



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Fajita: Generación automática de casos de test basada en verificación acotada

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Daniel Alfredo Ciolek

Directores: Frias, Marcelo

Galeotti, Juan Pablo

Buenos Aires, Marzo de 2012

Resumen

La generación de casos de test es una de las metodologías más utilizadas para verificar la calidad de un sistema de software. Si bien resulta más económica que otros acercamientos como la verificación formal, los costos de una validación exhaustiva siguen siendo muy elevados, por lo que en la mayoría de los casos se hace un balance entre el costo de afrontar las potenciales fallas y el esfuerzo destinado a la validación.

En esta tesis se presenta una técnica de generación automática de casos de test que puede ser utilizada para reducir los costos de la verificación del software. La técnica busca un conjunto minimal de casos de test que logre un alto índice de cobertura según un criterio dado. La metodología resulta flexible ya que admite la instrumentación de distintos criterios frecuentemente utilizados en la práctica.

La técnica presentada se basa en efectuar una verificación acotada siguiendo una especificación formal de un programa. La verificación acotada se hace mediante la traducción del programa junto con su especificación a un modelo formal. Luego, el modelo es analizado utilizando un SAT-Solver para obtener una traza de ejecución válida, resultado que es traducido a un caso test.

Como parte de este trabajo se construyó un prototipo experimental llamado Fajita. La herramienta fue evaluada mediante la comparación contra herramientas del estado del arte. Durante el trabajo se analizan las ventajas que introducen la utilización de SAT-Solving incremental junto con distintas tácticas para la reducción del espacio de búsqueda (predicados de rupturas de simetrías y exploración iterativa). Se proponen, también, mejoras en algunos de los criterios de selección de casos de test comúnmente utilizados.

Palabras claves: Testing, Generación automática de casos de test, Verificación acotada, SAT-Solving incremental, TACO, DynAlloy, Alloy.

Abstract

The generation of test cases is one of the most common software quality verification methodologies. While it tends to be more economical than other approaches such as formal verification, the costs of exhaustive validation still remain prohibitive; for that reason, most of the time a trade off between the costs of facing potential flaws and the effort destined to validation is made.

In this thesis a technique for automatic test case generation is presented, which can be used to reduce the costs of software verification. The technique searches a minimal test set that achieves a high coverage index for a given test selection criterion. The methodology turns out to be flexible since it admits the instrumentation of different criteria frequently used in practice.

The presented technique is based on making a bounded verification following a formal specification of a program. The bounded verification involves the translation of the program with its specification to a formal model. Then, the model is analyzed using a SAT-Solver in order to obtain a valid execution trace, which is finally translated to a test case.

As part of this work, an experimental prototype called Fajita was built. The tool was evaluated by comparing it with state of the art tools. The advantages introduced by the usage of incremental SAT-Solving and different search space reduction tactics (symmetry breaking predicates and iterative exploration) are analyzed in this text. Also, improvements for some test selection criteria of wide usage are proposed.

Keywords: Testing, Automatic test case generation, Bounded verification, Incremental SAT-Solving, TACO, DynAlloy, Alloy.

Índice general

Índice de Tablas	VII
Índice de Figuras	IX
1. Introducción	1
1.1. Testing	1
1.1.1. Testing de caja cerrada	3
1.1.2. Testing de caja abierta	3
1.2. Verificación formal	3
1.3. Terminología del testing	4
1.4. Calidad de un conjunto de casos de test	6
1.4.1. Cobertura de particiones	6
1.4.2. Cobertura de objetivos	7
1.4.3. Cobertura de sentencias	7
1.4.4. Cobertura de decisiones	8
1.4.5. Cobertura de caminos	9
1.4.6. Otros criterios	10
1.5. Objetivos	10
2. Preliminares	13
2.1. TACO	16
2.2. Alloy	17
2.3. Verificación acotada	18
2.4. Generación de casos de test	19
2.5. Innovaciones de TACO	20

3. Desarrollo	25
3.1. Generación de casos de test usando SAT-Solving incremental	26
3.2. Algoritmo de cobertura de particiones	26
3.3. Generación incremental de casos de test de caja cerrada	28
3.3.1. Cobertura de métodos booleanos sin restricciones	31
3.4. Generación incremental de casos de test de caja abierta	32
3.4.1. Cobertura de objetivos	33
3.4.2. Cobertura de sentencias	36
3.4.3. Cobertura de decisiones	37
3.5. Cubrimiento de caja gris	39
3.6. Exploración iterativa del espacio de estados	40
3.7. Especificaciones parciales	42
4. Evaluación	45
4.1. Comparación de herramientas	45
4.1.1. ROOPS	45
4.1.2. Herramientas	45
4.1.3. Configuración	46
4.1.4. Formato de los resultados	46
4.1.5. Caso de estudio: Operaciones sobre tipos primitivos	46
4.1.6. Caso de estudio: Estructuras de datos	49
4.2. Cobertura de caja gris	56
5. Conclusiones	61
Bibliografía	66

Índice de Tablas

4.1. Cobertura de objetivos de la clase <code>LinearWithoutOverflow</code>	48
4.2. Cobertura de objetivos de la clase <code>LongBenchmark</code>	48
4.3. Cobertura de objetivos de la clase <code>FloatBenchmark</code>	48
4.4. Cobertura de objetivos de la clase <code>IntArrayWithoutExceptions</code>	48
4.5. Resultados sobre el ROOPS benchmark usando exploración iterativa.	51
4.6. Resultados obtenidos para el caso de estudio de estructuras de datos.	53
4.7. Cobertura de métodos booleanos	57
4.8. Decisiones cubiertas por la iteración de caja cerrada	58

Índice de Figuras

2.1. Arquitectura de Fajita.	15
2.2. Modelo Alloy para números enteros de 32 bits.	18
2.3. Predicado de igualdad de números enteros de 32 bits.	18
2.4. Listas enlazadas de longitud 3 simétricas respecto a la relación next.	21
2.5. Lista enlazada de longitud 3 distinguida.	21
2.6. Relaciones eliminadas mediante ruptura de simetrías.	22
2.7. Arquitectura de TACO extendida con el repositorio de cotas.	23
3.1. Lista enlazada con métodos booleanos sin argumentos insuficientes.	31
3.2. Ejemplo de transformación con métodos booleanos sin restricciones.	33
3.3. Ejemplo de instrumentación del criterio de objetivos.	34
3.4. Esquematización de la instrumentación del criterio de decisiones sobre grafos de control de flujo de estructuras de control comunes.	38
4.1. Ejemplo de evaluación de condiciones con expresiones de punto flotante.	47
4.2. Ejemplo de evaluación de condiciones con aritmética entera de 64 bits.	47
4.3. Cobertura de objetivos (1/2)	54
4.4. Cobertura de objetivos (2/2)	54
4.5. Cobertura de decisiones (1/2)	55
4.6. Cobertura de decisiones (2/2)	55

Capítulo 1

Introducción

En la actualidad la actividad humana está continuamente acompañada por la utilización de software. Los mismos dispositivos que en décadas pasadas representaron la introducción de la electrónica en la vida diaria, como la televisión o el teléfono, hoy poseen comportamientos más complejos regidos por programas. Al mismo tiempo, las grandes redes de telecomunicaciones y sistemas de información, como las redes bancarias, también se han vuelto dependientes de distintas piezas de software.

El avance del software ha llegado a áreas críticas como el control del espacio aéreo o dispositivos médicos. Sin embargo, las fallas de estos sistemas son comunes y costosas. Distintas metodologías de verificación de calidad han surgido a lo largo del desarrollo de la disciplina, como la verificación formal y el testing. Estas metodologías trasladan el costo de afrontar las fallas al proceso de desarrollo, marginándolas al mercado de aplicaciones de áreas críticas.

1.1. Testing

El testing de software es el proceso de ejecutar un programa con la intención de encontrar errores o evaluar si un sistema cumple determinados requerimientos [1]. Es decir, el propósito de la ejecución de casos de test puede ser tanto la verificación de correctitud como la obtención de métricas de performance u otros atributos de calidad. Llamamos casos de test a los datos construidos con el fin de ser utilizados como parámetros del software siendo analizado.

El testing requiere un balance entre presupuesto y calidad. A medida que aumenta la necesidad de asegurar la calidad de un sistema es necesario aumentar los recursos destinados a intentar lograrlo.

En esta tesis se presenta Fajita, una herramienta de generación automática de casos de test orientada a la verificación del software. La generación automática de casos de test promete mejorar el balance entre la calidad del software y los recursos invertidos en su validación.

Según su alcance los tests pueden ser categorizados como:

- Tests de unidad: la ejecución del test afecta a una única unidad lógica o módulo.
- Tests de integración: evalúan a múltiples módulos con estrecha colaboración.
- Tests de sistema: la ejecución del test afecta a todo el sistema.

Esta separación se debe a que cada categoría es relevante en un estado distinto del ciclo de vida del desarrollo de un sistema. Los tests de unidad pueden usarse para detectar errores de forma temprana durante el desarrollo y para validar las especificaciones. Los tests de integración sirven para detectar inconsistencias en la interacción de los distintos módulos, frecuentemente producto de una falla en la comunicación entre los programadores involucrados. Mientras que los tests de sistema se utilizan para validar el sistema en su conjunto y obtener distintas métricas.

A pesar de conocer las ventajas de escribir tests de unidad se observa una tendencia a generar una cantidad mínima e insuficiente. Los estudios realizados [2] llegan a la conclusión de que el esfuerzo requerido para el desarrollo y mantenimiento de estos tests resulta, en apariencia, prohibitivo.

La generación automática de casos de test es actualmente un área de investigación prolífica donde varias técnicas y herramientas han sido desarrolladas recientemente. Entre los acercamientos más exitosos pueden citarse a las herramientas basadas en generación aleatoria [3, 4], verificación de modelos [5, 6], resolución de restricciones mediante SAT-Solving o SMT-Solving [7] y otras de búsqueda exhaustiva [8]. Llamamos SAT-Solver a un algoritmo que busca una valuación que satisfaga una fórmula de lógica proposicional. SMT-Solving es una generalización del procedimiento realizado por un SAT-Solver que analiza la satisfacibilidad de una fórmula de primer orden donde algunos símbolos admiten interpretaciones adicionales.

Por sus características Fajita se encuentra en el grupo de las herramientas basadas en resolución de restricciones y para ello utiliza SAT-Solving. Fajita se centra en la generación de casos de test de unidad de programas Java, pero es posible aplicar la misma técnica en escenarios de mayor tamaño, tales como tests de interacción o de sistema. Sin embargo, para poder lograr un buen desempeño en estos casos todavía debe encontrarse una forma de lidiar con el crecimiento del costo computacional que los caracteriza.

A lo largo de este trabajo se introducen los conceptos de testing con los que trabaja Fajita (capítulo 1) y se presentan las bases teóricas y las herramientas gracias a las cuales es posible su implementación (capítulo 2). Luego se explican los detalles de la técnica desarrollada (capítulo 3) y se comparan los resultados obtenidos al evaluar Fajita junto a otras herramientas del estado del arte (capítulo 4). Finalmente se analizan los resultados y se sacan conclusiones (capítulo 5).

1.1.1. Testing de caja cerrada

Un acercamiento clásico a la generación de casos de test es el conocido como de caja cerrada, que requiere generar los datos de entrada a partir de la especificación de los requerimientos funcionales sin contemplar la estructura del código [9]. En la literatura esta estrategia puede encontrarse con distintos nombres: de caja negra, dirigida por entrada/salida [1] o basada en requerimientos [10]. Dado que sólo la funcionalidad del software es de importancia, el testing de caja cerrada también es conocido como testing funcional [11].

Al ejecutar los tests el software se ve como una caja negra sólo permitiendo visualizar los valores de entrada y de salida. La funcionalidad es determinada observando las salidas obtenidas a partir de los distintos parámetros de entrada. El proceso requiere utilizar múltiples entradas y comparar los resultados obtenidos contra una especificación en pos de verificar la correctitud del objeto bajo test.

