de pruebas y verificación pueden ir del 50 % al 75 % del costo total de desarrollo de software [31, 33]. Una herramienta que soporte pruebas automáticas debería tener, entre otros, los siguientes componentes: un gestor de pruebas, gestiona la ejecución de las pruebas; un generador de datos de pruebas, genera datos de prueba automáticamente; un oráculo de prueba, genera predicciones de resultados esperados; un comparador de resultados, compara los resultados obtenidos con los esperados; y un generador de informes [12, 23].

Nuestro trabajo se enfoco en la gestión y ejecución de las pruebas, en la generación de datos de prueba y en la evaluación de los resultados de prueba. Un sistema generador de datos de prueba típico, consiste de tres partes: un programa analizador, un selector de ruta y un generador de datos de prueba [8].

La evaluación automática de los resultados obtenidos con las pruebas requiere de un conjunto de resultados esperados para cada caso de prueba con el fin de verificar los resultados de las mismas. La generación de estos resultados esperados es a menudo realizada por un mecanismo conocido como “oráculo de prueba” [37]. Estos oráculos establecen el comportamiento aceptable para las ejecuciones de prueba. Todas las técnicas de pruebas de software dependen de algún método con el cual se verifique si el programa bajo prueba ha tenido, o no, un comportamiento adecuado en una ejecución particular [38]. Una metodología que usa oráculos para la implementación y posterior verificación de un programa es el diseño por contrato [11, 12].

Capítulo 4

Pruebas aleatorias de software

4.1. Introducción

Como ya se había mencionado en el capítulo anterior, hay distintos tipos de técnicas para la generación de datos de prueba. En este trabajo nos enfocamos en las pruebas aleatorias, la cual es la técnica más sencilla y más comúnmente usada para la generación automática de los datos de prueba. En las siguientes secciones de este capítulo nos adentraremos con más detalle en esta técnica y también presentaremos algunas otras técnicas que han surgido como una mejora para menguar las desventajas que presentan las pruebas aleatorias.

4.2. Definición de pruebas aleatorias de software

Como se muestra en la figura 4.1, las pruebas aleatorias son una técnica dinámica y de caja negra en la cual el software es probado con datos, generados aleatoriamente, del dominio de entrada especificado. El dominio de entrada es el conjunto de todas las posibles entradas al software bajo prueba [39]. La característica fundamental de un técnica de generación de pruebas aleatorias es que genera entradas de prueba al azar, a partir de una gramática que describe el dominio de entrada.

De acuerdo a Richard Hamlet [39], para realizar las pruebas aleatorias se siguen los siguientes pasos:

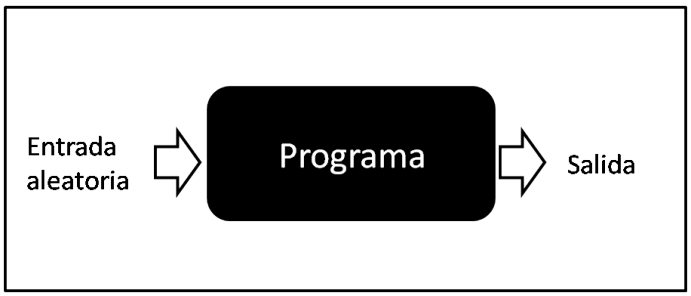

Figura 4.1: Pruebas aleatorias

1. Se define el dominio de entrada.
2. Se seleccionan entradas de prueba de forma independiente a partir del dominio definido.
3. El programa bajo prueba es ejecutado con las entradas seleccionadas.
4. Los resultados obtenidos son comparados con la especificación del programa.
5. La prueba falla si cualquier entrada lleva a un resultado incorrecto, de otra manera, la prueba pasa.

La generación de datos de prueba aleatoriamente, es un proceso sencillo y rápido que no requiere de muchos recursos de computo. Estas razones, además de la ausencia de intervención humana en la generación de los datos, la convierten en una técnica contra la cual se comparan otros métodos de generación de datos más complicados (como los vistos en la sección 3.3).

La generación de estos datos sin ningún uso de información de apoyo la hace altamente susceptible a la crítica. Sin embargo, los experimentos realizados en [39] y [40] confirman que la técnica de pruebas aleatorias es tan efectiva como cualquier otra técnica de pruebas sistemática.

4.2.1. Fortalezas de las pruebas aleatorias

La técnica de pruebas aleatorias es atractiva por dos razones:

- **Revela errores**: estudios han demostrado que esta técnica es efectiva en crear datos de prueba que revelan errores que otras técnicas no encuentran [41].

- **Es rentable**: dada la facilidad de implementar un algoritmo que genere datos de prueba aleatoriamente, el poco costo computacional, que requiere nada o poco esfuerzo de parte del probador, además de su capacidad de revelar errores, hacen a esta técnica muy interesante y llamativa desde el punto de vista del costo-beneficio [41].

4.2.2. Debilidades de las pruebas aleatorias

Esta técnica también cuenta con ciertas desventajas que limitan su efectividad [41]:

- **Puede generar datos prueba ilegales**: a pesar de que se puede definir el dominio del cual seleccionar los datos de prueba, incluso la gramática de la cual se generen los datos, puede producir datos ilegales, i.e., datos que violan los requerimientos del programa bajo prueba. Por ejemplo, alguna secuencia para generar caracteres aleatoriamente puede producir caracteres para alguna herramienta Unix que espera las entradas en un formato especifico; en ese caso el generador creará varias entradas ilegales.

- **Puede generar datos de prueba equivalentes**: puede crear datos que ejerciten el mismo comportamiento del programa, lo que genera una pérdida de tiempo.

- **Puede crear entradas largas**: usualmente, las herramientas que utilizan esta técnica, producen entradas largas (e.g., cadenas largas de caracteres o archivos grandes). A pesar de que estudios han sugerido que, en esta técnica, este tipo de entradas son más efectivas para encontrar errores [42], desafortunadamente, también pueden aumentar la complejidad de corregirlos. Principalmente, porque un caso de prueba grande y complejo puede ofuscar la posible causa del fallo, lo cual incrementa la cantidad de tiempo en depurar.

4.3. Variaciones de las pruebas aleatorias

Una prueba "completa" es imposible (i.e., una prueba no puede garantizar la ausencia de errores), y , en consecuencia, una prueba de cualquier programa es, por definición, incompleta. Myers [27] nos sugiere intentar reducir esta carencia tanto como sea posible. Dadas las restricciones de tiempo, costo, tiempo de computo, grupo de trabajo, etc., la cuestión clave se convierte en: "¿Qué subconjunto, de todos los casos de prueba posibles, tiene la más alta probabilidad de detectar la mayor cantidad de errores?"

Myers [27] nos dice que la metodología más pobre para generar este subconjunto óptimo es la de pruebas aleatorias. La probabilidad de que genere un subconjunto óptimo, o uno cercano al óptimo, es muy baja. Por esta razón es que han surgido distintas metodologías, basadas en pruebas aleatorias, pero dirigidas a encontrar un subconjunto cercano al óptimo. En las siguientes secciones de este capítulo se muestran algunas de estas metodologías.

4.3.1. Pruebas aleatorias plus

Esta variación de la estrategia de pruebas aleatorias es llamada "aleatoria plus", en inglés *Random plus (R+)* [9, 11]. Más que una variación es una extensión de la estrategia aleatoria. Usa valores especiales predefinidos, los cuales pueden ser valores límites (ver sección 2.6.1) o valores, que el probador considere, tengan alta probabilidad de encontrar fallos en el *SUT*. Estos valores pueden ser complementados con otros valores especiales que el probador considere efectivos para encontrar fallos en el *SUT*. Esta lista de valores debe ser actualizada manualmente antes de ejecutarse la prueba (si es necesario) y tiene una prioridad más alta que los valores seleccionados aleatoriamente debido a su relevancia y a su alta probabilidad de encontrar fallos.

4.3.2. Pruebas aleatorias adaptativas

Las pruebas aleatorias adaptativas [43], en inglés *Adaptive Random Testing* (*ART*), surgen como una mejora de la técnica de pruebas aleatorias. Para poder mejorarla, Chen et al. [43], tomaron en consideración un descubrimiento acerca de las entradas que causan fallas en los programas.

Este descubrimiento se dio en un trabajo de 1996. En este trabajo, Chan et al. [44] observaron que algunas de las entradas que causaban fallos formaban ciertos patrones en forma de figuras geométricas. Clasificaron a estos patrones de entradas que causan fallos (de aquí en adelante referidos como patrones o regiones de fallas) en tres categorías: de punto, de bloque y de banda.

La figura 4.2 muestra los patrones mencionados en un dominio de entrada de dos dimensiones, es decir, de dos variables de entrada: $x1$ y $x2$. Cada variable con un dominio limitado por: $a <= x1 <= b$ y $c <= x2 <= d$. El espacio en blanco representa a los valores que no provocan fallas, mientras que las formas en color negro representan los patrones de fallas.

Estos patrones de fallas se describen a continuación:

- Patrón de punto: en este tipo de patrón, las entradas que causan fallas se dispersan por todo el dominio en forma de puntos aislados. Un ejemplo de este patrón es la división por cero en una sentencia: *total = num1/num2*; donde las variables *total*, *num1* y *num2* son de tipo entero.

- Patrón de banda: este patrón es parecido al de bloque, pero las entradas, en lugar de formar bloques, forman líneas o bandas en el dominio de entrada.

- Patrón de bloque: en este patrón, varias entradas, que causan fallas, caen en lugares cercanos, lo que da lugar a bloques en el dominio de entrada.

De acuerdo a sus experimentos [44], descubrieron que para aquellos patrones de fallos diferentes al de punto, una dispersión uniforme de casos de prueba tiene más probabilidades de detectar fallos usando menos casos de prueba que las pruebas aleatorias ordinarias. Las pruebas aleatorias adaptativas buscan distribuir los casos de prueba de manera más uniforme dentro del espacio de entrada, con lo cual, logran mejorar la efectividad de las pruebas aleatorias para aquellos patrones de fallas diferentes al de punto.

De manera general, la forma en que trabaja *ART* es como sigue [43]:

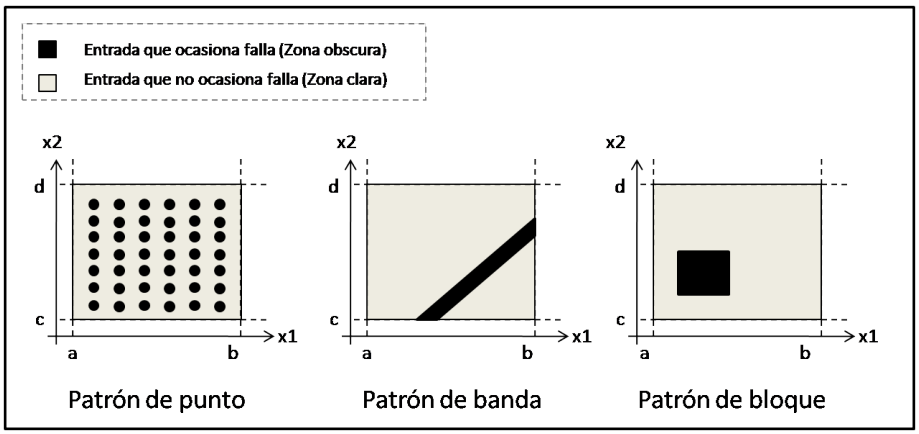

Figura 4.2: Patrones de entradas que causan fallas

- Usa dos conjuntos de casos de prueba: conjunto ejecutado (*executed set*) y conjunto candidato (*candidate set*), los cuales son disjuntos.
- El conjunto ejecutado es el conjunto que contiene casos de prueba distintos que fueron ejecutados y no revelaron ninguna falla. El conjunto candidato es el conjunto de casos de prueba distintos que fueron seleccionados aleatoriamente.
- El conjunto ejecutado está inicialmente vacío y el primer caso de prueba es seleccionado aleatoriamente del dominio de entrada.
- El conjunto ejecutado es actualizado incrementalmente seleccionando un elemento del conjunto candidato hasta que una falla sea revelada. El elemento del conjunto candidato seleccionado debe ser un elemento que esté "más alejado" de todos los elementos del conjunto ejecutado (i.e., el más alejado de todos los casos de prueba ejecutados) y representa el siguiente caso de prueba.

Esta técnica como varias otras que utilizan el concepto de "más alejado" o distancia entre casos de prueba, utilizan un enfoque propio para implementarlo.

4.3.3. Pruebas aleatorias adaptativas de espejo

Las pruebas aleatorias adaptativas de espejo [45], en inglés llamada *Mirror Adaptive Random Testing* (*MART*), es una mejora a técnica de pruebas aleatorias adaptativas. Su mejora la hacen por medio de una técnica de particionamiento de espejo que decrementa

las operaciones realizadas por *ART* y reduce el *overhead*.

Con esta técnica, *ART* no se aplica a todo el dominio de entrada. En lugar de eso, sólo es aplicada para generar ciertos patrones de distribución de casos de prueba en ciertas partes del dominio de entrada. Después, el algoritmo de espejo, duplica estos patrones de distribución de casos de prueba en otras partes del dominio.