Es evidente que a mayor cobertura del espacio de entrada, mayor es la cantidad de errores que pueden ser encontrados, aumentando así la confianza en la calidad del software. Por esta razón resulta tentador recorrer de forma exhaustiva el espacio de entrada. Sin embargo, esto resulta imposible para la mayoría de los programas, pues se produce una explosión combinatoria de las posibles entradas. Por consiguiente es necesario hacer una exploración inteligente del dominio, de forma tal de obtener una cantidad acotada de datos que logre ejercitar los distintos aspectos de la funcionalidad analizada.

1.1.2. Testing de caja abierta

A diferencia del testing de caja cerrada, en testing de caja abierta las estructuras de control del software a ser probado son visibles, por lo que los casos de test son generados en base a un conocimiento preciso de los detalles de la implementación. El testing de caja abierta también es llamado testing de caja blanca, de caja vidrio o dirigido por diseño [10].

Las técnicas de testing de caja abierta suelen traducir la estructura de control de flujo de un programa en un grafo dirigido, con el objetivo de construir parámetros de entrada que desencadenen ejecuciones que cubran de forma exhaustiva el grafo. El problema intratable de la generación de entradas es combatido con el conocimiento obtenido a partir de la inspección del código, utilizándolo para guiar la exploración del espacio de entradas más eficientemente.

1.2. Verificación formal

La utilización de métodos formales para probar la correctitud del software también representa una dirección de investigación atractiva, por ejemplo usando PVS o Isabelle [12]. Sin embargo, las técnicas actuales tampoco logran superar la complejidad del problema. Para programas sencillos estos métodos logran resultados satisfactorios, pero no escalan con la complejidad de grandes sistemas.

Si bien la debilidad del testing es bien conocida por la comunidad que trabaja en métodos formales: “los tests sólo pueden mostrar la presencia de errores pero no asegurar su ausencia” [13]; el testing resulta más económico especialmente durante el proceso de desarrollo, permitiendo avanzar en la construcción del sistema con cierta confianza en la calidad obtenida. El costo de utilizar métodos formales puede ser aceptable en las secciones del sistema que requieran un mayor grado de confiabilidad. Por este motivo el testing y la verificación formal no son rivales, sino que son herramientas complementarias a la hora de perseguir el objetivo de aumentar la confianza en la calidad del software a un costo aceptable.

Fajita se basa en herramientas de verificación formal “livianas”, que en lugar de efectuar una validación exhaustiva sobre todo el dominio, realizan un procedimiento de verificación acotada. La verificación acotada [14] es una técnica en la que todas las posibles ejecuciones de un procedimiento son exhaustivamente examinadas dentro de un espacio finito dado por una cota para el tamaño de la memoria utilizada y un número máximo de iteraciones para los ciclos (llamado scope de análisis). Dentro de este espacio se busca una traza de ejecución que viole una especificación dada en un lenguaje formal.

Muchas herramientas de verificación acotada se basan en traducir el código y la especificación a verificar a una fórmula proposicional y luego utilizar un SAT-Solver [15, 16, 17, 18] para encontrar una valuación que codifique un error. Si una violación es encontrada dentro del scope de análisis, una traza de ejecución exponiendo el error es exhibida al usuario. Es bien sabido que SAT-Solving es un problema NP-Completo [19], por lo que las implementaciones disponibles requieren para su resolución un tiempo exponencial con respecto a la cantidad de variables de la fórmula proposicional tratada. Si bien es cierto que en general estas implementaciones son exponenciales, en la práctica al trabajar con las fórmulas obtenidas a partir de traducciones de código se logran resultados mucho mejores.

Dado que Fajita trabaja sobre código Java y utiliza herramientas que requieren especificaciones formales, resulta natural la elección de JML (Java Modeling Language) [20] como lenguaje de especificación. JML es un lenguaje de especificación inspirado en la metodología de programación de diseño por contratos. JML provee anotaciones que permiten a los programadores definir contratos (usando estructuras como pre y pos-condiciones) e invariantes de representación. La principal ventaja de usar un lenguaje de especificación es que permite expresar invariantes de representación de estructuras complejas con un paradigma declarativo, a diferencia del acercamiento operacional que requiere expresar estas propiedades con el mismo lenguaje de programación.

1.3. Terminología del testing

En esta sección se introducen de forma precisa los términos de la disciplina del testing que son utilizados en este trabajo. Lamentablemente la terminología no está estandarizada por lo que puede encontrarse en la literatura los mismos términos utilizados con distintos significados.

Sean P un programa, D el conjunto de todos los datos que pueden ser introducidos como parámetros de P y R el conjunto de todos los resultados que pueden ser obtenidos a partir de la ejecución de P . La notación $P(d)$ denota el resultado de ejecutar el programa P con parámetros d , con $d \in D$ y $P(d) \in R$.

La especificación de un programa permite decidir si el resultado obtenido tras su ejecución es correcto o no. Se dice que P es correcto para un $d \in D$ dado, si $P(d)$ satisface la especificación del programa S_P . Luego P es correcto si es correcto para todo $d \in D$.

La presencia de un error es detectada al mostrar que $P(d)$ no satisface S_P para algún $d \in D$. Se llama falla a la manifestación de un error tras la ejecución de un programa. La presencia de un error no necesariamente genera una falla, por lo que el testing intenta aumentar la probabilidad de que los errores se manifiesten seleccionando casos de test apropiadamente. Dado que D es un conjunto potencialmente infinito la evaluación exhaustiva mediante la verificación de que $P(d)$ satisfaga S_P para todo $d \in D$, no es posible en general.

En estos términos un caso de test es un elemento $d \in D$. Un test suite T es un conjunto finito de casos de test, es decir, T es un subconjunto finito de D . Se dice que un programa P es correcto para un conjunto de casos de test T si es correcto para todos los elementos $d \in T$. Un test suite T se dice ideal si, siempre que exista un error en P , también existe un caso de test $d \in T$ que evidencia una falla.

Un criterio de selección de casos de test C es un subconjunto de 2^D (que denota el conjunto de todos los subconjuntos finitos de D), o en otras palabras, C denota una condición que debe cumplir un test suite. Un criterio C se dice consistente si para cualquier par de test suites T_1 y T_2 que satisfacen C , sucede que P es correcto para T_1 si y sólo si P es correcto para T_2 . Se llama completo a un criterio C que requiere la existencia de al menos un elemento d en un test suite, para cada error en un programa P tal que $P(d)$ evidencia la falla producida por el error. Por último, se dice que un criterio C es insatisfacible si representa el conjunto vacío, es decir, que no exista ningún test suite que pertenezca a C .

Por lo tanto, basta encontrar un test suite cualquiera que satisfaga un criterio consistente y completo para demostrar la correctitud de un programa [21]. Lamentablemente determinar si un conjunto de casos de test es ideal, o si un criterio de selección es completo, consistente o insatisfacible; constituyen problemas indecidibles al igual que determinar la correctitud, terminación o equivalencia de programas.

Sin embargo, a pesar de ser potencialmente insatisfacibles, algunos criterios son indicadores de la calidad o suficiencia de un test suite, por lo que han sido estudiados y utilizados exhaustivamente en la industria. Llamamos índice de cobertura al porcentaje de obligaciones, impuestas por un criterio, que son satisfechas por un test suite.

Fajita incorpora los criterios más comúnmente utilizados para generar un conjunto de casos de test minimal que maximice el índice de cobertura de un criterio dado. Es decir, se busca un test suite para el cual remover un caso de test del test suite disminuye el índice de cobertura logrado. En la sección siguiente se introducen los distintos criterios utilizados.

1.4. Calidad de un conjunto de casos de test

Existen múltiples criterios para determinar la calidad o suficiencia de un test suite, cada uno con sus fortalezas y debilidades. En general estos criterios se basan en el índice de cobertura sobre algún aspecto del código o su especificación, que se logra tras la ejecución de un conjunto de tests. A continuación se listan algunos de los criterios más comunes describiendo sus características principales. Fajita incorpora estos criterios permitiendo generar conjuntos de casos de test que maximicen la cobertura en cada caso, lo que lo habilita a variar la estrategia de generación de tests dependiendo de las características del código siendo manipulado.

1.4.1. Cobertura de particiones

La investigación en testing de caja cerrada se centra principalmente en cómo maximizar la adecuación de un conjunto de casos de test minimizando, en general, el número de tests. Si bien no es posible cubrir exhaustivamente el espacio de entrada, sí es posible hacerlo sobre un subconjunto. La división en clases de equivalencia es una de las técnicas más comunes para identificar un subconjunto representativo. Si se tiene una partición del espacio de entrada tal que los valores en cada partición son considerados equivalentes, entonces basta crear tests con un representante de cada partición para cubrirlo satisfactoriamente [22].

La motivación detrás de este acercamiento es que se observa empíricamente que los errores no están uniformemente distribuidos. Por lo que la separación en clases de equivalencia intenta segmentar el espacio de entradas según la densidad de potenciales errores que pueden ser encontrados mediante su inspección. Es decir, es deseable evitar generar casos de test similares que pueden conformar grandes áreas del espacio de entradas y en su lugar emplear el esfuerzo en encontrar casos de test significativamente distintos entre sí.

Lograr una buena división requiere conocimiento sobre la estructura del software. Fajita implementa una estrategia de segmentación llamada cobertura de métodos booleanos introducida por Bertrand Meyer y otros en [4] basada en la siguiente afirmación:

“Los métodos booleanos sin argumentos de una clase bien diseñada inducen una partición del espacio de estados de los objetos que ayuda a aumentar la efectividad de las estrategias de generación de casos de test.”

De esta forma basta el acceso al prototipo de una clase y a su especificación para poder generar casos de test. Dado que no es necesario inspeccionar el código este acercamiento entra en la categoría de testing de caja cerrada.

1.4.2. Cobertura de objetivos

La cobertura de objetivos mide la cantidad de objetivos alcanzados por una batería de casos de test. Cada objetivo es una anotación especial en el código sin impacto en la funcionalidad, que se introduce con el único fin de marcar un punto en la estructura de control de flujo. Se dice que un objetivo es cubierto por un test si la ejecución resultante genera una traza que atraviesa la anotación. Como los objetivos forman parte del código este es un criterio de caja abierta.

Dado que los objetivos son insertados manualmente el índice de cobertura reportado es significativo únicamente al operador. Por lo que de este criterio no puede deducirse la calidad de un programa de forma general, sino que se utiliza principalmente para corroborar que los tests ejerciten una funcionalidad particular. Por ejemplo, podría ser de interés verificar que en ciertos puntos de un programa se estén manejando correctamente las excepciones de entrada/salida. Para ello puede agregarse de forma manual un objetivo en los bloques de captura de excepciones y así verificar que los tests lo cubren.

```
try {
    FileReader f = new FileReader("prueba.txt");
    f.read();
} catch (IOException e) {
    @goal(1)
}
```

Si bien este criterio no es útil a la hora de generar casos de test de forma automática, ya que no fija un criterio claro, resulta sumamente útil para comparar distintos conjuntos de tests. Se puede anotar un programa con objetivos relevantes para luego corroborar qué conjunto logra la mayor cobertura. Esto es utilizado en el capítulo 4 para comparar los resultados obtenidos por Fajita con los obtenidos por otras herramientas.

1.4.3. Cobertura de sentencias

Esta métrica reporta qué porcentaje de sentencias ejecutables se ejercitan al ejecutar un conjunto de casos de test. Las sentencias de control de flujo, como los `if`, `for` y `switch` son cubiertas si la expresión que controla la estructura es ejecutada. También es conocida como: cubrimiento de líneas o instrucciones. La cobertura de bloques o segmentos es equivalente a la de sentencias pero la unidad de código medida es cada secuencia de instrucciones consecutivas sin estructuras de control.

Intuitivamente este criterio se basa en el hecho de que un error no puede ser descubierto si no se ejecuta la porción del programa que evidencia la falla. Por lo que se busca generar tests que ejerciten todas las sentencias al menos una vez. Llamamos código muerto a las sentencias para las que no existe ninguna entrada válida que las ejecute, muchas veces producto de prácticas de programación defensivas. Resulta evidente que para los programas que contengan código muerto el criterio de cobertura de sentencias es insatisfacible.

La principal ventaja de este criterio es que puede ser implementado sobre código objeto sin necesidad de procesar el código fuente, por lo que normalmente es incluido en diversas herramientas de análisis de programas. La principal desventaja es que no hace distinción en la cobertura de las estructuras de control, con lo que se ignoran muchos posibles comportamientos. Por ejemplo, consideremos el siguiente fragmento de código:

```
String s = null;
if (condicion)
    s = "Verdadero";
s = s.trim();
```

Sin un caso de test en el que la condición se evalúe a falso el cubrimiento de sentencias puede reportar un cubrimiento total del fragmento. Sin embargo, si la condición evaluara a falso la ejecución fallaría al llegar a la última línea, ya que se intentaría enviar un mensaje al objeto nulo.

1.4.4. Cobertura de decisiones

Esta métrica reporta el porcentaje de expresiones booleanas dentro de estructuras de control que evalúan tanto a verdadero como a falso. Este criterio también es conocido como cobertura de ramas o ejes.

Si bien este criterio evita algunos problemas del cubrimiento de sentencias, también puede ignorar errores que se evidencien únicamente al ejercitar una combinación específica de ramas. Por ejemplo:

```
String s1 = null;
String s2 = null;
if (condicion1)
    s1 = "Verdadero";
else
    s2 = "Falso";
if (condicion2)
    s1 = s1.trim();
else
    s2 = s2.trim();
```

En este caso se puede ver que un test suite formado por un caso que evalúa ambas condiciones a verdadero y otro que evalúa ambas condiciones a falso logra una cobertura de decisiones total sin encontrar ningún problema. Sin embargo, es fácil ver que un tercer caso que haga verdadera la primera condición y falsa la segunda encontraría una falla.

En principio, este acercamiento no considera el comportamiento de corto circuito de los operadores lógicos por lo que puede ignorar, además, otros defectos. Por ejemplo:

```
String s = null;
if (condicion1 && (condicion2 || s.isEmpty()))
    s = "Verdadero";
else
    s = "Falso";
```

La métrica podría considerar el fragmento completamente cubierto sin nunca haber ejercitado la llamada a `isEmpty()`, que al ser `s` una referencia al objeto nulo evidenciaría una falla. El criterio conocido como cobertura de condiciones requiere generar casos de test que evalúen cada componente en las condiciones de las estructuras de control tanto por verdadero como por falso. Un acercamiento común es la combinación de ambos criterios, lo que puede lograrse simplemente realizando una transformación sobre el código fuente que reemplace los operadores lógicos con estructuras de control. Por ejemplo el código anterior debería transformarse a:

```
String s = null;
if (condicion1)
    if (condicion2)
        s = "Verdadero";
    else if (s.isEmpty())
        s = "Verdadero";
    else
        s = "Falso";
else
    s = "Falso";
```

Este acercamiento resuelve el problema mencionado pero resulta demasiado engorroso para aplicarlo en un proceso manual, por lo que en la práctica sólo una herramienta que realice estas traducciones de forma automática la aprovecharía.

1.4.5. Cobertura de caminos

Este criterio reporta el porcentaje de caminos cubiertos por la ejecución de un conjunto de casos de test. Donde un camino es una secuencia única de ramificaciones desde el punto de entrada a un punto de salida.

Si bien este acercamiento resuelve el problema del cubrimiento de decisiones posee numerosos problemas en sí mismo. Primero, dado que los ciclos introducen una cantidad no acotada de caminos debe ponerse un límite al número de posibilidades. Segundo, la cantidad de caminos es exponencial con respecto al número de ramificaciones. Tercero, muchos caminos son imposibles de ejercitar, por ejemplo:

```
if (condicion1)
    s = "Verdadero";
else
    s = "Falso";
if (condicion1)
    s += " True";
else
    s += " False";
```

La métrica considera cuatro caminos potenciales para este ejemplo, pero en realidad sólo dos pueden ser cubiertos, por lo que en el mejor de los casos sólo puede reportarse un cubrimiento del 50%. Si bien los casos anteriores también pueden resultar insatisfacibles, en esos casos la insatisfacibilidad indica necesariamente la presencia de código muerto, mientras que en este caso no. Por lo que además de ser computacionalmente costosa, la salida de esta métrica no permite hacer una evaluación concienzuda sobre la calidad del código ni del test suite, aún cuando este sea completamente exhaustivo.

Este criterio resulta de interés únicamente teórico, ya que por sus características no resulta práctico. Sin embargo, los esfuerzos realizados en generar nuevos criterios apuntan, en muchos casos, a mejorar criterios como el de decisiones para contemplar escenarios adicionales considerados por esta métrica.

1.4.6. Otros criterios

Muchos otros criterios han sido elaborados con el objetivo de combatir los problemas mencionados y atacar problemas puntuales de dominios específicos. Por el momento Fajita sólo implementa los criterios presentados anteriormente pues conforman las recomendaciones de la literatura clásica del testing [1, 10, 23], exceptuando la cobertura de caminos ya que como fue mencionado no es de interés práctico.

Entre los criterios que no son contemplados por Fajita están los utilizados para detectar fallas en la sincronización de recursos en aplicaciones con múltiples hilos de ejecución. Actualmente Fajita sólo admite programas secuenciales, pero existe la posibilidad de que la técnica pueda extenderse a programas concurrentes.

Como trabajo futuro podría agregarse en Fajita los criterios de MC/DC (Modified condition/decision) y LCSAJ (Linear code sequence and jumps) [23], que son refinamientos de los criterios de cobertura de decisiones y condiciones. De forma similar podrían considerarse otros criterios como all-uses coverage, que requiere ejercitar cada camino entre la asignación de un valor a una variable y su uso posterior.

1.5. Objetivos

El prototipo presentado en esta tesis, es una herramienta de generación de casos de test concebida originalmente para testing de caja cerrada, con el objetivo de alcanzar una alta cobertura de particiones. La herramienta fue extendida para generar, también, casos de test según criterios de caja abierta.

Fajita emplea una especificación formal y un conjunto de predicados para construir entradas que satisfagan la especificación. Idealmente cada valuación factible de los predicados caracteriza una clase de equivalencia no vacía de la entrada, por lo que es deseable generar un caso representativo de cada una.

Una ejecución de Fajita que posea k predicados considera a lo sumo 2^k clases. En la práctica muchas combinaciones no son posibles, por ejemplo si p_1 es $\neg p_2$ la valuación

donde ambos son verdaderos es imposible. En el presente trabajo se muestra cómo hacer la elección de los predicados en pos de perseguir distintos criterios de cobertura, tanto con un acercamiento de caja cerrada, al considerar sólo la interfaz del objeto manipulado, como de caja abierta al considerar también detalles de la implementación.

Los objetivos de este trabajo pueden resumirse como:

1. Idear una técnica de generación de casos de test flexible que permita perseguir distintos criterios de cobertura.
2. Sintetizar eficientemente datos que conformen casos de test que logren un alto índice de cobertura según el criterio elegido.

En este contexto eficiencia se entiende como mayor índice de cobertura y menor tiempo de ejecución que otras herramientas del estado del arte.

Capítulo 2

Preliminares

Como ya fue mencionado en la sección 1.2, Fajita se basa en herramientas de verificación formal livianas, que en lugar de hacer una validación exhaustiva sobre todo el dominio, realizan un procedimiento de verificación acotada. La implementación actual trabaja haciendo un uso intensivo de las herramientas TACO [24] y Alloy [25]. Este capítulo introduce estas herramientas explicando brevemente su funcionamiento y el rol que cumplen dentro la arquitectura de Fajita. Se mencionan, también, las características e innovaciones de estas herramientas que impactan sobre el desempeño de Fajita.

La forma de trabajo de Fajita está esquematizada en la Figura 2.1 y puede resumirse de la siguiente manera:

1. Se toma como entrada código fuente Java con anotaciones JML y se aplican traducciones para instrumentar la técnica en pos de perseguir un criterio de selección de casos de test dado. Este paso representa uno de los principales aportes de este trabajo y será explicado en detalle en el capítulo 3. De forma simplificada puede entenderse como la elección automática de predicados y restricciones para particionar el espacio de entradas, con el fin de poder explorarlo en busca de casos de test.
2. Se utiliza el resultado de la instrumentación como entrada de TACO, que traduce código y especificaciones a un modelo formal con el fin de poder utilizar la técnica de verificación acotada mencionada en la sección 1.2. Esta traducción requiere limitar el modelo generado a un scope de análisis determinado, en principio, por el usuario. La salida de TACO es un modelo formal expresado en el lenguaje de modelado Alloy, que puede ser analizado con un SAT-Solver gracias a que está acotado a un scope de análisis finito. Más detalle sobre el funcionamiento de TACO es dado en la sección 2.1 y sobre Alloy en la sección 2.2.
3. Se modifica el modelo Alloy generado por TACO para incluir restricciones adicionales que permiten aprovechar las características de los SAT-Solvers incrementales utilizados en la verificación acotada. Estas modificaciones se basan, en general, en agregar axiomas que fuerzan la generación de valuaciones que representen soluciones con las características deseadas.

4. Se utiliza el modelo actualizado como entrada de la herramienta Alloy (nótese que Alloy es tanto un lenguaje de especificación como una herramienta de análisis) que ofrece una interfaz común a múltiples SAT-Solver incrementales. La herramienta Alloy retorna una valuación que satisface el modelo del programa o un mensaje de que ninguna valuación puede encontrarse con las restricciones actuales (UNSAT). El mensaje de UNSAT indica que no pueden encontrarse más soluciones dentro del scope de análisis elegido, esto es distinto a que el criterio de selección elegido sea insatisfacible, pues lo segundo requiere que no puedan encontrarse las soluciones en el caso general, o equivalentemente con un scope de análisis infinito.
5. Para cada solución retornada por la herramienta Alloy, Fajita construye un caso de test (paso explicado en la sección 2.4) y repite el procedimiento de verificación acotada agregando restricciones de manera incremental de forma tal que no se repita la misma solución, volviendo así al paso 3. Las ventajas de la utilización de SAT-Solvers incrementales son discutidas en la sección 3.1.

Como puede apreciarse Fajita requiere como entrada los límites para el scope de análisis además del código a ser analizado. Sin embargo, es posible aumentar el grado de automatización de la herramienta generando los valores para el scope de análisis de forma automática. Las consideraciones necesarias para lograrlo son explicadas en la sección 3.6.