La manera en que trabaja *MART* es la siguiente:

1. El dominio de entrada de un programa a probar es dividido en m subdominios disjuntos de igual tamaño y forma.

2. Un subdominio es designado como el "subdominio original". *ART* es aplicado solamente a este subdominio.

3. Un caso de prueba es seleccionado del "subdominio original". Este caso de prueba es ejecutado y, si no detecta un fallo, es "reflejado" en otro subdominio. Si en el siguiente subdominio tampoco detecta un fallo, es reflejado en otro subdominio, y así sucesivamente. A estos subdominios se les llama "subdominios espejo".

4. Este reflejo de casos de prueba se logra mediante "funciones espejo" (mostradas en la figura 4.3), las cuales generan $(m - 1)$ casos de prueba distintos, uno en cada "subdominio espejo". Si ninguno de estos casos de prueba detecta algún fallo, se vuelve a repetir el proceso de *ART* en el "subdominio original".

Los tipos de mapeo de casos de prueba (llamadas "funciones espejo") que utiliza esta técnica se muestran en la figura 4.3. La forma de dividir el dominio de entrada de *MART* es llamada "particionamiento espejo". Algunas formas de "particionamiento espejo" de un dominio de dos dimensiones se muestran en la figura 4.4.

4.3.4. Pruebas aleatorias restringidas

Las pruebas aleatorias restringidas [46, 47], en inglés *Restricted Random Testing* (*RRT*), es otra técnica creada con el fin de disminuir el *overhead* de *ART*. Para lograr esto, esta técnica, restringe el área del dominio de entrada; de la cual los siguientes casos de prueba

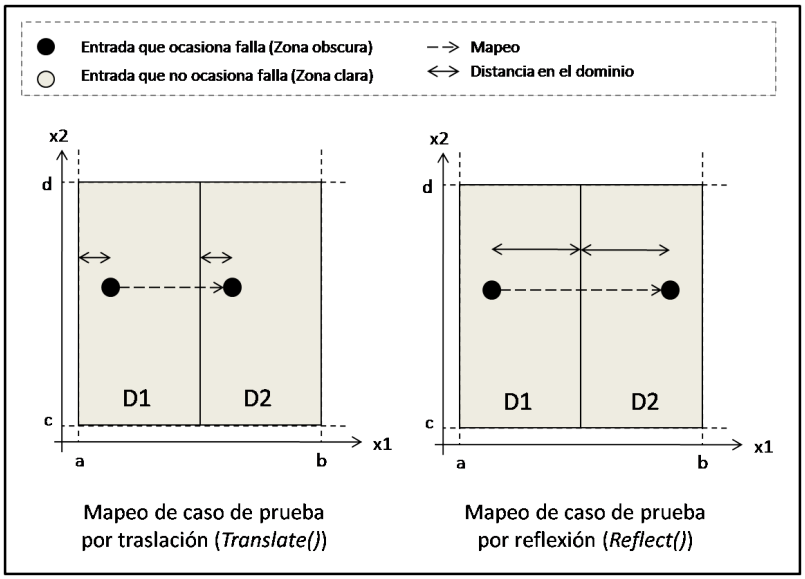

Figura 4.3: Funciones espejo para mapeo de casos de prueba en pruebas aleatorias adaptativas de espejo

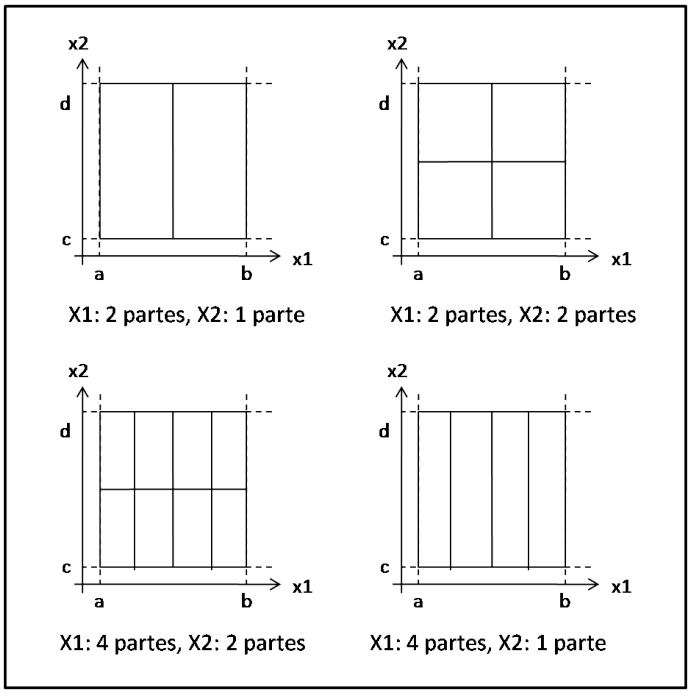

Figura 4.4: Formas de "particionamiento espejo"

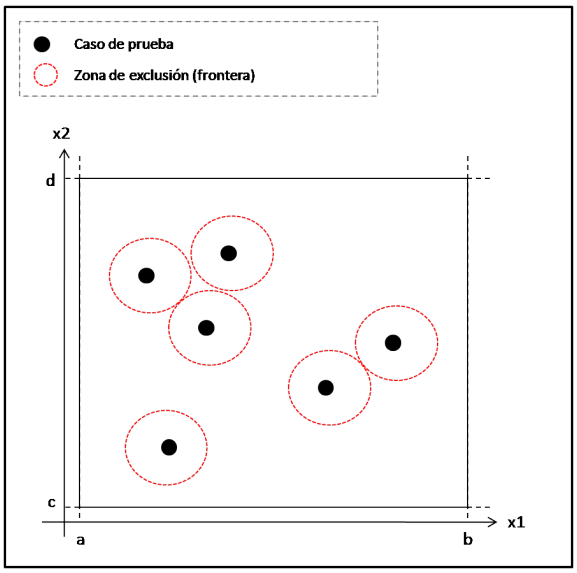

Figura 4.5: Dominio de entradas con zonas de exclusión alrededor de casos de prueba

pueden ser seleccionados. Esto lo hacen para asegurar una distancia mínima entre todos los datos de prueba. Para esto, una área circular es definida alrededor de entradas que no causan fallos. Los siguientes datos de prueba son seleccionados sólo si se encuentran fuera de la zona restringida por esas áreas (ver figura 4.5). El tamaño de la zona de exclusión es relativo al tamaño del dominio de entrada. De manera general, los pasos que realiza esta técnica para seleccionar los casos de prueba son:

1. Se selecciona aleatoriamente, un dato de prueba del dominio de entrada.

2. Se verifica si el dato de prueba cae en alguna zona de exclusión. Si está dentro de una de estas zonas, se regresa al paso 1. El paso 1 y 2 se repiten hasta que un candidato seleccionado se encuentre fuera de las zonas de exclusión.

3. El dato de prueba, que ahora forma parte de un caso de prueba, se ejecuta en el programa bajo prueba. Si la ejecución no revela un fallo, un área circular es definida alrededor de este dato de prueba y esta área es ahora una zona de exclusión.

4. Vuelve al paso 1 para seleccionar el siguiente dato de prueba.

4.3.5. Pruebas aleatorias dirigidas

En inglés llamada *Directed Automated Random Testing* (*DART*), es una técnica desarrollada por Godefroid et al. [6] e implementada en una herramienta que lleva su mismo nombre. Las características principales de *DART* pueden dividirse en tres partes:

1. **Extracción automática de interfaz**: identifica automáticamente interfaces de un software dado. Estas interfaces contienen la declaración de las funciones, variables externas y la función *main* del programa.

2. **Manejador automático de pruebas**: genera manejadores de prueba que ejecutan los casos de prueba. Estos casos de prueba son generados aleatoriamente.

3. **Análisis dinámico de ejecución**: los resultados obtenidos, durante las pruebas, son analizados en tiempo de ejecución. De esta forma, la generación de los siguientes casos de prueba es dirigida hacia un nuevo camino con el fin de obtener una máxima cobertura de código.

El algoritmo de *DART* es implementado en una herramienta la cual mostraremos en la sección 5.3, ya que es uno de los trabajos, relacionados a herramientas que realizan pruebas automáticas, más citados en el estado del arte.

4.3.6. Estrategia aleatoria de barrido de puntos

Esta estrategia, en inglés llamada *Dirt Spot Sweeping Random Strategy* (*DSSRS*) [9], es una combinación de la estrategia de pruebas aleatorias ordinaria y la estrategia de *random plus*, agregándole algo que llaman "barrido de puntos sucios" (patrones de fallas) en el código del programa bajo prueba.

En este trabajo sólo se enfocaron en los patrones de falla de banda y bloque (mencionados en la sección 4.3.2). De manera general, lo que hace la estrategia es lo siguiente:

1. Empiezan con las estrategias de pruebas aleatoria y aleatoria plus (ver sección 4.3.1), para buscar el primer fallo.

2. Una vez encontrado el primer fallo, agrega valores vecinos (vecinos al valor del dato de prueba que encontró el fallo) a una lista de valores de interés.

3. Entonces, si el fallo se encuentra en un patrón de bloque o de banda, estos valores vecinos exploraran todo el patrón repitiendo los pasos 2 y 3 hasta que todos los fallos del bloque sean identificados.

De acuerdo a sus experimentos, la estrategia es altamente efectiva en los casos que el programa a probar contenga patrones de fallos de banda y/o de bloque.

Su principal desventaja es que tiene que esperar a que se encuentre el primer fallo, además de que es poco efectiva (o quizá nada efectiva) para los programas que tienen como patrón de fallos el de punto.

4.3.7. Pruebas quasi-aleatorias

Llamada en inglés *Quasi Random Testing* (*QRT*) [48], es una técnica que usa la contigüidad de las regiones de fallos para distribuir los casos de prueba. Usa una formula para hacer que los casos de prueba estén menos agrupados y más distribuidos en el dominio de entrada. Para lograrlo, hacen uso de "secuencias quasi-aleatorias", también conocidas como "secuencias de baja dispersión o baja discrepancia".

Una "secuencia quasi-aleatoria" es una secuencia de puntos en un cubo semiabierto de n dimensiones, con la propiedad de que, en cualquier subintervalo, los puntos estarán igualmente distribuidos.

Para explicar cómo es que son utilizadas estas secuencias, recordemos que un dominio de entrada de un programa puede ser representado como un cubo de n dimensiones (donde cada entrada agrega una dimensión), y que podemos considerar a cada punto dentro de este cubo como entradas. También recordemos que las entradas que relevan fallas suelen estar juntas unas de otras, de manera que, representadas en el cubo de n dimensiones, aparecen formando grupos en ciertas regiones (ver figura 4.2). Considerando lo anterior, podemos generar una secuencia quasi-aleatoria para un cubo de n dimensiones (cubo que representa al dominio de entrada del programa bajo prueba) y sólo tomar como entradas de prueba a los puntos de esa secuencia. Como los puntos de la secuencia quasi-aleatoria están distribuidos de manera uniforme por todo el hipercubo, de manera análoga, las entradas que corresponden con esos puntos están distribuidas de manera uniforme por todo

el dominio de entrada.

4.3.8. Pruebas aleatorias dirigidas por retroalimentación

Las pruebas aleatorias dirigidas por retroalimentación [49] (en inglés *Feedback-Directed Random Testing*, *FDRT* por sus siglas), es una técnica que genera, aleatoriamente, *suites* de pruebas unitarias para programas bajo el paradigma orientado a objetos. Usa la retroalimentación recibida de la ejecución de un conjunto previo de pruebas unitarias para poder generar el siguiente conjunto de una manera más dirigida. Las pruebas unitarias redundantes e ilegales, que se pudieran generar, son eliminadas incrementalmente con la ayuda de filtros y contratos. Esta técnica se implemento en la herramienta *RANDOOP*, explicaremos más de ambas en la sección 5.5.

4.3.9. Pruebas aleatorias adaptativas para software orientado a objetos

Es una técnica creada por Ciupa et al. [50]. Es una extensión de las pruebas aleatorias adaptativas, desarrollada para software orientado a objetos.

Desarrollaron una noción de distancia entre objetos y una estrategia (a la que llamaron *Adaptive Random Testing for Object-Oriented software*, *ARTOO* por sus siglas), la cual selecciona, como entradas, objetos que tengan la distancia más grande con otros objetos que ya han sido ocupados como entradas de prueba.

Definen a la "distancia entre objetos", como una medida de qué tan diferentes son dos objetos. Para medir esta distancia se toman en consideración tres propiedades:

- Distancia elemental: una medida de la diferencia que hay entre los valores directos de los objetos.
- Distancia de tipo: una medida de la diferencia que hay entre los tipos de los objetos. Esta diferencia es independiente de los valores mismos del objeto.
- Distancia de campos: una medida de la diferencia que hay entre los campos de los objetos. Usan el término campo para referirse, indiscriminadamente, a los campos

de un objeto y a los atributos de una clase. Los campos deben ser comparados uno a uno, sólo empatando aquellos campos que correspondan a los mismos atributos en ambos objetos; los campos que no empaten también tienen una diferencia, pero es calculada por la "distancia de tipo". Esta medida es la misma "distancia de objetos", pero aplicada recursivamente.