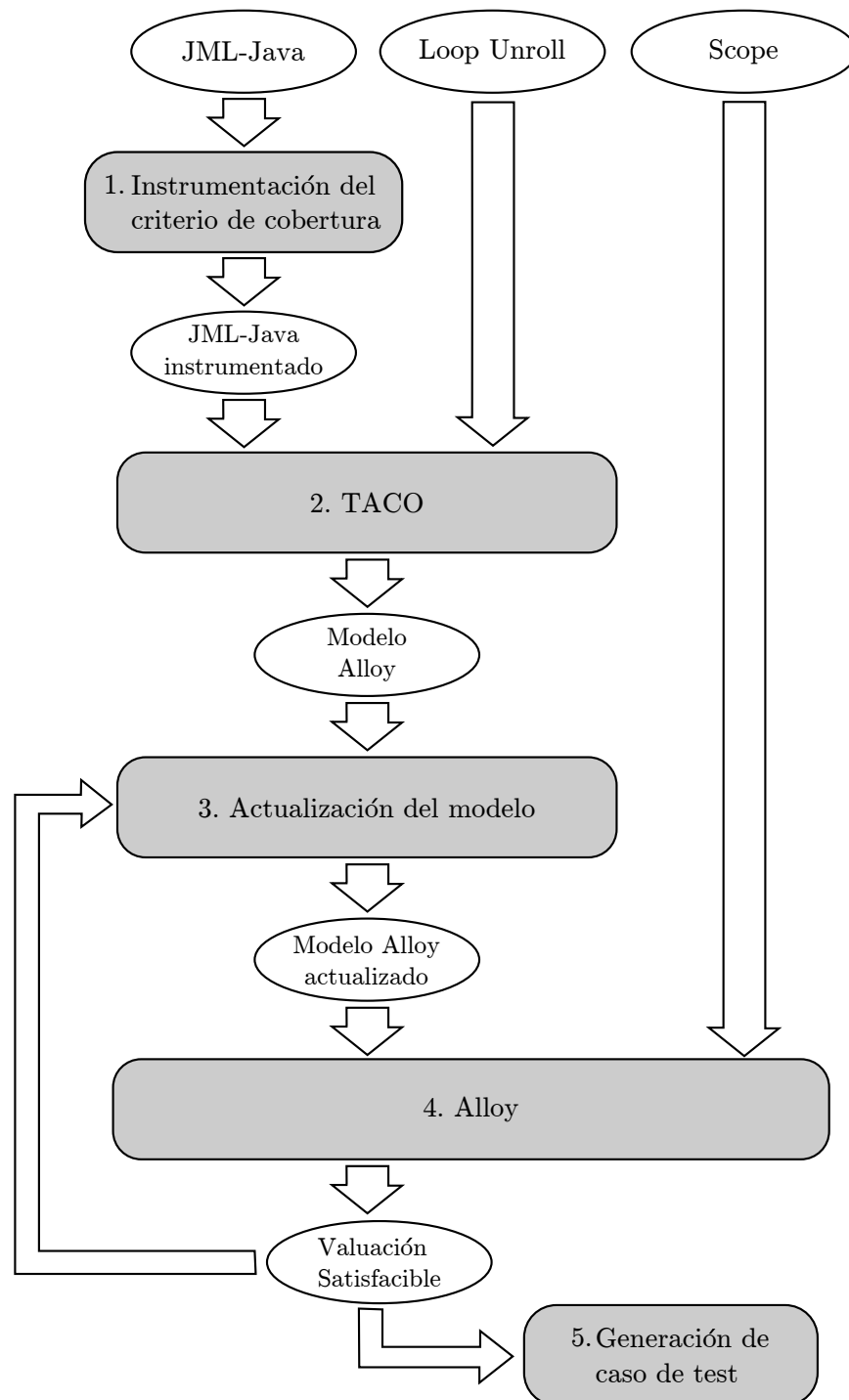


Figura 2.1: Arquitectura de Fajita.

2.1. TACO

TACO (Translation of Annotated COde) es una herramienta que traduce código fuente Java anotado con una especificación JML en un problema SAT. TACO usa Alloy como lenguaje intermedio para expresar el problema SAT, la elección se debe a que este lenguaje es una representación cercana a JML que simplifica el acceso a múltiples SAT-Solvers.

Para lograr la conversión entre los fuentes Java y el modelo Alloy, TACO utiliza la herramienta DynAlloy [26], una extensión de Alloy que permite especificar el comportamiento de acciones Java con efectos sobre el estado de un programa. La motivación detrás de la extensión es poder brindar un entorno en el que sea posible capturar las relaciones entre los datos de entrada y salida de un programa. Es necesario utilizar DynAlloy ya que Alloy no posee una sintaxis que permita expresar el comportamiento iterativo y dinámico del software. El compilador DynAlloy es capaz de generar un modelo Alloy limitando los aspectos dinámicos del software dentro del scope de análisis elegido. DynAlloy está inspirado en el lenguaje “*dynamic logic*” [27].

El funcionamiento normal de TACO puede resumirse en los siguientes pasos:

1. Dado un código fuente Java anotado con especificaciones JML, éste es traducido a un modelo DynAlloy. Para ello TACO internamente utiliza la herramienta DynJML [24] que es un traductor de sintaxis Java con anotaciones JML a un modelo DynAlloy.
2. El compilador DynAlloy realiza una traducción del resultado anterior a un modelo Alloy preservando la semántica del programa original. Para poder manejar los ciclos se restringe el tamaño máximo de las trazas de ejecución que se desea considerar en el análisis. Ésto se logra usando la técnica de loop unroll, que se basa en replicar el código en el cuerpo de los ciclos una cantidad finita de veces (parámetro loop unroll en la Figura 2.1). Por lo que el análisis sólo es capaz de encontrar errores en ejecuciones que realicen a lo sumo esa cantidad de iteraciones.
3. Finalmente, el modelo Alloy es traducido a una fórmula SAT utilizando la herramienta Alloy. Para poder construir una fórmula proposicional finita, TACO impone cotas para los distintos dominios, lo que representa una restricción en la precisión del análisis (parámetro scope de la Figura 2.1). Basta incrementar estas cotas para lograr analizar una mayor porción del conjunto de ejecuciones del programa.

Si bien el comportamiento normal de TACO incluye la ejecución de un SAT-Solver utilizando la interfaz provista por Alloy, Fajita interrumpe el proceso después de obtener el modelo Alloy. Esto se debe a que Fajita realiza unas alteraciones menores sobre el modelo y luego invoca directamente a Alloy para poder explotar las características del SAT-Solver incremental subyacente.

El funcionamiento de TACO está inspirado en los proyectos Miniatur [17] y JForge [28]. Pero además de dirigir el proceso introduce optimizaciones novedosas que impactan en el desempeño de Fajita por lo que son discutidas en la sección 2.5.

2.2. Alloy

Alloy es un lenguaje de especificación formal y una herramienta de análisis que pertenece a la clase llamada métodos formales orientados a modelos. El lenguaje Alloy está definido en términos de una semántica relacional simple, su sintaxis incluye construcciones comunes a los lenguajes con orientación a objetos. Alloy tiene sus raíces en el lenguaje de especificación Z [29], pero en contraste con Z, Alloy fue diseñado con el objetivo de hacer las especificaciones automáticamente analizables, por lo que la herramienta Alloy provee acceso a distintos SAT-Solver modernos, como MiniSat [30], ZChaff [31] y Berkmin [32].

Alloy utiliza Kodkod [33] como representación intermedia. Esta representación permite aplicar algunas optimizaciones que son mencionadas en la sección 2.5. Luego de la manipulación de la representación en Kodkod se traduce el modelo a forma normal conjuntiva (CNF) para ser procesado por el SAT-Solver.

La representación de sistemas en Alloy está basada en modelos abstractos, que están definidos, esencialmente, en términos de dominios de datos y operaciones entre estos dominios. En particular, se pueden utilizar dominios para especificar el espacio de estados de un sistema y emplear operaciones para especificar un cambio en el estado del sistema. La semántica de Alloy no refleja las nociones de salida y precedencia temporal, por lo que estos conceptos deben ser introducidos en el modelo como parte de la especificación. Es decir, cada modelo Alloy debe contar con predicados que modelen el funcionamiento de las construcciones del lenguaje, como por ejemplo las excepciones. Por este motivo los modelos generados por TACO a partir de programas Java resultan extensos y de difícil lectura, pero modelan su comportamiento de forma completa.

Por consiguiente las trazas de ejecución pueden ser definidas como la composición de los estados intermedios de ejecuciones específicas. Así, agregando trazas a un modelo, se obtiene un mecanismo para analizar ejecuciones de forma automática.

La sintaxis de Alloy permite definir conjuntos de átomos (`sig`), predicados (`pred`), axiomas y otras construcciones para representar las distintas propiedades de los modelos. Las firmas definen un tipo de datos nuevo que puede ser utilizado para declarar el tipo de una variable (`var : tipo`). Las variables pueden ser utilizadas como parámetros o atributos en las distintas construcciones. Al ser un lenguaje relacional provee un soporte primitivo de conjuntos de relaciones. Provee, también, la capacidad de realizar operaciones de clausura reflexiva y reflexo-transitiva y brinda la capacidad de hacer afirmaciones utilizando cuantificadores de primer orden.

Cabe destacar que los elementos base de la construcción son valores booleanos, por lo que por ejemplo los enteros de 32 bits son representados mediante 32 valores de verdad (Fig 2.2). Las operaciones son representadas con predicados que especifican un comportamiento trabajando sobre la representación elegida, replicando con exactitud todos los factores como el overflow de las operaciones aritméticas. Por este motivo, el soporte de los tipos primitivos aumenta de forma considerable el tamaño del modelo generado.

```

sig integer {
  bit31: boolean,
  bit30: boolean,
  ...
  bit0:  boolean
}

```

Figura 2.2: Modelo Alloy para números enteros de 32 bits.

```

pred EQ[i1, i2 : integer] {
  i1.bit31 = i2.bit31 and
  i1.bit30 = i2.bit30 and
  ...
  i1.bit0  = i2.bit0
}

```

Figura 2.3: Predicado de igualdad de números enteros de 32 bits.

2.3. Verificación acotada

Como ya fue mencionado en la sección 1.2, el proceso de verificación acotada toma como entrada un programa con su especificación y las cotas del análisis para luego buscar, dentro de los límites impuestos, una traza que viole el contrato. Al igual que con JForge y Miniatur, con TACO el programa y la especificación son traducidos a una fórmula SAT y procesados con un SAT-Solver.

En TACO el usuario debe limitar el tamaño de las trazas de ejecución y el espacio de estados a ser considerado al seleccionar un scope de análisis. Como es explicado en la disertación de doctorado de Dennis [18], estas limitaciones resultan en una subaproximación, eliminando posibles comportamientos pero nunca agregando nuevos. Por este motivo si un contraejemplo es encontrado durante el análisis no puede ser un resultado espurio, por lo que efectivamente evidencia una violación en el contrato. Si ningún contraejemplo es encontrado, dado que el análisis es exhaustivo dentro del espacio de búsqueda acotado, puede concluirse que la especificación es válida dentro del alcance del análisis. No obstante, la especificación puede no ser válida en general ya que podría encontrarse un contraejemplo si se utilizara un scope de análisis más amplio.

Estas características de la verificación acotada y el pronunciado crecimiento del tiempo requerido por los algoritmos de SAT-Solving motivaron el comportamiento iterativo de Fajita. Trabajando sobre la hipótesis de que gran parte de los errores pueden ser detectados con instancias pequeñas, en la sección 3.6 se introduce un mecanismo de exploración del espacio de estados. Este mecanismo incrementa gradualmente el scope de análisis, con la esperanza de que las iteraciones con cotas muy restrictivas logren alcanzar un gran índice de cubrimiento del criterio elegido. Esto se realiza con el fin de poder eliminar del modelo los aspectos ya resueltos y repetir el análisis con un espacio de búsqueda menos acotado pero con un modelo más simple.

Se considera que en la práctica muchos errores pueden detectarse con instancias pequeñas por la naturaleza iterativa de los algoritmos, pues es esperable que un programador determine una o más condiciones de parada o casos base y que describa un mecanismo iterativo que conduzca a la ejecución hacia uno de los casos base. Por consiguiente si un conjunto de tests ejercita todos los casos base, y todos los esquemas de iteración, estos últimos deberían comportarse de forma similar tanto para instancias de tamaño limitado como para instancias más grandes dado que el mismo código se repite una y otra vez.

2.4. Generación de casos de test

En esta sección se explica el procedimiento realizado para construir el caso de test correspondiente a una valuación obtenida a partir de un modelo construido con TACO. Para lograrlo se utilizan las funcionalidades de las herramientas involucradas que permiten interpretar los resultados obtenidos, gracias a ellas el procedimiento resulta trivial. El flujo de ejecución comienza con Alloy dado que la entrada es una valuación retornada por un SAT-Solver, sigue con TACO y termina con Fajita escribiendo el caso de test en código Java.

El procedimiento puede resumirse en los siguientes pasos:

1. A partir de una valuación para un modelo Alloy (encontrada utilizando un SAT-Solver) compuesta por valores de verdad para las variables proposicionales utilizadas en el modelo, se utiliza la herramienta Alloy para obtener la expresión en lenguaje Alloy que la representa.
2. Una vez obtenida la expresión Alloy se utiliza TACO, que reconstruye a partir de dicha expresión el estado interno de la clase bajo test y el de los objetos usados como parámetros. Normalmente TACO utiliza esta funcionalidad para mostrar un contraejemplo encontrado durante la verificación de una pieza de software.
3. Finalmente Fajita escribe en un archivo un método Java que representa el caso de test. Fajita inicializa los objetos involucrados en el test con el estado de los objetos en el contraejemplo devuelto por TACO, utilizando la librería de *reflection* provista en la librería estándar de Java.

El procedimiento se repite para cada valuación encontrada, agregando los casos de test al mismo archivo. Para simplificar la ejecución de los tests generados, el código es escrito usando la librería de gestión de casos de test JUnit 4.0.

Cabe destacar que la utilización de *reflection* permite crear instancias que, mediante el uso normal de la interfaz del objeto podría no ser posible generarlas. Sin embargo, toda instancia generada respeta el invariante de representación, por lo que se considera válida aunque no sea posible construirla. En este caso se dice que el invariante provisto no se ajusta a la interfaz de la clase.

2.5. Innovaciones de TACO

Como ya fue mencionado, el proceso basado en análisis de satisfacibilidad requiere realizar la exploración del espacio de modelos válidos (aquellos que satisfacen la especificación) en la búsqueda de una valuación que no satisfaga la propiedad siendo verificada. Dado que esto requiere un tiempo exponencial en la cantidad de variables proposicionales, cualquier optimización del proceso puede representar una mejora sustancial. En esta sección se mencionan las optimizaciones de esta naturaleza incluidas en TACO que impactan en el desempeño de Fajita.

Las permutaciones o simetrías de los objetos representados en Alloy, también llamados átomos, no alteran el valor de verdad de las fórmulas Alloy. Por lo que una vez que una valuación es considerada, las resultantes de las permutaciones de los valores de los átomos que la conforman deberían ser ignoradas. Una forma de lograr esto es introduciendo predicados de rupturas de simetrías [34]. Dado que la ruptura de todas las simetrías requiere, en el caso general, la construcción de un predicado exponencialmente largo, Alloy sólo elimina algunas simetrías con predicados de propósito general con un tamaño polinomial.

En su trabajo de tesis doctoral, Juan Pablo Galeotti [24] propone predicados de rupturas específicos para evitar simetrías en la representación de los objetos en memoria. Allí, se muestra que la reducción del espacio de estados al eliminar las permutaciones mejora el análisis en varios ordenes de magnitud. Por lo que se concluye que la habilidad de reducir el espacio de estados es central para lograr la escalabilidad de la técnica.

Una ventaja de utilizar TACO como un componente de Fajita es que implementa una técnica que combina los predicados de rupturas de simetrías mencionados con una característica de Alloy, la utilización de Kodkod [33] como representación intermedia. En esta representación cada campo f es traducido a una matriz de variables proposicionales junto con dos cotas L_f (la cota inferior de f) y U_f (la cota superior de f), que imponen restricciones a la interpretación del campo. En cualquier modelo, la interpretación del campo f debe satisfacer que $L_f \subseteq f \subseteq U_f$. Es decir, las tuplas que son parte de L_f necesariamente pertenecen a f , mientras que las que no pertenecen a U_f no pueden ser parte de f .

Dado que la traducción de código Java modela los campos Java como relaciones Alloy y que cada tupla en la relación es una variable proposicional cuyo valor de verdad indica si pertenece o no a la relación, las variables correspondientes a tuplas que no pertenezcan a U_f pueden ser directamente reemplazadas con falso en el proceso de traducción. Como ha sido mencionado, este tipo de reducciones de variables puede mejorar significativamente el tiempo requerido por el análisis.

Usualmente las estructuras de datos complejas tienen invariantes de representación complejos que imponen restricciones en la topología de los datos. Si antes de efectuar la traducción a lógica proposicional, se pudiera determinar si una relación entre dos átomos no puede formar parte de una instancia que cumpla el invariante de clase, las tuplas que representan la relación podrían ser removidas de la cota superior (reemplazándolas con falsos). Cuando una tupla es removida de la cota superior se

dice que la cota resultante es más ajustada que la anterior. Para que este análisis sea posible es necesario que los átomos sean generados de forma tal que se eviten permutaciones que permitan a los átomos relacionarse de formas distintas.

Consideremos la representación de una lista enlazada de longitud 3, donde los átomos N_0 , N_1 y N_2 están relacionados mediante la relación “next” y el invariante de representación exige que no existan ciclos. En principio 6 permutaciones de nodos son posibles, siendo todas las instancias simétricas (Fig. 2.4).

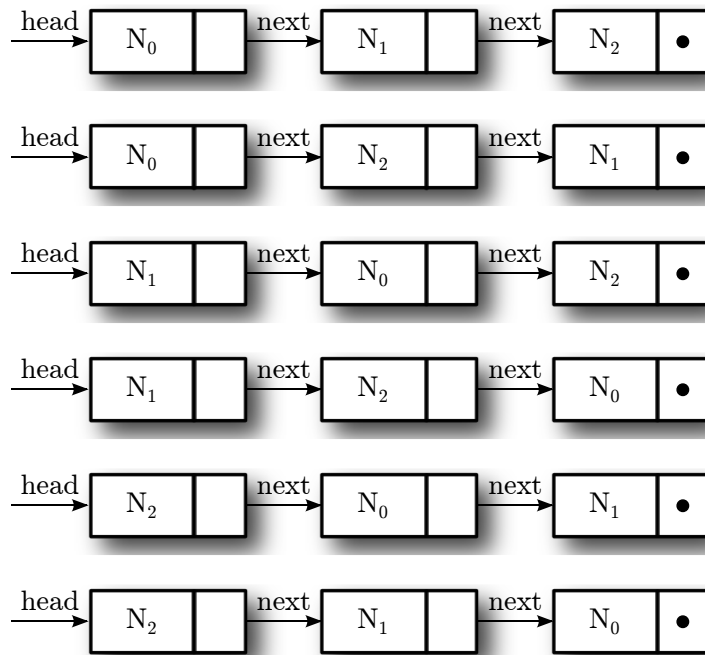


Figura 2.4: Listas enlazadas de longitud 3 simétricas respecto a la relación next.

Gracias a los predicados de rupturas de simetrías sólo se considera en el análisis la secuencia que se muestra en la Figura 2.5. Reduciendo efectivamente el espacio de entradas a ser explorado.

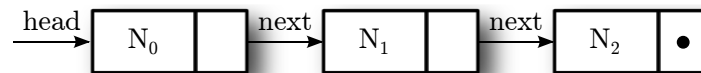


Figura 2.5: Lista enlazada de longitud 3 distinguida.

Puede verse que 4 relaciones quedan descartadas, por lo que pueden ser eliminadas de la cota superior (Fig. 2.6).

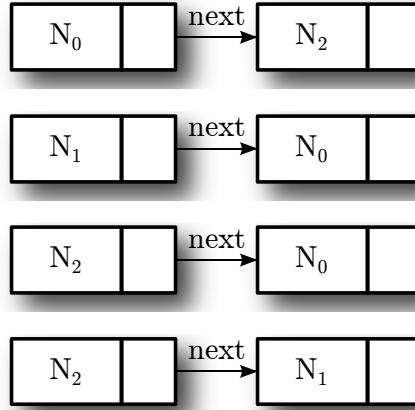


Figura 2.6: Relaciones eliminadas mediante ruptura de simetrías.

El problema de este acercamiento es que el cálculo de cotas superiores ajustadas es computacionalmente costoso. Sin embargo, estas cotas son frecuentemente reutilizadas, por ejemplo al utilizar distintos criterios durante el análisis, ya que las cotas son válidas mientras el invariante de representación no cambie. Por consiguiente el costo de computarlas puede ser amortizado.

TACO incorpora un repositorio de cotas precalculadas utilizando un cluster de computadoras [24], al que se agregaron las cotas para algunos de los objetos de estudio utilizados en este trabajo. La arquitectura de TACO extendida con el repositorio de cotas se muestra en la Figura 2.7. Estas cotas, junto con restricciones adicionales agregadas por Fajita correspondientes al criterio de selección utilizado, son usadas para construir el modelo Kodkod que finalmente es traducido a una fórmula proposicional en CNF.

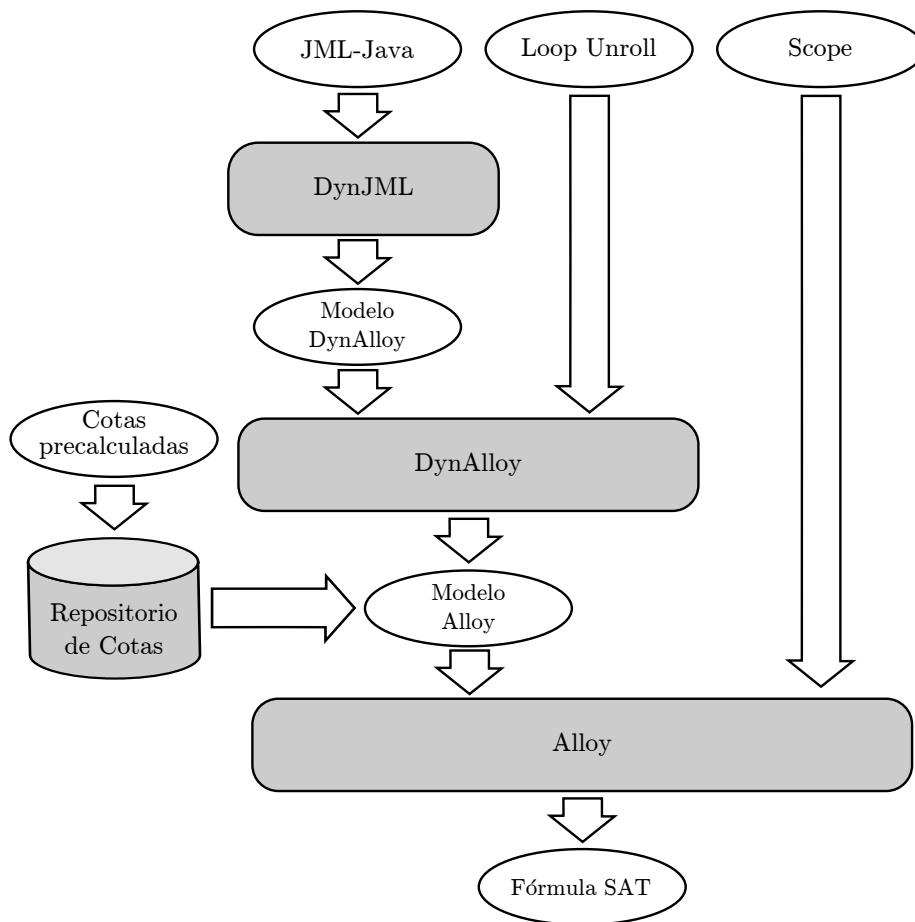


Figura 2.7: Arquitectura de TACO extendida con el repositorio de cotas.

Capítulo 3

Desarrollo

En este capítulo se describen en detalle los mecanismos que permiten utilizar las herramientas de verificación formal acotadas, introducidas en el capítulo 2, para la generación de casos de test. Se presentan, también, estrategias introducidas para explorar paulatinamente el espacio de entradas con el fin de mantener bajo control el crecimiento del espacio de búsqueda. Las técnicas y algoritmos aquí presentados constituyen el principal aporte de esta tesis. Fajita es el prototipo experimental resultado de la implementación de estos métodos.

Fajita utiliza SAT-Solving incremental para la generación de datos que conformen casos de test en pos de maximizar el índice de cobertura de un criterio de selección dado. El procedimiento es presentado en función de un criterio de selección genérico, generando una partición del dominio de entradas de acuerdo al criterio seleccionado, garantizando que todas las clases de equivalencia que puedan ser cubiertas dentro del scope de análisis establecido sean cubiertas produciendo al menos un test para cada una.