4.3.10. Pruebas aleatorias adaptativas a través de particionamiento dinámico

Chen et al. [51], definen dos algoritmos basados en pruebas aleatorias adaptativas, combinando "pruebas aleatorias adaptativas" con "pruebas de partición". Los algoritmos son los siguientes:

- **Pruebas aleatorias adaptativas por particionamiento aleatorio**: divide el dominio de entrada en subdominios y selecciona casos de prueba, aleatoriamente, de los subdominios más grandes.
- **Pruebas aleatorias adaptativas por bisección**: divide el dominio de entrada en particiones de igual tamaño. Un caso de prueba es seleccionado de cada partición. Las particiones son divididas en dos partes y se seleccionan casos de prueba de aquellos subdominios donde no se hayan elegido casos de prueba. De esta manera, garantizan que los casos de prueba estén distribuidos ya que todos se encuentran en diferentes particiones.

Las figuras 4.6 y 4.7 muestran el funcionamiento de los algoritmos "pruebas aleatorias adaptativas por particionamiento aleatorio" y "pruebas aleatorias adaptativas por bisección" respectivamente.

4.4. Resumen

Las pruebas aleatorias de software son una técnica de pruebas dinámica y de caja negra, en la cual el software es probado con datos de prueba impredecibles (aleatorios) elegidos de un dominio de entrada especificado [39]. El dominio de entrada es un conjunto de todas las posibles entradas al software bajo prueba. Esta técnica es efectiva en crear

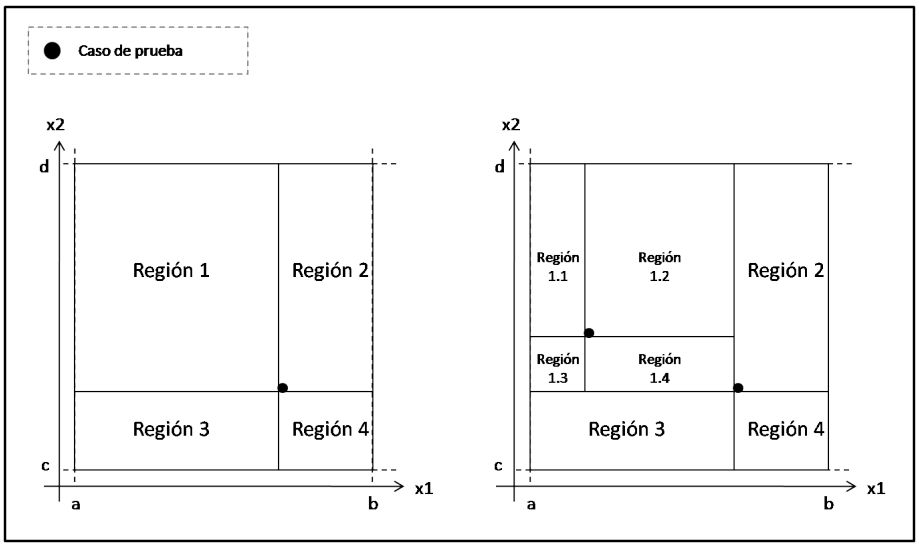

Figura 4.6: Funcionamiento del algoritmo “pruebas aleatorias adaptativas por particionamiento aleatorio”

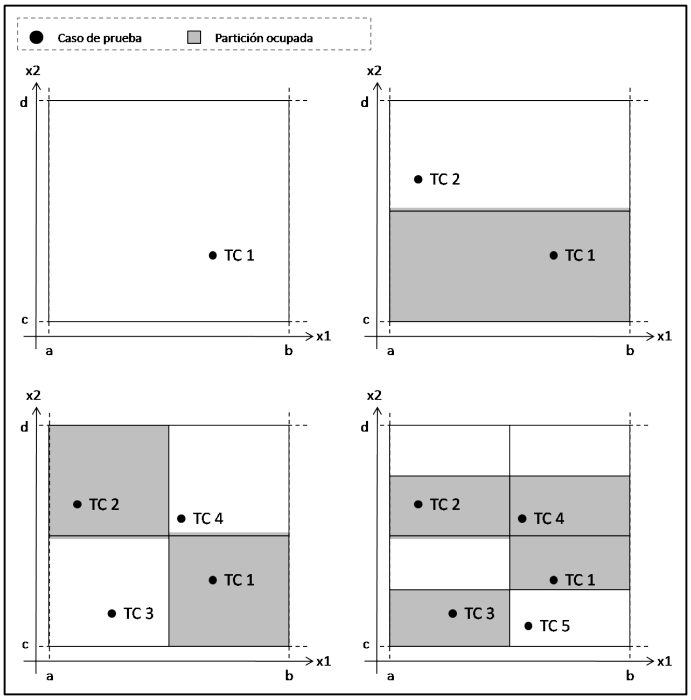

Figura 4.7: Funcionamiento del algoritmo “pruebas aleatorias adaptativas por bisección”

datos de prueba que revelan errores que otras técnicas no encuentran [41] y es fácil de implementar. Una de sus principales desventajas es que puede crear datos que ejerciten el mismo comportamiento del programa, lo que genera una pérdida de tiempo.

Se expusieron distintas variaciones de la técnica de pruebas aleatorias que pretenden aumentar la probabilidad de encontrar un fallo, pero que se siguen basando en el aspecto aleatorio. Dentro de estás técnicas se encuentran las pruebas aleatorias plus, las pruebas aleatorias adaptativas, las pruebas aleatorias adaptativas de espejo, las pruebas aleatorias restringidas, las pruebas aleatorias dirigidas, etc.

En el capítulo 5 se exponen algunas herramientas que soportan pruebas automáticas y que usan la metodología aleatoria para la generación de las pruebas. Algunas de estas herramientas usan alguna variación de las pruebas aleatorias descritas en este capítulo.

Capítulo 5

Herramientas de pruebas de software

5.1. Introducción

Como hemos visto en capítulos anteriores, hay distintos trabajos (investigaciones, metodologías, herramientas) que se han desarrollado como apoyo para automatizar el proceso de pruebas. Dentro de estos trabajos, han surgido herramientas que realizan pruebas automáticas. En las siguientes secciones se mencionarán algunas de estas, dando un mayor énfasis en *AutoTest* (ver sección 5.2), ya que nuestro trabajo se baso inicialmente en esta herramienta.

5.2. AutoTest

AutoTest [11–15] es una herramienta que permite automatizar ciertas actividades del proceso de pruebas. Fue creada por Bertrand Meyer y permite probar programas que cuentan con ciertos elementos que permiten su propia verificación, llamados "contratos" (ver sección 5.2.4), por lo cual su estrategia de pruebas es llamada pruebas dirigidas por contratos. *AutoTest* permite las pruebas manuales y automáticas en orden de combinar los beneficios de ambos enfoques.

5.2.1. Características

Los tres principales aspectos de *AutoTest*, los cuales son [12]:

- **Generación de pruebas:** crea y ejecuta casos de prueba automáticamente.
- **Extracción de pruebas:** produce casos de prueba de ejecuciones fallidas.
- **Integración de pruebas manuales:** permite que el probador escriba sus propias pruebas de forma manual.

Además de que *AutoTest* ayuda a provocar los fallos, también administra la información acerca de cada uno de los fallos ocurridos durante las pruebas.

5.2.2. Arquitectura

Las partes principales de la arquitectura de *AutoTest* (figura 5.1) se describen a continuación [11]:

- **Estrategia de pruebas**: Componente conectable (*pluggable*) que determina que instrucciones deben ser ejecutadas en el sistema bajo prueba. Una estrategia de pruebas recibe el ámbito de prueba, i.e., el conjunto de clases que se van a probar. Usa esta información para sintetizar los casos de prueba que le da al *proxy*.
- **Interprete**: Ejecuta las instrucciones en el sistema bajo prueba. Se ejecuta como un proceso aparte para incrementar la robustez. Las instrucciones típicas son: crear objeto (los objetos son creados aleatoriamente por medio de la técnica aleatoria plus, ver sección 4.3.1), invocar rutina, asignar resultado.
- ***Proxy***: Componente que maneja la comunicación entre procesos. Recibe la petición de ejecución desde la estrategia y la redirige al intérprete. Los resultados de la ejecución son enviados al oráculo.
- **Oráculo**: Esta basado en la idea de pruebas dirigidas por contratos, ver sección 5.2.4. Recibe los resultados de la ejecución y determina el éxito de la ejecución. El oráculo despliega los resultados de la prueba en documentos XML, en una representación gráfica en la *GUI* de *AutoTest* y en archivos usando un formato de *Gobo Eiffel Test* [15].

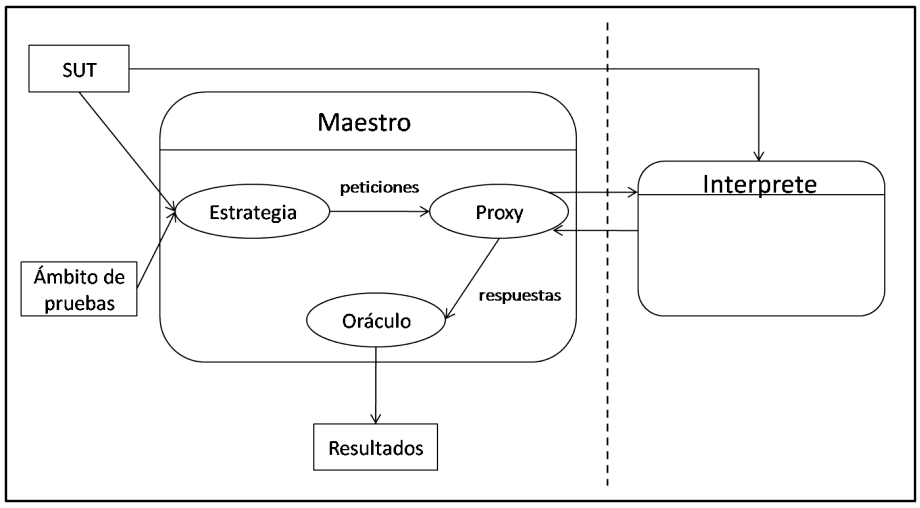

Figura 5.1: Arquitectura de *AutoTest*

5.2.3. Generación de pruebas

Cuando se requiere probar una función de alguna clase bajo prueba, se necesita una instancia de esa clase y los argumentos de la función. El algoritmo de generación de datos de entrada de *AutoTest* usa la técnica de pruebas aleatorias plus (ver sección 4.3.1). Si se necesita un objeto de cierto tipo, se crea llamando aleatoriamente a alguno de los procedimientos de creación de objetos de dicha clase. Una vez creado, el objeto es agregado a una pila de objetos. Entonces un modificador aleatorio es llamado para modificar algún objeto de la pila, y después un objeto es seleccionado aleatoriamente de la pila [13].

A continuación, se muestran los pasos para probar un conjunto de clases en *AutoTest* [12] (ver figura 5.2):

- Genera instancias de las clases bajo prueba.
- Selecciona algunos de los objetos a probar.
- Selecciona los argumentos de las funciones a ser llamadas.
- Ejecuta las pruebas.
- Verifica la salida: verifica si pasa o falla aplicando los contratos como oráculos.

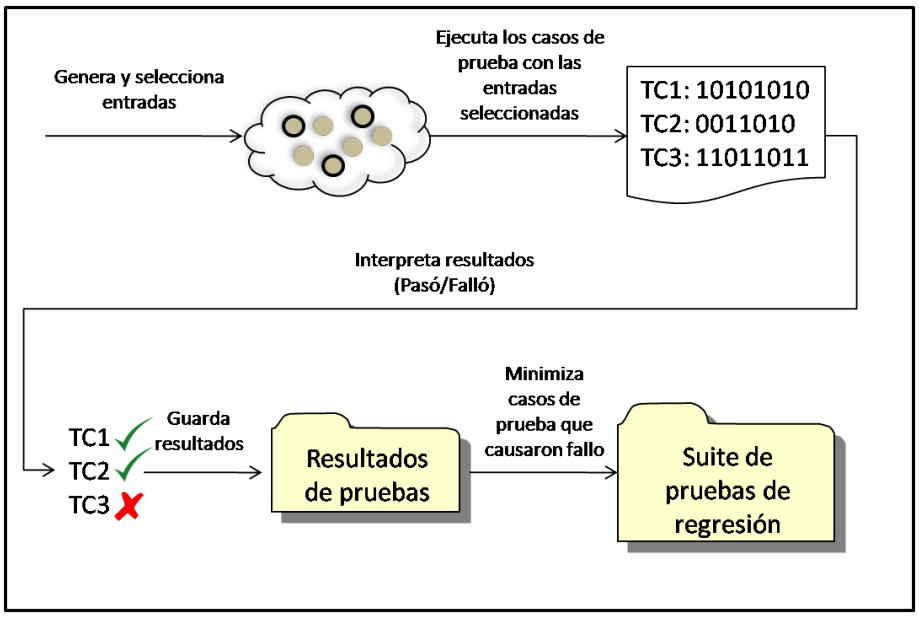

Figura 5.2: Proceso automático de generación de pruebas de *AutoTest*

- Guarda los resultados de las pruebas y los casos de prueba que revelaron algún fallo.
- Construye una forma minimizada de cada caso de prueba guardado y lo agrega a la *suite* de pruebas de regresión.

5.2.4. Diseño por contrato

Como mencionamos en el capítulo 3, las pruebas automáticas automatizan no sólo la ejecución de las pruebas, sino también la generación de los casos de pruebas y la verificación de los resultados. Una automatización completa de las pruebas de software debería de ser capaz de probar el software sin necesitar de ninguna intervención del probador [11].

Las pruebas dirigidas por contratos logran una automatización completa haciendo uso de contratos como oráculos de prueba. Los contratos son precondiciones, postcondiciones e invariantes incrustadas dentro del código fuente del programa bajo prueba [11].

El diseño por contrato [11–13, 52, 53] es una metodología para el diseño e implementación de aplicaciones y componentes popularizada por el lenguaje de programación Eiffel

[54] que hace uso de estos contratos.

Los contratos de software se especifican mediante la utilización de expresiones lógicas denominadas aserciones [11, 12]. Se denominan aserciones porque son condiciones que deben cumplirse. En el diseño por contrato se utilizan los siguientes tipos de aserciones:

- Precondiciones
- Postcondiciones
- Invariantes de Clase
- Variantes e invariantes de ciclo
- Instrucciones *check*

Las precondiciones sirven para filtrar las entradas invalidas; las postcondiciones sirven para detectar fallas en el sistema bajo prueba [11]. Cualquier caso de prueba que cometa una violación de contrato, excepto los casos de prueba que violen la precondición de la rutina bajo prueba (caso de prueba invalido), es marcado como "falló". Si todos los contratos se cumplieron durante la ejecución de un caso de prueba, el caso de prueba es marcado como "pasó" [11].

5.3. DART

DART (*Directed Automated Random Testing*) es una herramienta que sirve para probar software de manera automática y que combina tres técnicas principales [6]:

- Extracción automática de la interfaz de un programa con su ambiente externo usando análisis gramatical de código fuente.
- Generación automática de un manejador de pruebas para esta interfaz. Realiza pruebas aleatorias para simular el ambiente más general en el que el programa puede operar.
- Análisis dinámico de cómo el programa se comporta bajo pruebas aleatorias y generación automática de nuevas entradas de prueba para dirigir sistemáticamente la ejecución sobre rutas de programa alternativas.

Juntas, estas tres técnicas constituyen lo que se denomina Pruebas Aleatorias Automáticas Dirigidas o *DART* por sus siglas en inglés.

El principal objetivo de *DART* es llevar a cabo pruebas que se ejecuten automáticamente en cualquier programa que compile (no hay necesidad de escribir un manejador de pruebas o algún arnés de pruebas). Durante las pruebas, *DART* detecta errores estándar tales como finalizaciones de programas (interrupción por fallos), violaciones de aseveraciones, y no terminación de programas.

5.4. JCrasher

Es una herramienta para pruebas automáticas para código Java. *JCrasher* [55] genera aleatoriamente entradas en un intento de causar que una aplicación Java falle, i.e., que lance una excepción no controlada. Esta herramienta toma el *bytecode* de un programa en Java como su entrada y produce una serie de casos de prueba aleatoriamente, en busca de entradas que causen que el programa objetivo falle. Ayuda a probar varios escenarios inesperados, ya que busca la manera de combinar métodos a partir de las interfaces públicas disponibles para crear datos y estados que son del tipo correcto, pero potencialmente erróneos.

JCrasher define heurísticas para determinar si una excepción de Java debería ser considerada como un error en el programa o debería tomarse como que alguna entrada violo alguna de las precondiciones del código. Otra característica de esta herramienta es que se asegura que cada ejecución de las pruebas se haga con "borrón y cuenta nueva": esto quiere decir que los cambios que realizó una prueba anterior a datos estáticos no afectan a la prueba actual. Puede ser usado en modo consola o como un *plug-in* del *IDE* Eclipse.

5.5. RANDOOP

Es una herramienta en la que se implemento la técnica *Feedback-Directed Random Testing* (ver sección 4.3.8). *RANDOOP* (*Random tester for Object Oriented Programs*) [56] es una herramienta que, automática y aleatoriamente, genera pruebas unitarias para

código en Java (también existe una versión para *.NET*).

RANDOOP toma como entrada un conjunto de clases a probar, un tiempo limite (dentro del cual se realizarán las pruebas) y un conjunto de "verificadores de contratos" (opcional). Como salida, *RANDOOP* genera dos suites de prueba (ver figura 5.3). Una contiene pruebas que violan los contratos, las cuales exhiben escenarios donde el código bajo prueba conduce a la violación de un contrato. La segunda suite de pruebas, contiene pruebas de regresión. Estas pruebas no violan ningún contrato ni causan ninguna excepción, sino que exhiben un comportamiento normal. Los casos de prueba que exhiben un comportamiento ilegal son descartados.

La manera en la que trabaja *RANDOOP* es la siguiente (ver figura 5.3):

1. Crea secuencias de métodos incrementalmente, seleccionando aleatoriamente una llamada a un método y argumentos a partir de secuencias ya generadas.
2. Tan pronto como es creada, una nueva secuencia es ejecutada y comprobada contra un conjunto de contratos.
3. Las secuencias que violan algún contrato forman parte de la suite de pruebas que violan los contratos, la cual se da como salida al usuario.
4. Las secuencias que exhiben un comportamiento normal (i.e., no violan ningún contrato ni causan ninguna excepción) forman parte de las pruebas de regresión, que también se dan como salida al usuario.
5. Las secuencias que exhiben un comportamiento ilegal (violan contratos o causan excepciones) son descartadas.
6. Sólo las secuencias que exhiben un comportamiento normal son usadas para generar nuevas secuencias de métodos.

5.6. Otras herramientas

En la tabla 5.1 se puede ver un resumen de las herramientas ya mencionadas y algunas otras que realizan pruebas automáticas.

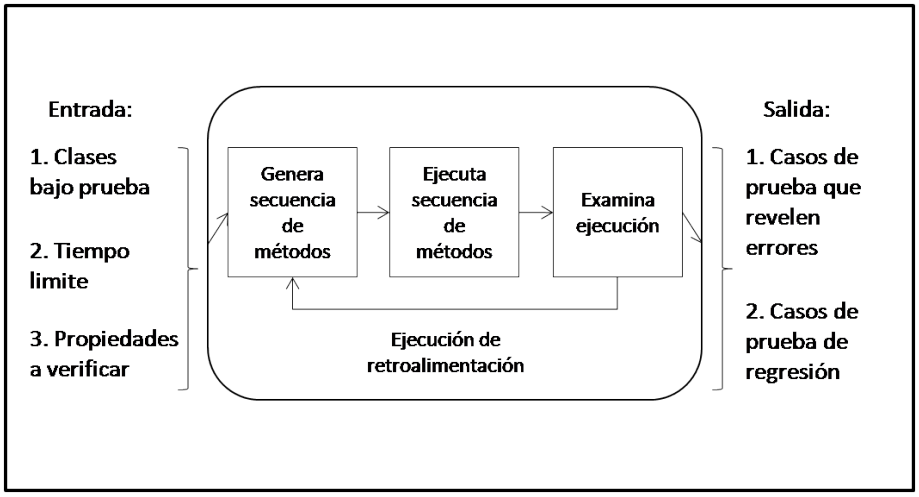

Figura 5.3: Flujo de trabajo de *RANDOOP*

Herramienta	Disponible	Lenguaje	Entrada	Salida	Licencia
AgitarOne	Sí	Java	Código fuente	Pruebas en JUnit, resultados de pruebas	Comercial con 30 días de evaluación
Austin	Sí	C	Código fuente	Resultados de pruebas	Licencia BSD
AutoTest	Sí	Eiffel	Código fuente	Pruebas en Eiffel, resultados de pruebas	Comercial y código abierto
Check n Crash	Sí	Java	Código fuente	Pruebas en JUnit	Licencia MIT
C++test	Sí	C/C++	Código fuente y	Pruebas unitarias código binario	Comercial con 14 días de evaluación
DART	No	C	Código fuente	Resultados de pruebas	-
Eclat	Sí	Java	Código fuente y una prueba exitosa	Pruebas en JUnit	Licencia MIT
Jartage	No	Java	Código fuente	Pruebas en JUnit	-
JCrasher	Sí	Java, JML	Código fuente	Pruebas en JUnit, resultados de pruebas	Licencia MIT
JTest	Sí	Java	Código fuente	Prueba en JUnit	Comercial con 14 días de evaluación
Korat	Sí	Java	Especificaciones y pruebas manuales	Fallos por violación de contratos	Código abierto
QuickCheck	Sí	Haskell	Especificaciones y funciones	Resultados de pruebas	Licencia BSD
RANDOOP	Sí	Java, .NET	Especificaciones, código fuente y tiempo	Pruebas en JUnit, violaciones de contratos	Licencia MIT
TestEra	Sí	Java	Especificaciones y pruebas manuales	Fallos por violación de contratos	Código abierto
YETI	Sí	Java, JML, .NET	Código fuente, tiempo, otras opciones	Pruebas unitarias, fallos encontrados	Licencia BSD

Tabla 5.1: Herramientas que realizan pruebas automáticas [1, 2]

5.7. Resumen

En este capítulo presentamos a *DART* [6], una herramienta que detecta errores estándar tales como terminación incorrecta de programas, violaciones de aseveraciones, y no terminación de programas. A *JCrasher* [55], una herramienta para pruebas automáticas para código Java, que genera aleatoriamente entradas en un intento de causar que una aplicación Java falle, es decir, que lance una excepción no controlada. Y también vimos a detalle a la herramienta *AutoTest* [11, 12], que es una colección de herramientas que automatizan el proceso de pruebas en programas que cuentan con ciertos elementos de su propia verificación, esto es, contratos. Además hablamos del diseño por contrato [11, 12], el cual se compone de precondiciones, postcondiciones e invariantes. *AutoTest* los usa para verificar la correctitud de las ejecuciones, es decir, los usa como oráculos de prueba.

En el siguiente capítulo (capítulo 6) se expone la herramienta desarrollada en este trabajo de tesis. Para realizar nuestra herramienta, se requirió de realizar las investigaciones sobre las pruebas de software en general (capítulo 2), las pruebas automáticas de software (capítulo 3), las pruebas aleatorias de software (capítulo 4) y las herramientas que soportan pruebas automáticas y aleatorias presentadas en este capítulo.

Capítulo 6

AutoTest4C

6.1. Introducción

Nuestra herramienta, al igual que *AutoTest*, trata de combinar los enfoques de las pruebas automáticas y las pruebas manuales. Debemos tomar en cuenta que las pruebas automáticas requieren menos esfuerzo del lado del desarrollador, pero no pueden reemplazar a las pruebas manuales. Esto es debido a que los desarrolladores, al conocer el código del programa, son mejores estableciendo datos complejos de entrada y encontrando casos de prueba que tengan mayor probabilidad de descubrir un *bug*. Por esta razón, es necesario tener una herramienta que unifique las pruebas automáticas y manuales.

En este capítulo se presenta primero el tipo de pruebas que permite realizar esta herramienta (sección 6.2.1). Después, describiremos a detalle los aspectos principales del diseño de la herramienta que se desarrolló en este trabajo. Después, mencionaremos sus características (sección 6.2.2), su arquitectura (sección 6.2.3) y cada uno de los componentes que componen a la herramienta *AutoTest4C*. De manera general, nuestra herramienta se divide en tres módulos o componentes: la interfaz gráfica de usuario (*GUI*, por sus siglas en inglés) (sección 6.2.5), la estrategia (sección 6.2.6) y el compilador y ejecutor (sección 6.2.7). Finalmente se describen los tipos de resultados que pueden arrojar las pruebas realizadas con *AutoTest4C* (sección 6.2.8).

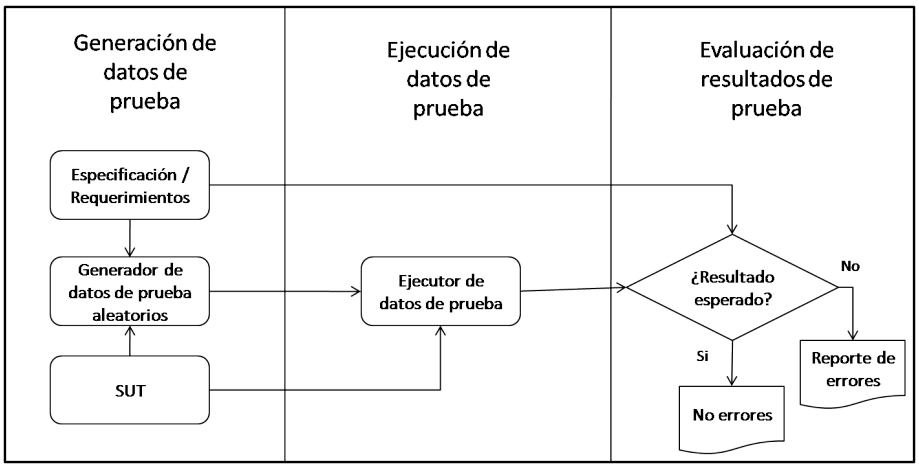

Figura 6.1: Generación y ejecución automática de pruebas aleatorias

6.2. Análisis y diseño de herramienta AutoTest4C

AutoTest4C (***Auto****mated* ***Test****ing* ***for C*** *programs*), al igual que *AutoTest* (ver sección 5.2), trata de combinar los enfoques de las pruebas automáticas y de las pruebas manuales. Debemos tomar en cuenta que las pruebas automáticas requieren menor esfuerzo del lado del desarrollador, pero no pueden reemplazar a las pruebas unitarias manuales. Los desarrolladores son mejores diseñando complejos datos de entrada y encontrando casos de prueba que descubran un *bug*.