Cualquier criterio de selección de casos de test, ya sea de caja abierta o de caja cerrada, define una forma de considerar dos posibles entradas como equivalentes. Por ejemplo, el criterio de cubrimiento de decisiones considera que dos entradas son equivalentes si al ejecutar el programa para cada una se logra que la evaluación de cada condición retorne el mismo valor de verdad. De la misma forma el criterio de cubrimiento de objetivos permite considerar dos casos de test como equivalentes si cubren los mismos objetivos.

Formalmente, sea S el conjunto de estados que representan el espacio de entrada válido para el programa bajo test. Todo criterio da origen a un conjunto de predicados p_1, \dots, p_k pertenecientes a S , que inducen una partición en S , donde cada p_i caracteriza una clase de equivalencia del conjunto de entradas. Una instrumentación de un criterio de selección es una elección de estos predicados y su correcta incorporación al modelo relacionándolos con los aspectos observados por el criterio.

3.1. Generación de casos de test usando SAT-Solving incremental

En esta sección se explica la ventaja que representa la utilización de un SAT-Solver incremental. A diferencia de un SAT-Solver común, un SAT-Solver incremental tiene dos características distintivas. Cuando una valuación para una fórmula ϕ es encontrada:

1. Una nueva formula α puede ser agregada de forma tal de reducir el espacio de búsqueda intentando encontrar una valuación que satisfaga la fórmula $\phi \wedge \alpha$.
2. El estado interno del SAT-Solver conserva registro del espacio de búsqueda explorado, por lo que al continuar la búsqueda de la fórmula modificada no vuelve a explorar los estados ya visitados.

En este punto es necesario introducir la notación $\text{I-SAT}(\phi, \alpha)$ que denota la invocación del SAT-Solver incremental en una fórmula ϕ agregando la información provista por la fórmula α . La secuencia de llamadas $\text{I-SAT}(\phi, \alpha_1), \dots, \text{I-SAT}(\phi, \alpha_1 \wedge \dots \wedge \alpha_m)$ genera valuaciones para las fórmulas

$$\begin{aligned} &\phi \wedge \alpha_1, \\ &\phi \wedge \alpha_1 \wedge \alpha_2, \\ &\vdots \\ &\phi \wedge \alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_m \end{aligned}$$

Dados los predicados p_1, \dots, p_k mencionados anteriormente, el objetivo es generar datos de entrada que conformen casos de test para cubrir cada clase de equivalencia determinada por los predicados. Sea ϕ la fórmula que caracteriza el espacio de estados S y supongamos que cada predicado correspondiente a una clase de equivalencia es capturado por una variable proposicional P_i . Luego, cuando una valuación es obtenida por el SAT-Solver incremental, ésta determina un caso de test donde una variable P_i es verdadera. Entonces simplemente agregando la cláusula $\neg P_i$ a ϕ se garantiza que las valuaciones obtenidas por invocaciones futuras corresponderán a clases de equivalencia distintas de la capturada por P_i .

3.2. Algoritmo de cobertura de particiones

En esta sección se presenta una definición intuitiva de cubrimiento óptimo y un algoritmo que obtiene este cubrimiento óptimo aprovechando las cualidades de los SAT-Solvers incrementales expuestas en la sección anterior. El algoritmo toma como parámetros de entrada la fórmula proposicional que modela el programa bajo análisis y luego de múltiples invocaciones al SAT-Solver incremental retorna un conjunto de casos de test que cubre todas las particiones que pueden cubrirse dentro del scope de análisis utilizado.

Definición. Una cobertura óptima para una partición inducida por predicados p_1, \dots, p_k pertenecientes a un dominio \mathbb{S} , es un conjunto de valores V que cumplen que:

1. $V \subseteq \mathbb{S}$
2. No hay dos elementos en V tal que satisfagan el mismo predicado p_i .
3. Cada p_i satisfacible dentro del scope de análisis posee un representante en V .

Recordando la noción de eficiencia de un conjunto de casos de test introducida en la sección 1.5, se busca cubrir exhaustivamente el espacio de entrada con la menor cantidad de casos test. Al particionar este espacio resulta natural que un cubrimiento óptimo tenga por lo menos un representante de cada clase de equivalencia. Es evidente que considerar un caso por clase es lo mínimo necesario para cubrirlo de forma completa.

Como las clases cubiertas pueden dejar de ser consideradas gracias al mecanismo de incorporación de cláusulas de forma incremental, puede diseñarse un algoritmo que genere exactamente un caso de test por clase de equivalencia, obteniendo un cubrimiento óptimo.

```

ObtenerCoberturaDeParticiones(phi) : suite
  suite = Vacio
  valuacion = I-SAT(phi, true)
  while (valuacion != UNSAT)
    test = GenerarTest(valuacion)
    Agregar(test, suite)
    clausula = clausula and
      CrearClausulaDeRemocion(valuacion)
  valuacion = I-SAT(phi, clausula)

```

Algoritmo 3.1: Cubrimiento incremental de particiones.

Este algoritmo opera sobre una fórmula proposicional con la representación de un programa (**phi**) y retorna un conjunto de casos de test (**suite**). Como fue explicado en el capítulo 2, Fajita parte de un programa Java con anotaciones JML y utilizando la herramienta TACO obtiene un modelo Alloy que lo representa. Luego a partir de los aspectos relevantes del programa de entrada (que dependen del criterio elegido) se sintetizan automáticamente predicados p_1, \dots, p_k que son agregados al modelo Alloy obtenido. Para remover la cláusula del espacio de búsqueda del SAT-Solver se utiliza el procedimiento **CrearClausulaDeRemocion**, su comportamiento depende del criterio de selección elegido por lo que será explicada junto a las otras decisiones necesarias para implementar cada criterio.

En las secciones siguientes se describe cómo definir los predicados p_1, \dots, p_k de forma tal de capturar distintos criterios de selección, tanto de caja cerrada como de

caja abierta. De esta forma se muestra cómo cada criterio se ajusta al escenario general propuesto en esta sección y cómo el Algoritmo 3.1 permite obtener una cobertura óptima según estos criterios.

3.3. Generación incremental de casos de test de caja cerrada

El procedimiento genérico descrito en la sección anterior puede ser instanciado para generar casos de test con una cobertura óptima respecto a un criterio de caja cerrada. Como fue mencionado al inicio de este capítulo, cualquier criterio establece una relación de equivalencia entre las distintas entradas posibles. En el caso de criterios de caja cerrada, la relación entre la cobertura de particiones y la separación en clases de equivalencia es directa.

Como ya fue mencionado en la sección 1.4.1, Fajita incorpora el criterio de caja cerrada conocido como cobertura de métodos booleanos introducido en [4]. El cual considera los métodos booleanos sin argumentos de una clase para definir la partición del espacio de estados. Fajita exige, además, que los métodos considerados no tengan efectos de lado, pues en ese caso sería necesario considerar las posibles permutaciones en el orden en el que se combinan los predicados, aumentando la complejidad del análisis.

Se requiere, también, que los métodos puedan invocarse sobre cualquier instancia válida, es decir, que no tengan precondiciones sobre el objeto receptor del mensaje (o parámetro implícito). Esto se debe a que los métodos con precondiciones inducen una subpartición de las clases de equivalencia generadas por las precondiciones. En la sección 3.3.1 se analiza la factibilidad de ampliar este criterio a métodos booleanos con argumentos y precondiciones.

Puede considerarse como ejemplo una `Pila` capaz de contener una cantidad máxima de elementos, fijada por ejemplo al momento de su construcción. Es razonable que tal clase posea dos métodos booleanos sin argumentos `estaVacía()` y `estaLlena()`. Estos dos métodos definen cuatro clases, correspondientes a las cuatro combinaciones posibles de sus resultados y constituye una partición del espacio de estados de cualquier instancia del tipo `Pila`.

Cabe destacar que es posible que no puedan cubrirse todas las clases de equivalencia, esto depende fuertemente del invariante del objeto. Si bien en el ejemplo es esperable que pueda construirse una instancia vacía, una llena y una en un estado intermedio; no necesariamente es deseable poder construir una que esté simultáneamente vacía y llena. Si la clase `Pila` no admite ser construida con una capacidad nula esta partición no puede ser cubierta, por el contrario si el invariante no lo restringe puede generarse una instancia con estas características, lo que constituiría un caso de test denominado normalmente “de borde”. Cabe destacar que este hecho no afecta la cualidad de optimalidad del algoritmo, pues la definición presentada se refiere sólo a las clases que pueden ser cubiertas dentro del scope de análisis.

Puede considerarse, de forma más general, una clase C con k métodos booleanos sin argumentos $q_1(X), \dots, q_k(X)$ (donde X representa el parámetro implícito de los

métodos). Tuplas de la forma $\langle q_1(o), \dots, q_k(o) \rangle$, para una instancia o de la clase C , inducen una partición en el conjunto de posibles estados del objeto o , o equivalentemente, en el espacio de estados de la clase C . Esto es garantizado por el siguiente teorema.

Teorema 1. *Sea \mathbb{S} un conjunto de estados y q_1, \dots, q_k predicados sobre \mathbb{S} .*

Sea R una relación binaria en \mathbb{S} definida de la siguiente manera:

$$s_1 R s_2 \Leftrightarrow \langle q_1(s_1), \dots, q_k(s_1) \rangle = \langle q_1(s_2), \dots, q_k(s_2) \rangle$$

Entonces, R es una relación de equivalencia.

La demostración es trivial ya que la relación está construida sobre la igualdad entre tuplas de valores booleanos, que es una relación de equivalencia. Luego las propiedades de reflexividad, simetría y transitividad pueden ser probadas a partir de las correspondientes a la igualdad de tuplas de valores booleanos.

Este teorema nos permite construir el conjunto de predicados $P_R(X)$, que contiene predicados p_j formados por la conjunción de los predicados $q_1(X), \dots, q_k(X)$ y sus negaciones $\neg q_1(X), \dots, \neg q_k(X)$, donde cada q_i aparece exactamente una vez. Es decir, $P_R(X)$ posee 2^k predicados, con todas las combinaciones de los predicados q_i y sus negaciones.

Luego el Algoritmo 3.1 puede instanciarse utilizando estos predicados para distinguir las clases de equivalencia. Sin embargo, para completar el algoritmo es necesario indicar el mecanismo por el cual se construye, a partir de una valuación encontrada, la cláusula incremental que remueve la partición cubierta del espacio de búsqueda.

Para ello necesitamos que la validez de cada predicado $q_i(X)$ sea capturada por una variable booleana Q_i . Esta condición es fácil de cumplir, ya que basta con codificar las $q_1(X), \dots, q_k(X)$ como una fórmula proposicional en CNF y forzar el hecho de que a cada variable Q_i le sea asignado el valor de verdad correspondiente al predicado $q_i(X)$. Esto puede lograrse agregando al modelo Alloy un axioma de la forma:

$$(Q_1 \Leftrightarrow q_1(X)) \wedge \dots \wedge (Q_k \Leftrightarrow q_k(X)) \quad (3.1)$$

Luego dada una valuación val debe construirse una cláusula proposicional que remueva la partición cubierta del espacio de búsqueda. En este caso basta con construir una cláusula de remoción c de la siguiente forma:

$$c = \bigvee_{1 \leq i \leq k, val(Q_i)=true} \neg Q_i \vee \bigvee_{1 \leq j \leq k, val(Q_j)=false} Q_j \quad (3.2)$$

Donde la notación $val(Q_i)$ denota el valor de la variable Q_i en la valuación val .

El siguiente teorema muestra que agregando cláusulas de esta forma el Algoritmo 3.1 efectivamente logra una cobertura óptima.

Teorema 2. *Sea C una clase con k métodos booleanos sin parámetros $q_1(X), \dots, q_k(X)$. El Algoritmo 3.1, instanciado con los predicados pertenecientes al conjunto P_R (con las 2^k combinaciones de los predicados q_i y sus negaciones) y usando la fórmula de remoción de clases (3.2), produce una cobertura óptima.*

Demostración. Basta verificar que la cláusula de remoción (3.2) elimina únicamente una clase de equivalencia cubierta (una clase de la que se disponga una valuación representativa) al ser utilizada en el Algoritmo 3.1.