El proceso de generar pruebas aleatoriamente y ejecutarlas automáticamente se compone de tres aspectos (ver figura 6.1): generación de datos de prueba, ejecución de datos de prueba y evaluación de resultados de prueba. Estos tres aspectos de *AutoTest4C* serán explicados en las secciones 6.2.9, 6.2.10 y 6.2.11, respectivamente.

Después de un pequeño análisis sobre distintas herramientas que soportan pruebas automáticas (ver capítulo 5), definimos ciertos requerimientos para *AutoTest4C* que se traducen en características de la herramienta, las cuales detallaremos en la sección 6.2.2.

6.2.1. Software a probar

AutoTest4C fue propuesta como una herramienta que permite realizar pruebas automáticas y al mismo tiempo, permitir la integración de las pruebas manuales (escritas por el probador) para programas escritos en lenguaje C.

Tipos de pruebas de software

Esta herramienta realiza pruebas unitarias. Como ya se había mencionado en la sección 2.7.2, las pruebas unitarias se encargan de probar individualmente subprogramas, subrutinas o procedimientos en un programa [27]. Tomamos como una "unidad a probar" a una función de un programa en C.

Para poder diseñar casos de prueba para pruebas unitarias, se necesitan dos tipos de información: una especificación de la unidad y el código fuente de la misma [27]. La especificación define los parámetros de entrada y la salida de la unidad. Recordemos que un caso de prueba se compone de un conjunto de entradas (datos) de prueba, condiciones de ejecución y resultados esperados [26]. De modo que para generar los casos de prueba automáticamente, se deben generar también los resultados esperados o algún otro mecanismo para verificar si el resultado de la prueba fue el correcto (oráculo de prueba). Debido a que *AutoTest4C* no cuenta con un oráculo de prueba o algún otro mecanismo para verificar si el resultado arrojado por la función bajo prueba es el correcto, estrictamente hablando, no genera casos de prueba automáticamente; pero sí genera los datos de prueba con los cuales se realizan las pruebas unitarias.

Con el fin de obtener la información necesaria para generar los datos de prueba es necesaria la especificación o declaración de la función (o funciones) a probar. Por ejemplo, el código 6.1 establece la declaración de una función llamada "foo" que recibe dos parámetros de tipo entero como entrada y regresa un entero como salida:

```
int foo(int, int);
```

Listado 6.1: Declaración de función foo en C

De modo que el prototipo tomado como una unidad de prueba es el siguiente:

```
<type_return> function_name
 (<type_paramater_1> [name_parameter_1], ...,
 <type_paramater_n> [name_parameter_n]);
```

Listado 6.2: Prototipo de unidad de prueba en C para AutoTest4C

6.2.2. Características de la herramienta AutoTest4C

Las principales características con las que cuenta nuestra herramienta son:

- **Soporte de pruebas automáticas y manuales**.
- **Generación automática de casos de prueba**: esto mediante una estrategia que puede ser cambiada, aunque no por el probador (no inicialmente al menos).
- **Permite determinar duración de las pruebas**: una función se prueba n veces, en donde n es especificado por el usuario o también podrá especificar la profundidad en tiempo (en minutos).
- **Prioridad a pruebas manuales**: si el usuario especificó la duración de las pruebas, los casos de prueba manuales (en caso de existir) son ejecutados primero ya que la existencia de estos casos de prueba indica que el usuario está más interesado en esas pruebas. En caso contrario, únicamente se generan y ejecutan tantos casos de prueba como sea posible.
- **Las pruebas manuales y automáticas se deben ejecutar de una sola vez**: se debe mostrar únicamente un conjunto de resultados que incluya a las pruebas manuales y automáticas.
- **Los resultados de las pruebas se escriben en un archivo HTML**.
- **Los resultados muestran qué casos de prueba fallaron**.
- **Resiliencia (*Resilience*)**: se refiere a la capacidad de recuperación. Es decir, en una *suite* de pruebas es probable que existan algunos casos de prueba que, en una ejecución en particular, hagan que el programa finalice de manera inadecuada. Resiliencia

significa que el proceso debe continuar, sin importar los fallos ocurridos, con los casos restantes.

- ***GUI***: interfaz gráfica desde la cual el desarrollador puede proporcionar el software a probar y otras especificaciones opcionales (ver sección 6.2.5).

6.2.3. Arquitectura general de la herramienta AutoTest4C

Los principales componentes de nuestra herramienta (ver figura 6.2) se listan a continuación:

- *GUI* (interfaz gráfica de usuario)
- Estrategia
  - Analizador/Extractor de interfaces
  - Generador de datos de prueba
  - Generador de manejador de prueba
- Compilador y ejecutor

Cada uno de estos componentes se describen en las siguientes secciones de este capítulo.

6.2.4. Ámbito de pruebas

El ámbito de pruebas define las funciones del código en C a probar. Las funciones son seleccionadas desde la *GUI*. En el ámbito de pruebas deben de estar declaradas las funciones de acuerdo al prototipo de función mostrado en el código 6.2.

6.2.5. Interfaz gráfica de usuario

En la interfaz gráfica el desarrollador puede proporcionar el ámbito de pruebas, la profundidad (en número de pruebas o en tiempo), la función oráculo (si es que tiene una), el algoritmo de generación de números aleatorios (elegir uno de los que vamos a proporcionar o uno escrito por el desarrollador) y la semilla del generador de números aleatorios.

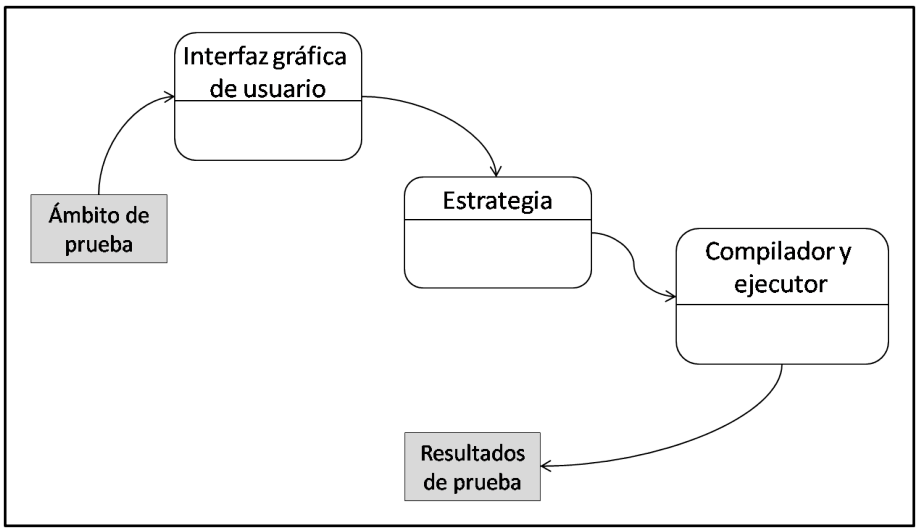

Figura 6.2: Arquitectura general de AutoTest4C

El usuario debe especificar el conjunto de funciones a probar a través de la interfaz gráfica de usuario (*GUI*), a este conjunto de funciones llamamos el *Test Scope* o ámbito de pruebas.

Los resultados de las pruebas se deben escribir en un archivo y podrán ser visualizados en un documento HTML. Deben mostrar qué casos de prueba fallaron, cuántos fallaron, y cuántos pasaron la prueba (no se considera importante saber cuáles pasaron puesto que no se utilizan las pruebas de regresión).

6.2.6. Estrategia

Es un componente que a su vez tiene otros módulos: el analizador/extractor de interfaces, el generador de datos de prueba y el generador de manejador de prueba (ver figura 6.3). La función de estos módulos se explicó en la sección 3.3. El componente estrategia es nuestro componente principal y en el cual se encuentra nuestra mayor aportación, pues es aquí donde reside el algoritmo para generar pruebas aleatorias, así como otros procedimientos que son necesarios para su correcta ejecución (ver capítulo 3).

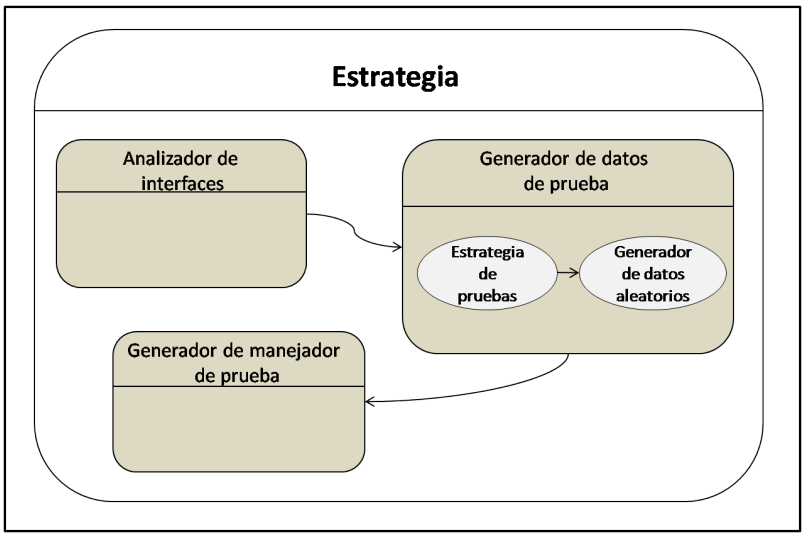

Figura 6.3: Componente Estrategia

Analizador/Extractor de interfaces

El analizador de interfaces se encarga de obtener información acerca del dominio de entrada de cada función a probar, obteniendo los parámetros de entrada (tipo de dato), el nombre de la función a probar y el tipo de dato que devuelve la función.

Generador de datos de prueba

El generador de datos de prueba se encarga de generar, mediante alguna estrategia de generación de datos de prueba (lo que incluye al algoritmo generador de datos aleatorios), los datos de prueba de acuerdo a la información que obtuvo el analizador.

Este módulo, junto con la estrategia de pruebas, es nuestra principal aportación, pues es aquí donde recae la responsabilidad de crear datos de prueba óptimos de manera aleatoria.

Estrategia de pruebas

Este módulo es parte del generador de datos de prueba. Es una parte indispensable del generador de datos de prueba y es también conocido como selector de ruta (*path selector*) en las pruebas de caja blanca. Esta parte define el criterio de selección de datos de prueba. Este criterio puede ser de alta cobertura de código, de selección de caminos, etc.

El generador produce datos de prueba de acuerdo al criterio de selección definido por la estrategia de pruebas. En nuestro trabajo, el criterio es definido por las estrategias aleatorias implementadas (las cuales se mencionan a continuación) y permite encontrar datos de prueba óptimos. Los datos de prueba óptimos pertenecen al subconjunto, del dominio de entradas, que tiene la más alta probabilidad de detectar la mayor cantidad de fallos. Las estrategias aleatorias basan su criterio de selección de datos de prueba en los patrones de fallas explicados en el capítulo 4.

Se implementaron dos estrategias que el usuario (probador) puede usar para generar los datos de prueba. Las estrategias implementadas son: las pruebas aleatorias plus (ver sección 4.3.1) y las pruebas aleatorias adaptativas (ver sección 4.3.2). En especial se creo una estrategia que combina estas técnicas. Los algoritmos utilizados se describen en la sección 6.2.9.

Generador del manejador de prueba

El generador del manejador de prueba, se encargará de generar un manejador de prueba para cada archivo de funciones a probar (archivos .h y .c). Este tendrá algunos prefijos ya definidos: `test_nombreArchivo.h` (ejecución por tiempo y/o por profundidad), `test_nombreArchivo.c`.

6.2.7. Compilador y ejecutor

Se encargan de compilar y ejecutar los archivos manejadores de pruebas. En la parte de la ejecución deben crear los datos de prueba e invocar a las funciones en el sistema bajo prueba. También se encargan de comparar los resultados obtenidos con el oráculo (función oráculo ingresada por el probador) y el comportamiento del software durante la ejecución de las pruebas.
En cada llamada a una función a probar, este módulo crea dos hilos: el primero es para ejecutar la función, el segundo se ocupa para determinar si la función bajo prueba entro en un ciclo infinito. La detección de ciclos infinitos es uno de los tipos de defectos que detecta nuestra herramienta (ver sección 6.2.11).

6.2.8. Tipos de resultados de una prueba

Los tipos de resultados de prueba que genera la herramienta son:

- *Pasó* *: No se encontró *bug*.
- *Falló* *: Se encontró algún *bug*.

*Aplica también para pruebas con oráculo escrito por el probador.

6.2.9. Generación de datos de prueba

Las estrategias de generación de datos de prueba implementadas son: pruebas aleatorias plus (ver sección 4.3.1) y las pruebas aleatorias adaptativas (ver sección 4.3.2).

Debido a que la estrategia de pruebas aleatorias plus sólo requiere de agregar casos de prueba especiales (valores límite y valores agregados por el usuario) a los creados aleatoriamente, aquí no es necesario definir su algoritmo.

Para implementar la estrategia de pruebas aleatorias adaptativas, se utiliza una versión de esta estrategia llamada *Fixed Size Candidate Set ART (FSCS-ART)* [43], la cual consta de dos algoritmos. El algoritmo 6.3 [43] es usado para seleccionar el mejor caso de prueba y generar un conjunto candidato.

```
1  /*
2  selected_set := { test data already selected };
3  candidate_set := {};
4  total_number_of_candidates := 10;
5  */
6  function Select_The_Best_Test_Data(selected_set, candidate_set,
       total_number_of_candidates);
7     best_distance := -1.0;
8     for i := 1 to total_number_of_candidates do
9        candidate := randomly generate one test data from the program input domain, the
             test data cannot be in candidate_set nor in selected_set;
10          candidate_set := candidate_set + { candidate };
11          min_candidate_distance := Max_Integer;
12          foreach j in selected_set do
13             min_candidate_distance := Minimum(min_candidate_distance, Euclidean_Distance
                   (j, candidate));
14          end_foreach
15          if (best_distance < min_candidate_distance) then
```

```
16              best_data := candidate;
17                  best_distance := min_candidate_distance;
18          end_if
19      end_for
20      return best_data;
21  end_function
```

Listado 6.3: Algoritmo 1 de *ART*

El algoritmo 6.4 [43] comienza el proceso de *ART*, el cual llama al algoritmo 6.3 para generar el conjunto candidato, ejecuta los casos de prueba candidatos y actualiza el conjunto ejecutado hasta encontrar una falla.

```
1  initial_test_data := randomly generate a test data from the input domain;
2  selected_set := { initial_test_data };
3  counter := 1;
4  total_number_of_candidates := 10;
5  use initial_test_data to test the program;
6  if (program output is incorrect) then
7     reveal_failure := true;
8  else
9     reveal_failure := false;
10 end_if
11 while (not reveal_failure) do
12    candidate_set := {};
13    test_data := Select_The_Best_Test_Data(selected_set, candidate_set,
          total_number_of_candidates);
14    use  test_data to test the program;
15    if(program output is incorrect) then
16       reveal_failure := true;
17    else
18       selected_set := selected_set + { test_data };
19       counter := counter + 1;
20    end_if
21 end_while
22 output counter;
```

Listado 6.4: Algoritmo 2 de *ART*

La razón por la cual el número de candidatos (*total_number_of_candidates*) es inicializado en 10, es porque Chen et al. [43] descubrieron que el tamaño del conjunto candidato afecta el rendimiento de *ART*. Mientras más grande sea el conjunto candidato menos casos de prueba son requeridos para detectar el primer fallo. Chen et al. [43] descubrieron que a partir de conjuntos candidatos de 10 elementos no había mucha diferencia en cuanto al número de casos de prueba requeridos para detectar el primer fallo. Por esta razón es que

el número de candidatos es inicializado en 10.

6.2.10. Ejecución de datos de prueba

La ejecución de los datos de prueba se hacen por medio de pequeños *scripts*. Se crearon dos *scripts*: uno para Windows (ver código 6.5) y otro para Ubuntu (ver código 6.6).

Listado 6.5: Script para ejecutar pruebas automaticamente en Windows

```
@echo off
echo Cambiando a directorio: 'dirProyect'
cd 'dirProyect'
echo Compilando archivos manejadores de prueba, espere...
gcc TAUTOTEST.c ManejadorDeExcepcionesEnC\e4c.c
echo Ejecutando archivos manejadores de prueba, espere...
a.exe
Resultados\Resultados.html
pause
exit
```

Listado 6.6: Script para ejecutar pruebas automaticamente en Ubuntu

```
#!/bin/bash
echo Cambiando a directorio: 'dirProyect'
echo Compilando archivos manejadores de prueba, espere...
gcc TAUTOTEST.c ManejadorDeExcepcionesEnC/e4c.c
echo Ejecutando archivos manejadores de prueba, espere...
./a.out
Resultados/Resultados.html
```

6.2.11. Evaluación de resultados de pruebas

AutoTest4C no usa ningún oráculo de pruebas basado en contratos (ver sección 3.4). En lugar de eso, se utilizan dos mecanismos para verificar el comportamiento adecuado del software bajo prueba:

1. **Un oráculo de prueba ingresado por el usuario**: es un mecanismo mediante el cual se verifica si el resultado arrojado por la función probada es correcto.

2. **Un manejador de excepciones**: para detectar fallos en tiempo de ejecución.

Estos mecanismos permiten detectar dos tipos de defectos, los cuales se describen a continuación.

Tipos de defectos de software

AutoTest4C permite encontrar dos tipos de defectos:

- **Defectos en tiempo de ejecución**: este tipo de defectos ocurren cuando el programa está en ejecución.

- **Defectos que causan resultados erróneos**: este tipo de errores son los más difíciles de corregir, ya que no hay indicio de donde pudo haber ocurrido el error. Es decir, el programa parece funcionar bien, pero el resultado que arroja no es el que debería de arrojar.

El primer tipo de defecto o *bug* se detecta por medio de un manejador de excepciones, el cual se describirá en la sección 6.3.1. Por medio de este manejador de excepciones podemos detectar defectos que provocan los siguientes fallos:

- División por cero
- Problemas aritméticos
- Desbordamientos de arreglos o estructuras
- Ciclos infinitos
- *Bad pointer exception*
- *Null pointer exception*

El segundo tipo de defecto se detecta por medio de oráculos que el probador debe escribir para poder comprobar la salida que arroja la función. De esta manera, se comprueba la salida obtenida con la salida esperada.

6.3. Implementación de la herramienta AutoTest4C

En esta sección se presentan algunos detalles de la implementación de nuestra herramienta. Primeramente, se describen algunos detalles de desarrollo del manejador de excepciones. En seguida, se muestra la interfaz gráfica de usuario desarrollada. Por último, se muestran las plantillas implementadas para mostrar los resultados de las pruebas.

6.3.1. Manejador de excepciones en C

Se ha utilizado la biblioteca *exceptions4c* [1] como manejador de excepciones para código escrito en C. Esta biblioteca es un *framework* para manejar excepciones en C, con licencia *GNU Lesser GPL*. Provee las siguientes macros para usar:

- *try*
- *catch*
- *finally*
- *throw*

Un ejemplo usando los bloques *try*/*catch*/*finally* es el siguiente:

```
1  #include "e4c.h"
2
3  int foobar(){
4     int foo;
5     void * buffer = malloc(1024);
6
7     if(buffer == NULL){
8       throw(NotEnoughMemoryException,"Could not allocate buffer");
9     }
10
11    try{
12       foo = get_user_input(buffer, 1024);
```

[1] https://code.google.com/p/exceptions4c/

```
13      }catch(BadUserInputException){
14          foo = 123;
15      }finally{
16          free(buffer);
17      }
18
19      return(foo);
20  }
```

Listado 6.7: Ejemplo de uso de biblioteca *exceptions4c*

Las excepciones manejadas por este *framework* están organizadas por jerarquías. La excepción *RuntimeException* es la raíz de esta jerarquía. De modo que cualquier excepción puede ser capturada por un bloque *catch(RuntimeException)*.

6.3.2. Interfaz gráfica de usuario

La interfaz gráfica de usuario fue implementada usando la biblioteca de código libre *wxWidgets* [2] con C++. Se eligió esta biblioteca debido a que es multiplaforma y, personalmente hablando, es con la que se tiene mayor experiencia de desarrollo.

En las figuras 6.4, 6.5 y 6.6 se muestran las principales ventanas que conforman a la interfaz gráfica de usuario de AutoTest4C y que corresponden con los pasos para la creación de un proyecto de pruebas.

En la figura 6.4 se muestra la ventana inicial de la *GUI* de *AutoTest4C*. Es en esta ventana donde el usuario debe proporcionar el código fuente del programa a probar. Esto se hace en la parte de implementación. En la parte de la interfaz, el usuario define que funciones del código indicado se van a probar (ámbito de pruebas). El usuario puede asignarle un nombre al nuevo proyecto de pruebas a crear o dejar el nombre que se asigna automáticamente. Se debe proporcionar un directorio de salida donde se guardarán los resultados de las pruebas.

[2] https://www.wxwidgets.org/

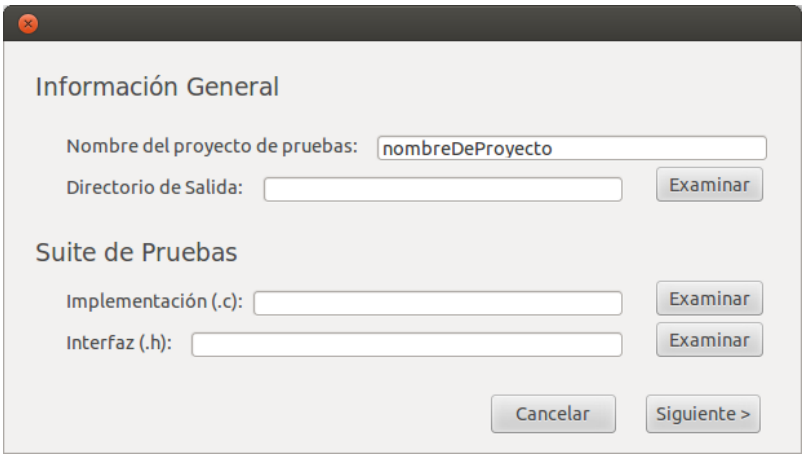

Figura 6.4: 1er ventana de GUI de AutoTest4C

En la figura 6.5 se muestra la interfaz donde el usuario puede definir que datos especiales se agregaran a los datos de prueba generados aleatoriamente. Esos valores son para ser utilizados por la estrategia aleatoria plus.

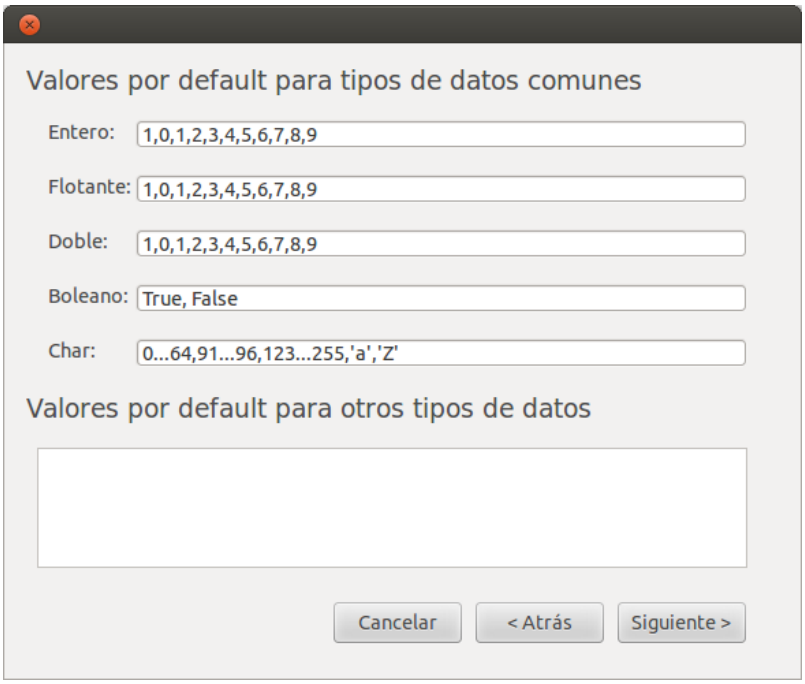

Figura 6.5: 2da ventana de GUI de AutoTest4C

En la figura 6.6 se muestra la opción que da *AutoTest4C* para poder ingresar un orácu-

lo que se haya programado manualmente, elegir el algoritmo de generación de números aleatorios de los disponibles, insertar la semilla de este algoritmo, colocar la duración de las pruebas en tiempo o número de pruebas y definir el tiempo límite de ejecución para cada función a probar (si la función no termina su ejecución en ese tiempo se determina que entro en un ciclo infinito y se fuerza a terminar su ejecución).