Sea val una valuación encontrada por una invocación al procedimiento I-SAT. Tal valuación satisface exactamente un predicado p_j del conjunto P_R y no satisface la cláusula de remoción obtenida a partir de val . Se desea verificar que la clase inducida por este predicado es efectivamente removida al agregar la cláusula.

Consideremos cualquier fórmula CNF ϕ que contenga la cláusula (3.2). Supongamos que la clase inducida por el predicado p_j no fue removida de ϕ , entonces debe existir una valuación val' que satisfaga $\phi \wedge p_j$ y que al menos una de sus variables Q'_1, \dots, Q'_k difiera de las variables Q_1, \dots, Q_k de val , dado que val no satisface la cláusula.

En otras palabras, dado que $val \neq val'$, debe existir un índice i , con $1 \leq i \leq k$, tal que $Q_i \neq Q'_i$. Pero en el caso de que se cumpla que $Q_i = true$ y que $Q'_i = false$ sucede que val' no puede satisfacer p_j . Lo mismo sucede en el caso que $Q_i = false$ y $Q'_i = true$. Por lo que llegamos a un absurdo que proviene de suponer que la incorporación de la cláusula no remueve la clase.

Debemos verificar, también, que al agregar la cláusula sólo se remueve la clase inducida por el predicado p_j y ninguna otra. Sean ϕ una fórmula CNF, val una valuación que la satisface, α la cláusula de remoción correspondiente, $\theta = \phi \wedge \alpha$ y val' otra valuación que satisface ϕ distinta de val .

Al igual que antes debe existir un índice i tal que $1 \leq i \leq k$ y que $Q'_i \neq Q_i$. Entonces, si $Q_i = true$ sucede que $Q'_i = false$ y por consiguiente $\neg Q'_i = true$, por lo tanto val' satisface θ . Por otro lado si $Q_i = false$ sucede que $Q'_i = true$, por lo que en este caso también se ve que val' satisface θ . En conclusión agregar la cláusula (3.2) solamente remueve la clase correspondiente a la valuación val .

□

Observación. Es posible que con el scope de análisis elegido no sea posible construir un elemento representativo de una partición. El hecho de que tal instancia no pueda ser generada no depende de las cláusulas agregadas sino únicamente de las restricciones impuestas en el espacio de búsqueda. En tal caso el resultado de la invocación al SAT-Solver retornará UNSAT al igual que en el caso en que la partición no pueda cubrirse en general. Si existiera un scope de análisis mayor que lograra efectivamente cubrir la partición faltante, el algoritmo generaría una cláusula de remoción distinta para tal scope, pues representa una valuación distinta de los predicados q_1, \dots, q_k . El procedimiento para este scope eventualmente generaría, además, las mismas cláusulas que el caso de menor tamaño, pero no necesariamente generando los mismos datos para los casos de test o en el mismo orden.

Esto indica que las cláusulas de remoción obtenidas por una ejecución del método para un scope de análisis dado, pueden ser reutilizadas para un scope mayor con el fin de no volver a generar las soluciones ya encontradas, reduciendo así el espacio de búsqueda en el segundo caso. En la sección 3.6 se presenta una extensión del Algoritmo 3.1 que incrementa paulatinamente el scope de análisis aprovechando este hecho.

3.3.1. Cobertura de métodos booleanos sin restricciones

El criterio de cobertura de métodos booleanos se basa en la suposición que en objetos bien diseñados los métodos booleanos sin argumentos dan suficiente información sobre las características de la clase a la que pertenecen, para mejorar la selección de casos de test [4]. No obstante, si pudiera incluirse más información manteniendo un criterio de caja cerrada, podría lograrse una mejor división del espacio de estados. En esta sección se evalúa una extensión al criterio de métodos booleanos, considerando también los métodos con argumentos y precondiciones.

```
class List {
    Node head;

    boolean isEmpty() { return head == null; }

    /* @requires: object != null */
    boolean contains(Object object) {
        Node it = head;
        while (it != null) {
            if (object.equals(it.elem)) return true;
            it = it.next;
        }
        return false;
    }
}
```

Figura 3.1: Lista enlazada con métodos booleanos sin argumentos insuficientes.

En el ejemplo de la Figura 3.1, los métodos booleanos sin argumentos no dan suficiente información para generar casos que cubran exhaustivamente los dos métodos. Sin embargo, al considerar los métodos con argumentos pueden distinguirse otros casos relevantes. El caso más directo es el de cubrir ambos puntos de salida del método `contains`, pero también podría mejorarse el índice de cobertura de un método `remove` cuyo funcionamiento dependa de si el elemento a remover está o no en la lista.

Suponiendo que los argumentos del método f no tienen precondiciones, es decir, que es válido invocar el método con cualquier parámetro y sobre cualquier instancia, basta agregar a la representación de la clase un objeto por argumento, lo que aumenta el tamaño del espacio de entradas. Luego, es posible generar un nuevo método booleano sin parámetros f' que simplemente retorne el resultado de llamar al original parametrizándolo con los atributos agregados, volviendo al escenario de caja cerrada analizado en la sección anterior.

Por otro lado, el caso en el que un método posea una precondition sobre su parámetro implícito resulta más complejo, dado que se desea utilizar el método para generar una partición del espacio de estados, pero su comportamiento no es válido para cualquier elemento de ese espacio. Luego, para poder utilizar esta estrategia es necesario generar una instrumentación sobre el modelo Alloy con predicados $r_1(X), \dots, r_k(X)$ donde cada predicado r_i representa la precondition del método q_i , y predicados $q_1(X), \dots, q_k(X)$ que representan la invocación del método $q_i(X)$ sobre el parámetro implícito X . Finalmente debe agregarse, en lugar de la fórmula (3.1), un axioma que relacione los valores de las variables Q_1^r, \dots, Q_k^r y Q_1^q, \dots, Q_k^q de la siguiente forma:

$$\bigwedge_{i=1}^k (Q_i^r \Leftrightarrow r_i(X)) \wedge (Q_i^q \Leftrightarrow (r_i(X) \Rightarrow q_i(X))) \quad (3.3)$$

Donde la variable Q_i^r captura el valor de verdad de la precondition r_i y la variable Q_i^q captura el valor de verdad del método booleano q_i si se cumple su precondition. Esta distinción es necesaria para poder garantizar que sólo se considerarán los métodos booleanos invocados correctamente, o en otras palabras, satisfaciendo sus condiciones.

Combinando ambos casos se puede extender el procedimiento anterior a métodos con condiciones que se refieran al estado de sus argumentos. Basta agregar al método duplicado f' la misma precondition que el método original f , sustituyendo los argumentos con los atributos generados para reemplazarlos. Cabe destacar que al introducir un objeto a la representación de la clase bajo análisis se amplía el espacio de búsqueda, dado que es un agregado de carácter existencial. El resultado de estas transformaciones se ejemplifica con la Figura 3.2, donde se agrega el atributo `contains_arg.0` para reemplazar el argumento del método `contains` y se lo duplica con una versión sin argumentos `contains_argumentless`.

En conclusión gracias a la forma en que Fajita realiza la construcción de instancias válidas para la generación de casos de test, es posible considerar un criterio de selección de caja cerrada de métodos booleanos con argumentos. No obstante, para instrumentarlo es necesario agregar al modelo el doble de variables proposicionales que en el acercamiento clásico, que sólo admite métodos sin parámetros. De esta forma puede lograrse una cobertura de particiones más granular, realizando un proceso computacionalmente más costoso.

3.4. Generación incremental de casos de test de caja abierta

En esta sección se muestra cómo instanciar el proceso genérico presentado en el Algoritmo 3.1 de forma de obtener una cobertura óptima según distintos criterios de testing de caja abierta. Como en el caso de caja cerrada, el acercamiento consiste en definir apropiadamente predicados que determinen una relación de equivalencia que pueda ser utilizada para particionar el espacio de entrada y la elección de las cláusulas de remoción. Se presenta cómo instrumentar los criterios de cobertura de objetivos, de sentencias y de decisiones.

```

class List {
    Node head;
    Object contains_arg_0;

    boolean isEmpty() { return head == null; }

    /* @requires: object != null */
    boolean contains(Object object) {
        Node it = head;
        while (it != null) {
            if (it.elem.equals(object)) return true;
            it = it.next;
        }
        return false;
    }

    /* @requires: contains_arg_0 != null */
    boolean contains_argumentless() {
        return contains(contains_arg_0);
    }
}

```

Figura 3.2: Ejemplo de transformación con métodos booleanos sin restricciones.

3.4.1. Cobertura de objetivos

Como fue mencionado en la sección 1.4.2, la cobertura de objetivos es utilizada como terreno de pruebas para las distintas herramientas de generación de casos de test, por lo que es de especial interés. En la sección 4.1.6 se comparan los resultados obtenidos por Fajita utilizando este criterio contra los obtenidos por otras herramientas del estado del arte.

Se desea generar casos de test que cubran objetivos introducidos en el código fuente. Para poder hacerlo una instrumentación sobre el código es realizada de forma automática por Fajita, agregando a la representación del objeto una variable booleana G_i para cada objetivo i . Estas variables son inicializadas con *false* y asignadas con *true* en los lugares marcados como objetivos, como se muestra en la Figura 3.3.

Al efectuar la traducción entre el código fuente y el modelo Alloy se plasma el comportamiento de las variables booleanas agregadas automáticamente. Luego, predicados pg_1, \dots, pg_k son agregados al modelo y relacionados con cada variable G_1, \dots, G_k de forma tal de poder particionar el dominio y así poder aplicar el Algoritmo 3.1.

Para poder expresar el cambio de estado de la variable booleana G_i al considerar una traza que alcanza el objetivo i , el modelo Alloy introduce una variable proposicional adicional G'_i que indica si la traza atraviesa el objetivo. Por lo tanto, pueden escribirse los predicados pg de la siguiente forma:

$$pg_i(X) = G'_i \quad (3.4)$$

```

class Clase {

    static void metodo(Object arg1, Object arg2) {
        if (arg1 != null) {
            @goal(1)
        }
        if (arg2 != null) {
            @goal(2)
        }
    }
}

/* Clase se transforma en ClaseInstrumentada. */

class ClaseInstrumentada {

    static boolean G1;           // variables de objetivo
    static boolean G2;

    static void metodo(Object arg1, Object arg2) {
        G1 = G2 = false;       // inicializacion
        if (arg1 != null) {
            G1 = true;         // objetivo 1 cubierto
        }
        if (arg2 != null) {
            G2 = true;         // objetivo 2 cubierto
        }
    }
}

```

Figura 3.3: Ejemplo de instrumentación del criterio de objetivos.

Esto relaciona los predicados pg_i con las variables booleanas G_i aprovechando el hecho que la variable proposicional agregada para modelar el cambio de estado es verdadera sólo cuando la asignación es efectuada. Adicionalmente, se agrega un axioma al modelo que exige la consideración de valuaciones que cubran como mínimo un objetivo.

Para este caso, la función `CrearClausulaDeRemocion` utilizada en el Algoritmo 3.1 debe producir, a partir de una valuación válida, una cláusula c de la siguiente forma:

$$c = \bigvee_{\substack{1 \leq i \leq k, \\ G_i \text{ no cubierto anteriormente}}} G_i \quad (3.5)$$

El siguiente teorema muestra que bajo estas condiciones el Algoritmo 3.1 enumera un conjunto de soluciones que se caracteriza por cubrir todos los objetivos alcanzables dentro del scope de análisis.