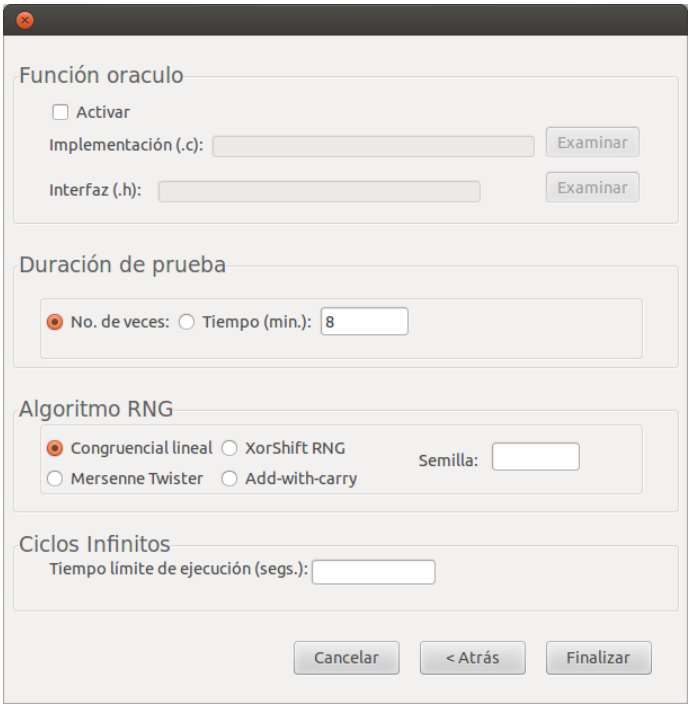

Figura 6.6: 3er ventana de GUI de AutoTest4C

6.3.3. Despliegue de resultados

Se desarrollaron plantillas en código HTML y CSS para mostrar los resultados. Estas plantillas se muestran en las figuras 6.7 y 6.8.

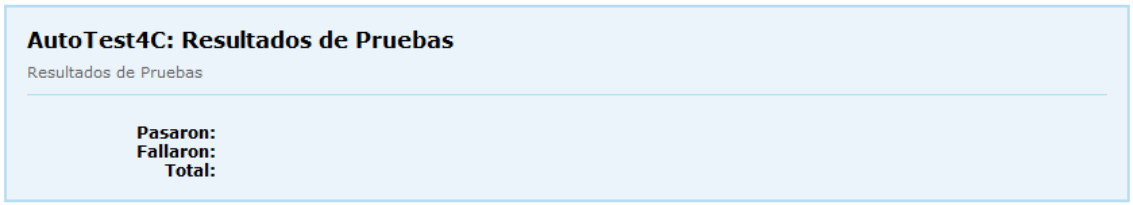

Figura 6.7: Plantilla 1 de resultados de prueba en HTML de AutoTest4C

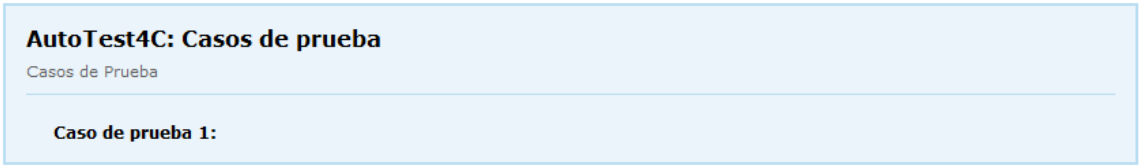

Figura 6.8: Plantilla 2 de resultados de prueba en HTML de AutoTest4C

6.4. Resumen

En este capítulo presentamos a *AutoTest4C*, una herramienta que permite la generación aleatoria y ejecución automática de pruebas unitarias en programas escritos en el lenguaje de programación C. Las principales características de nuestra herramienta son el soporte de pruebas automáticas y manuales, la generación automática y aleatoria de datos de prueba, y el reporte de los resultados en un archivo HTML donde el probador pueda verificar cuantas y que pruebas fallaron y cuantas pasaron. *AutoTest4C* ofrece dos mecanismos para verificar el comportamiento adecuado del software bajo prueba: (1) verificación mediante una función oráculo ingresada por el usuario y (2) verificación mediante manejador de excepciones que permite detectar fallos en tiempo de ejecución.

Capítulo 7

Pruebas y resultados

7.1. Introducción

En este capítulo se muestran algunos de los casos de estudio que usamos para probar nuestra herramienta. Nuestra búsqueda de casos de estudio, con los cuales probar la herramienta desarrollada, se baso en los siguientes aspectos:

1. *Benchmarks* para evaluar herramientas de pruebas y detección de *bugs*.
2. Repositorio de programas con *bugs* reportados.
3. Programas con errores sembrados.

Los casos de estudio ideales serían aquellos que provienen de un *benchmark* o de un reportes de errores conocidos en programas y que, además, el código de esos programas esté disponible. La mayoría de las herramientas estudiadas son y vistas en el capítulo 5 realizan pruebas de código escrito en lenguajes de programación distintos de C. Algunas herramientas realizan pruebas con casos de estudio que ellos mismos desarrollan y que no hacen público, i.e., programas con errores sembrados. Otras herramientas, principalmente las dirigidas a programas en Java, realizan pruebas con *benchmarks* disponibles. Al haber pocas herramientas que realicen pruebas en programas escritos en C, es muy difícil encontrar un *benchmark* o repositorios con programas en C con errores.

Se encontraron pocos *benchmarks* que dicen tener programas con errores escritos en diferentes lenguajes de programación, entre ellos C. Sin embargo, algunos como *BugBench*

[57] mencionan que su trabajo está en desarrollo y que, una vez terminen, lo harán publico en su página web (la cual no mencionan). Un *benchmark* cuyo sitio web está disponible es *ClabureDB* [58], desafortunadamente, para el momento en que se visitó su sitio web, no se podía acceder a los reportes de errores.

Por estas razones es que se decidió tomar los casos de estudio de algoritmos conocidos y otros programas encontrados en ciertos libros cuyos autores hacen publico el código fuente. Los programas que usamos para las pruebas son funciones que ejecutan algún algoritmo, como: calcular si un número es número primo, *insertionSort*, *bubblesort*, etc.). Tales casos de estudio y el proceso de pruebas realizado se describen en las siguientes secciones.

7.2. Usando oráculos de prueba

Este tipo de pruebas se realizaron usando el oráculo de prueba que el usuario tiene que proveer. Para lo cual, se desarrollaron funciones que realizaran la misma operación que las funciones a probar pero de una manera distinta.

Se realizaron pruebas a distintas funciones que implementan distintos algoritmos. Dentro de estos programas se pusieron algunos mutantes (cambios en el código fuente del programa con el fin de sembrar errores en el mismo) que modifican el comportamiento de las funciones bajo prueba. Los resultados de estas pruebas se muestran en la tabla 7.1.

7.3. Usando el manejador de excepciones

Este tipo de pruebas se realizaron usando sólo el manejador de excepciones. Es decir, no se comparo si el resultado devuelto por la función bajo prueba era el correcto, sólo se detectaron fallas ocasionadas por alguna excepción.

Se realizaron pruebas a distintas funciones que implementan distintos algoritmos. Dentro de estos programas se pusieron algunos mutantes que modificarán el comportamiento del programa de modo que llevaran a una excepción. Los resultados de estas pruebas se

Nombre de programa	Dimensión de dominio de entrada	Fallos de resultado incorrecto sembrados	Fallos en tiempo de ejecución sembrados	No. total de fallos detectados	No. total de pruebas ejecutadas
esPrimo	1	20	7	27	100
euclidesMCD	2	38	4	42	100
divide	2	0	21	21	100
bubbleSort	2	27	0	27	100
selectionSort	2	34	0	34	100
insertionSort	2	13	0	13	100
heapSort	2	18	0	18	100

Tabla 7.1: Resultados de pruebas de software con errores sembrados

muestran en la tabla 7.1.

En la tabla 7.2 se muestran los errores sembrados en las funciones probadas cuyos resultados se muestran en la tabla 7.1.

Se realizaron otras pruebas sobre casos de estudio seleccionados de diferentes fuentes, a estos casos de estudio no se les sembraron errores. Los resultados de estas pruebas se muestran en la tabla 7.3.

Para estas pruebas solo se crearon funciones oráculo para algunas de las funciones probadas, es por esta razón que algunos de los fallos encontrados por resultado incorrecto son nulos. Los casos de estudio fueron tomados de los siguientes recursos:

- Código fuente del libro *Data structures and algorithm analysis in C*[1].
- Código fuente del libro *Sorting and searching algorithms: A cookbook*[2].
- Página web donde son publicados distintos algoritmos de distintas áreas de la computación en distintos lenguajes de programación[3].

[1] http://users.cis.fiu.edu/~weiss/dsaa_c2e/files.html
[2] https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.htm
[3] http://www.scriptol.com/programming/list–algorithms.php

Nombre de programa	Código original	Errores sembrados	Fallo de resultado incorrecto	Fallo en tiempo de ejecución
esPrimo	*return 1;*	*return 0;*	x	
	while (res != 0)	*while (res == res)*		x
euclidesMCD	*if (num1 >num2)*	*if (num1 <num2)*	x	
	while (r != 0)	*while (r == r)*		x
divide	*if(divisor != 0)*	*elimina línea*		x
bubbleSort	*if (a[j] > a[j+1])*	*if (a[j] > a[j+1] - 2)*	x	
selectionSort	*if (a[j] < a[min])*	*if (a[j] < a[min] + 2)*	x	
insertionSort	*j > 0 && a[j-1] > index*	*j > 0 && a[j-1] > index - 2*	x	
heapSort	*downHeap(a, 0, i-1);*	*downHeap(a, 0, i-2);*	x	

Tabla 7.2: Errores sembrados en código de funciones probadas. En la segunda columna se muestra la línea de código en su forma original. La columna tres muestran el cambio sobre la línea de código original que ocasiona un fallo del tipo mostrado en las columnas cuatro y cinco.

Nombre de programa	Dimensión de dominio de entrada	Fallos de resultado incorrecto	Fallos en tiempo de ejecución	No. total de fallos detectados	No. total de pruebas ejecutadas
fig1_2	1	34	0	34	1000
fig1_3	1	67	0	67	1000
fig1_4	1	24	1	25	1000
fig2_9	3	41	0	41	1000
fig2_10	2	0	11	11	1000
cocktail–sort	2	123	1	124	1000
comb–sort	2	46	1	47	1000
gnome–sort	2	62	1	63	1000
sort	2	68	14	82	1000
tiny-encryption	2	0	5	5	1000

Tabla 7.3: Resultados de pruebas realizadas a código libre

Algunos casos de estudio tienen como datos de entrada a un arreglo de tipo *t* y un dato tipo *int* que define el tamaño del arreglo. Puesto que nuestro analizador toma ambos argumentos como independientes uno del otro y se crean valores aleatorios para cada argumento (lo que implica que el argumento que define el tamaño del arreglo no defina el tamaño del arreglo), se creo una estructura de dato cuyo tipo esta declarado como sigue:

```
1 typedef struct{
2    <type> object;      /*puntero a array, <type> = int, float,
        double, ...*/
3    int length;           /*tamano (dimension) de objeto array*/
4 }ARRAY_<TYPE>_TYPE;      /*<TYPE> = INT, FLOAT, DOUBLE, ...*/
```

Listado 7.1: Ejemplo de tipo de dato arreglo

De manera que las funciones que reciban argumentos de tipo arreglo y que requieran otro argumento para conocer el tamaño del arreglo deben utilizar el tipo de dato definido en 7.1. Para esto, deben modificar su código e insertar la siguiente línea *#include "GenCasosPrueba/ArrayType.h"* en Ubuntu.

El código fuente de los algoritmos utilizados como casos de estudio en nuestras pruebas se encuentra en un repositorio público[4] alojado en la Web.

[4] https://github.com/Fer–10/CaseTests_AutoTest4C

Capítulo 8

Conclusiones y trabajo futuro

En este trabajo se implementó una herramienta que permite la generación y ejecución automática de pruebas en programas escritos en el lenguaje de programación C. En este capítulo se presentan las conclusiones de nuestro trabajo y algunas expectativas para el trabajo futuro.

8.1. Conclusiones

En este trabajo se describe el diseño e implementación de una herramienta que permite la generación aleatoria y ejecución automática de pruebas unitarias en programas escritos en el lenguaje de programación C.

Nuestra herramienta, a la que dimos por nombre *AutoTest4C*, se enfoca en tres aspectos de la automatización del proceso de pruebas. Estos tres aspectos son: (1) la gestión y ejecución de las pruebas, (2) la generación de datos de prueba y (3) la evaluación de los resultados de las pruebas. Para permitir que nuestra herramienta automatizara estos aspectos, se desarrollaron tres módulos principales: (1) la interfaz gráfica de usuario, (2) el módulo estrategia y (3) el módulo compilador y ejecutor. La implementación de la gestión y ejecución de las pruebas se encuentra repartida en el módulo 1 y 3. La generación de datos de prueba se encuentra implementada en el módulo 2. La evaluación de los resultados de las pruebas esta implementada en el módulo 3.

La interfaz gráfica de usuario fue implementada usando la biblioteca de código libre *wxWidgets* con C++. Los otros módulos están programados en C. *AutoTest4C* funciona

para los sistemas operativos Windows y Ubuntu.

Para el módulo estrategia, se desarrolló un analizador de interfaces que obtiene la información necesaria de cada función a probar: los parámetros de entrada (tipo de dato), el nombre de la función a probar y el tipo de dato que devuelve la función. También se implementó un generador de datos aleatorios que genera aleatoriamente valores para los tipos de datos primitivos de C.

Se implementaron dos estrategias de generación de datos de prueba: pruebas aleatorias plus y las pruebas aleatorias adaptativas. Puesto que distintas investigaciones han realizado diversas versiones de la estrategia *ART*, se programó la versión de esta estrategia llamada *Fixed Size Candidate Set ART (FSCS-ART)*.

Para la evaluación de los resultados de las pruebas y detección de fallos, se implementaron dos mecanismos:

1. **Un oráculo de prueba ingresado por el usuario**: es un mecanismo mediante el cual se verifica si el resultado arrojado por la función probada es correcto.

2. **Un manejador de excepciones**: para detectar fallos en tiempo de ejecución.

El despliegue de los resultados en archivos HTML permite al usuario el conocer qué datos de provocaron un fallo. Esto da la posibilidad de realizar una depuración con alguna otra herramienta provocando los fallos con esos datos de prueba.

La herramienta desarrollada durante este trabajo puede considerarse como una de las herramientas que soportan pruebas automáticas en los tres aspectos que mencionamos anteriormente: la gestión y ejecución de las pruebas, la generación de datos de prueba y la evaluación de los resultados de las pruebas. Si bien, no realiza estos aspectos tan bien y de una manera más completa como herramientas como *AutoTest*, *JCrasher* o *Randoop* (herramientas que tienen años de desarrollo), puede ser usada como una herramienta en la que implementar otras estrategias de pruebas para aumentar sus características o simplemente para realizar experimentos de investigación de otras técnicas de prueba (como lo es *Randoop*). Hay muy pocas herramientas que realizan pruebas de software en lenguaje C, nuestra herramienta se convierte en una de esas pocas.

8.2. Trabajo futuro

Los siguientes puntos describen algunas propuestas de trabajo futuro:

- Implementar e integrar otras estrategias de generación de datos de prueba. La mejor estrategia de pruebas es aquella que combina los enfoques de las pruebas de caja negra y las pruebas de caja blanca. Es importante tener otras estrategias, no solo de caja negra sino también de caja blanca, que complementen a las que ya se tienen con el fin de realizar pruebas más completas. Integrar una estrategia que permita una cobertura de código de rama sería una buena opción.

- Considerar otros mecanismos de verificación de resultados de pruebas. Con el fin de realizar una completa automatización de las pruebas, es necesario que el usuario no intervenga en ellas. El usuario solo debe preocuparse por qué probar y no el cómo. La mayor parte de la automatización de la evaluación de resultados se basa en el uso de contratos como oráculos de prueba (e.g., diseño por contrato). Sin embargo, esto precisaría de que los programas a probar tengan esos contratos embebidos en el código. Lamentablemente, muy poco software es programado así en el mundo real; lo que genera que al probar las estrategias usando contratos se prueben programas "juguete". Algunas investigaciones se enfocan a detectar fallos en tiempo de ejecución relacionados con el uso de memoria, esta podría ser una buena consideración.

- Integración a *IDE*. Es importante integrar nuestra herramienta a un *IDE* con el fin de poder extender las características que puede implementar. Algunas de estas características pueden ser soporte a pruebas de regresión y desarrollo dirigido por pruebas.

- Integrar algoritmos de minimización de datos de prueba. Obtener el dato de prueba mínimo que manifieste el mismo defecto significa menos esfuerzo para localizar ese defecto.

- Implementar e integrar algún mecanismo de depuración. Nuestra herramienta detecta fallos, pero para realizar un proceso completo de verificación es necesario corregir los defectos que provocan esos fallos. Para poder corregirlos, es necesario localizarlos, y para eso se ocupa de un depurador.

Bibliografía

[1] S. J. Galler and B. K. Aichernig. "Survey on test data generation tools". *International Journal on Software Tools for Technology Transfer*, 15(2):1–25, Springer-Verlag, April 2013.

[2] M. A. Ahmad. *New Strategies for Automated Random Testing*. Ph.D. thesis, Department of Computer Science, The University of York, York, England, November 2013.

[3] B. Meyer. "Seven principles of software testing". *IEEE Computer*, 41(8):99–101, IEEE Computer Society, August 2008.

[4] R. Pressman. *Software Engineering: A Practitioner's Approach*. 6th edition. McGraw-Hill, Inc., New York, NY, USA, 2005.

[5] A. M. Davis. *201 Principles of Software Development*. 1st edition. McGraw-Hill, Inc., New York, NY, USA, 1995.

[6] P. Godefroid, N. Klarlund, and K. Sen. "Dart: directed automated random testing", In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM Press, pages 213–223, Chicago, IL, USA, June 12-15 2005.

[7] B. Korel. "Automated software test data generation". *IEEE Transactions on Software Engineering*, 16(8):870–879, IEEE Computer Society, August 1990.

[8] J. Edvardsson. "A survey on automatic test data generation", In *Proceedings of the Second Conference on Computer Science and Engineering*, ECSEL, pages 21–28, Linköping, Sweden, October 21-22 1999.

[9] M. Asbat and M. Oriol. "Dirt spot sweeping random strategy". *Lecture Notes on Software Engineering*, 2(4):294–299, IACSIT Press, November 2014.

[10] E. Gamma and K. Beck. "Junit: A cook's tour". *Java Report*, 4(5):27–38, May 1999.

[11] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. "Reconciling manual and automated testing: The autotest experience", In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, page 261a, Waikoloa, Big Island, HI, USA, January 3-6 2007.

[12] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. "Programs that test themselves". *IEEE Computer*, 42(9):46–55, IEEE Computer Society, September 2009.

[13] I. Ciupa and A. Leitner. "Automatic testing based on design by contract", In *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, Springer-Verlag, pages 545–557, Erfurt, Germany, September 19-22 2005.

[14] I. Ciupa. *TestStudio: An environment for automatic test generation based on Design by Contract*. Diploma Thesis, ETH Zurich (Swiss Federal Institute of Technology Zurich), Zurich, Switzerland, July 2004.

[15] K. Arnout, X. Rousselot, and B. Meyer. *Test Wizard: Automatic Test Case Generation Based on Design by Contract*. Draft report, ETH Zurich (Swiss Federal Institute of Technology Zurich), Zurich, Switzerland, June 2003.

[16] M. Ben-Ari. "The bug that destroyed a rocket". *SIGCSE Bull.*, 33(2):58–59, ACM Press, June 2001.

[17] D. N. Arnold. *Two disasters caused by computer arithmetic errors*, September 1996. http://www.ima.umn.edu/~arnold/455.f96/disasters.html.

[18] N. G. Leveson and C. S. Turner. "An investigation of the therac-25 accidents". *IEEE Computer*, 26(7):18–41, IEEE Computer Society, July 1993.

[19] G. Goth. *Heartbleed Bug Provokes Open Source Soul-Searching*, April 2014. http://cacm.acm.org/news/174366-heartbleed-bug-provokes-open-source-soul-searching/fulltext.

[20] National Institute for Standards and Technology. *The economic impacts of inadequate infrastructure for software testing*. Planning report 02-3, National Institute of Standards and Technology, May 2002.

[21] K. Beck. *Test Driven Development: By Example*. 1st edition. Addison-Wesley Professional, Boston, MA, USA, 2002.

[22] J. A. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. 1st edition. Addison-Wesley Professional, Westford, MA, USA, 2012.

[23] I. Sommerville. *Ingeniería del software*. 7th edition. Pearson Educación, Madrid, España, 2005.

[24] B. W. Boehm. *Software Engineering Economics*. 1st edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[25] IEEE. "IEEE guide for software verification and validation plans". *IEEE Std 1059-1993*, pages i–87, IEEE Computer Society, December 1994.

[26] IEEE. "IEEE standard glossary of software engineering terminology". *IEEE Std 610.12-1990*, pages 1–84, IEEE Computer Society, December 1990.

[27] G. J. Myers. *Art of Software Testing*. 1st edition. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[28] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. 2nd edition. CRC Press, Inc., Boca Raton, FL, USA, 2002.

[29] J. N. Buxton and B. Randell. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee*. 1st edition. NATO, Rome, Italy, 1969.

[30] S. S. Brilliant, J. C. Knight, and N. G. Leveson. "The consistent comparison problem in n-version software". *SIGSOFT Softw. Eng. Notes*, 12(1):29–34, ACM, January 1987.

[31] B. Beizer. *Software Testing Techniques*. 2nd edition. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[32] P. Ammann and J. Offutt. *Introduction to Software Testing*. 1st edition. Cambridge University Press, New York, NY, USA, 2008.

[33] B. Hailpern and P. Santhanam. "Software debugging, testing, and verification". *IBM Syst. J.*, 41(1):4–12, IBM Corp., January 2002.

[34] P. Evans. *Heartbleed bug: RCMP asked Revenue Canada to delay news of SIN thefts*, April 2014. http://www.cbc.ca/news/business/heartbleed-bug-rcmp-asked-revenue-canada-to-delay-news-of-sin-thefts-1.2609192.

[35] V. Vipindeep and P. Jalote. *List of Common Bugs and Programming Practices to avoid them*, 2005.

[36] S. Mahmood. *A systematic review of automated test data generation techniques*. Master's thesis, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden, October 2007.

[37] D. Hoffman. "A taxonomy for test oracles", In *Quality Week*, ACM Press, pages 52–60, San Francisco, CA, USA, May 26-29 1998.

[38] L. Baresi and M. Young. *Test Oracles*. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[39] R. Hamlet. "Random testing", In *Encyclopedia of Software Engineering*, Wiley, pages 970–978, New York, NY, USA, February 1994.

[40] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. "Finding faults: Manual testing vs. random+ testing vs. user reports", In *Proceedings of the 2008 19th Internatio-*

nal Symposium on Software Reliability Engineering, IEEE Computer Society, pages 157–166, Seattle, WA, USA, November 10-14 2008.

[41] C. Pacheco. *Directed Random Testing*. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 2009.

[42] J. H. Andrews, A. Groce, M. Weston, and R. Xu. "Random test run length and effectiveness", In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, pages 19–28, LÁquila, Italy, 2008.

[43] T. Y. Chen, H. Leung, and I. K. Mak. "Adaptive random testing". In Michael J. Maher (editor), *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pages 320–329, 2005.

[44] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu. "Proportional sampling strategy: guidelines for software testing practitioners". *Information and Software Technology*, 38(12):775–782, Elsevier B.V., February 1996.

[45] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. "Mirror adaptive random testing", In *Proceedings of the Third International Conference on Quality Software: QSIC 2003*, IEEE Computer Society, pages 1001–1010, Dallas, TX, USA, November 6-7 2003.

[46] K. Chan, T. Chen, and D. Towey. "Normalized restricted random testing". In Jean-Pierre Rosen and Alfred Strohmeier (editors), *Reliable Software Technologies — Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pages 368–381, 2003.

[47] K. Chan, T. Chen, and D. Towey. "Restricted random testing". In Jyrki Kontio and Reidar Conradi (editors), *Software Quality — ECSQ 2002*, volume 2349 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pages 321–330, 2002.

[48] T. Y. Chen and R. Merkel. "Quasi-random testing". *IEEE Transactions on Reliability*, 56(3):562–568, IEEE Reliability Society, September 2007.

[49] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. "Feedback-directed random test generation", In *Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, pages 75–84, Minneapolis, MN, USA, May 20-26 2007.

[50] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. "Artoo: adaptive random testing for object-oriented software", In *Proceedings of the 30th international conference on Software engineering*, ACM Press, pages 71–80, Leipzig, Germany, May 10-18 2008.

[51] T. Y. Chen, R. Merkel, P. K. Wong, and G. Eddy. "Adaptive random testing through dynamic partitioning", In *Proceedings of the Fourth International Conference on Quality Software*, IEEE Computer Society, pages 79–86, Braunschweig, Germany, September 8-9 2004.

[52] B. Meyer. *Object-Oriented Software Construction*. 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[53] B. Meyer. "Applying design by contract". *IEEE Computer*, 25(10):40–51, IEEE Computer Society Press, October 1992.

[54] B. Meyer. "Eiffel as a framework for verification". In Bertrand Meyer and Jim Woodcock (editors), *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pages 301–307, 2008.

[55] C. Csallner and Y. Smaragdakis. "Jcrasher: an automatic robustness tester for java". *Software: Practice and Experience*, 34(11):1025–1050, John Wiley & Sons, Inc., September 2004.

[56] C. Pacheco and M. D. Ernst. "Randoop: Feedback-directed random testing for java", In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ACM Press, pages 815–816, Montreal, Quebec, Canada, October 21-25 2007.

[57] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. "Bugbench: Benchmarks for evaluating bug detection tools", In *Workshop on the Evaluation of Software Defect Detection Tools*, ACM, pages 1–5, Chicago, IL, USA, June 12 2005.

[58] J. Slaby, J. Strejek, and M. Trtík. "Claburedb: Classified bug-reports database". In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (editors), *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pages 268–274, 2013.