Teorema 3. *Partiendo del código fuente instrumentado con las variables booleanas G_1, \dots, G_k indicando la posición de los objetivos, habiendo actualizado el modelo Alloy con los predicados pg_1, \dots, pg_k definidos en (3.4) y utilizando la implementación de la función de creación de cláusulas de remoción (3.5); el Algoritmo 3.1 produce una cobertura de objetivos.*

Demostración. Observemos que cada vez que se obtiene una valuación a partir de una invocación a I-SAT, en el Algoritmo 3.1, la valuación cubre la clase inducida por la tupla $\langle pg_1, \dots, pg_k \rangle$. Consideremos las cláusulas c_1, \dots, c_{n-1} introducidas por la función `CrearCláusulaDeRemocion` en las iteraciones $1, \dots, n$ respectivamente. Entonces la valuación val obtenida durante la n -ésima iteración debe satisfacer:

$$\bigwedge_{1 \leq i \leq n-1} c_i$$

La valuación debe cubrir una clase en la que una variable G_j , que no haya sido asignada con *true* anteriormente, sea asignada con *true*. Por lo que cada iteración cubre al menos un objetivo nuevo.

Debemos demostrar que cada objetivo alcanzable dentro del scope de análisis sea cubierto por el algoritmo y que ningún objetivo inalcanzable sea cubierto. Lo segundo es sencillo porque las valuaciones son coherentes con las ejecuciones del método y no puede existir una ejecución que atraviese un objetivo inalcanzable, como tampoco puede obtenerse una traza fuera del scope de análisis.

Supongamos que el algoritmo termina en la n -ésima iteración sin cubrir el objetivo G alcanzable dentro del scope de análisis. Luego la fórmula obtenida es insatisfacible aunque existe una traza de ejecución que atraviesa G . A partir de esta traza se puede construir una valuación val' que represente el estado de la entrada del método invocado sobre el objeto o y en donde se cumpla que cada $pg_i(o) = true$ si el objetivo G_i fue cubierto e igual a *false* en caso contrario.

Por lo tanto la valuación val' satisface la fórmula original y las cláusulas c_1, \dots, c_{n-1} . Lo que contradice la insatisfacibilidad de la fórmula, llegando a un absurdo producto de suponer que el algoritmo puede terminar sin cubrir el objetivo G .

□

Corolario. Como en cada iteración se cubre al menos un objetivo nuevo, dada una clase con k objetivos el algoritmo termina después de a lo sumo k invocaciones al SAT-Solver.

Observación. A diferencia del caso de cobertura de particiones, en este caso no hay una noción estricta de eficiencia, pues en el caso general no es sencillo determinar la cantidad mínima de tests necesaria para lograr la cobertura óptima. Este hecho se refleja en la generación de la cláusula de remoción, que en el caso del criterio de cobertura de particiones elimina exactamente una partición mientras que en este caso puede eliminar múltiples objetivos.

3.4.2. Cobertura de sentencias

Como fue mencionado en la sección 1.4.3, un criterio popular es el de cobertura de sentencias, donde el conjunto de tests debe ejercitar toda sentencia ejecutable al menos una vez. La cobertura de sentencias puede ser lograda como un caso particular de la cobertura de objetivos, donde se introduce un objetivo después de cada sentencia. Sin embargo este acercamiento requiere la introducción de demasiadas variables aumentando la carga computacional del problema innecesariamente. Recordando que el criterio de cobertura de bloques resulta equivalente al de sentencias [22], el problema puede ser reducido a agregar un objetivo después de cada punto de decisión en el código. Cabe destacar que este acercamiento puede llegar a requerir una cantidad de variables proposicionales dos veces mayor a la cantidad de condiciones de ramificación. No obstante, considerando la capacidad de los SAT-Solvers modernos esta representación resulta perfectamente viable.

3.4.3. Cobertura de decisiones

El criterio de cobertura de decisiones requiere que el conjunto de tests evalúe cada condición de ramificación tanto por verdadero como por falso. Para instrumentar el criterio es necesario seleccionar predicados de forma tal de definir clases de equivalencia y generar cláusulas para eliminar las clases indeseadas.

La instrumentación puede extenderse de la presentada para cobertura de bloques de forma tal que, además de agregar un objetivo luego de cada ramificación, también se agregue uno que cubra el caso en el que la condición evalúa a falso. Es decir, en los puntos en donde se realiza una decisión debe incluirse un objetivo tanto en la rama verdadera como en la falsa. En el caso de las estructuras condicionales que no explicitan un comportamiento para el caso falso, es necesario agregarlo para poder marcar el objetivo. En el caso de los ciclos es necesario agregar la infraestructura que permita verificar si una traza no ejercita el cuerpo del ciclo.

En la Figura 3.4 se esquematiza la instrumentación mediante la utilización de grafos de control de flujo. A la izquierda se encuentra el grafo correspondiente a la estructura de control original y a la derecha el instrumentado. Los círculos denotan sentencias, las flechas indican las posibles direcciones del flujo de la ejecución dependiendo del resultado de la sentencia anterior y los bloques B_i subgrafos correspondientes a los bloques de ejecución de cada posible ramificación. Los círculos negros indican la presencia de una sentencia agregada para marcar un objetivo, se asume que las variables utilizadas para marcar los objetivos son inicializadas al inicio de la ejecución. Finalmente para el caso de los ciclos se utiliza una variable adicional (marcada con un círculo doble) que se utiliza para forzar que se cubra el caso en donde nunca se entra al ciclo.

Siguiendo el Teorema 3 puede demostrarse que esta instrumentación produce un cubrimiento de decisiones. Y puede deducirse como corolario que dado un método con k condiciones, el Algoritmo 3.1 termina a lo sumo después de $2k$ iteraciones. Se puede observar, también, que la cobertura de decisiones incluye a la de sentencias.

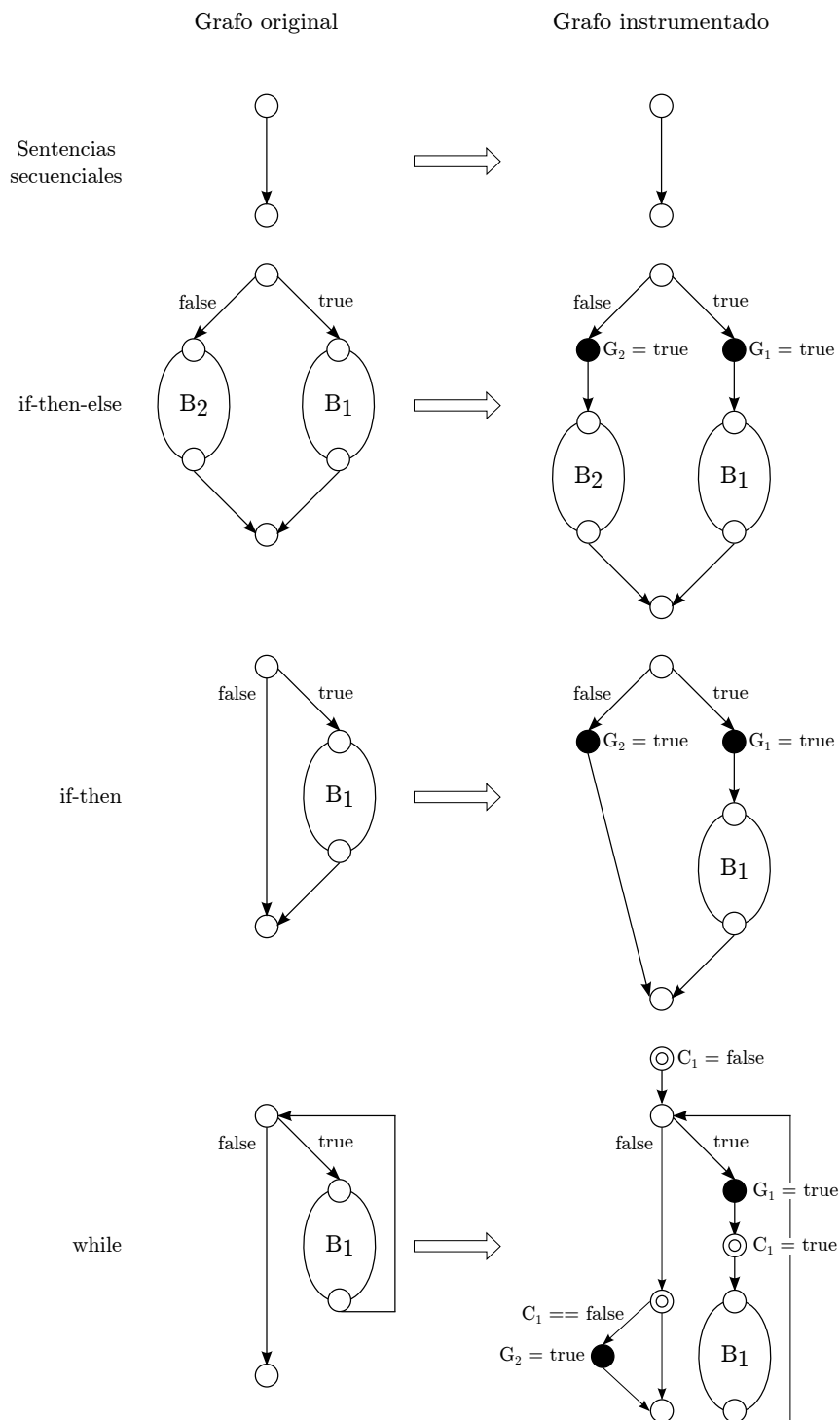


Figura 3.4: Esquematización de la instrumentación del criterio de decisiones sobre grafos de control de flujo de estructuras de control comunes.

3.5. Cubrimiento de caja gris

En esta sección se analiza un criterio de selección combinando criterios de caja abierta con criterios de caja cerrada, comúnmente denominados criterios de caja gris. Se propone una instrumentación que combine una iteración de cobertura de métodos booleanos seguida de una iteración de cobertura de decisión. Con la particularidad de que las ramas cubiertas durante la iteración de caja cerrada sean ignoradas desde el principio en la iteración de caja abierta.

Esta elección está motivada por el hecho de que los casos de test producto del criterio de métodos booleanos pueden expresarse claramente en los términos del programador. Recordando el ejemplo de la clase `Pila` (sección 3.3) con métodos `estaVacía()` y `estaLlena()`, los cuatro casos de test generados pueden ser fácilmente interpretados.

Por un lado de evidenciarse una falla, un programador podría comprender la naturaleza del error más rápidamente. Por otro lado, la mayoría de los estándares de calidad recomiendan cumplir con un alto cubrimiento según un criterio de decisiones (o alguna versión extendida como el criterio de decisiones múltiples usado para la certificación de equipos aéreos [35]). Esto es razonable, dado que los criterios de caja cerrada pueden no ser lo suficientemente exhaustivos como para validar de forma completa todos los detalles de la implementación de un programa.

La instrumentación de esta técnica es una combinación de las necesarias para cada uno de los dos criterios que la componen. En la primera etapa el código fuente es decorado con variables booleanas marcando un objetivo luego de cada una de las condiciones de ramificación, tanto para el caso verdadero como para el falso (como es explicado en la sección 1.4.4). Luego se generan los predicados correspondientes a la partición con métodos booleanos y se ejecuta el algoritmo para el criterio de caja abierta (Alg. 3.5).

```
ObtenerCoberturaDeCajaGris(codigo) : vals
instr = InstrumentarDecisiones(codigo)
instr = InstrumentarMetodosBooleanos(instr)
vals = ObtenerCoberturaMetodosBooleanos(instr)
instr = RemoverDecisionesCubiertas(vals)
if (HayDecisionesNoCubiertas(vals, codigo))
    vals = ObtenerCoberturaDecisiones(instr)
```

Algoritmo 3.2: Algoritmo de cobertura de caja gris.

Cada vez que se obtiene una valuación nueva, ésta además de representar un caso de test, indica una cobertura de decisiones. Al terminar con la etapa de caja abierta se vuelve a manipular el código fuente removiendo los objetivos de ramificación cubiertos en la primera etapa, lo que requiere volver a instrumentar el código Java y generar todos los modelos intermedios. En el caso en que todos los objetivos hayan sido cubiertos, el algoritmo se detiene. En caso contrario continúa persiguiendo los objetivos faltantes.

Observemos que para una clase con n métodos booleanos y k condiciones de ramificación esta instrumentación requiere $n + 2 \times k$ variables booleanas adicionales. En la sección 4.2 se hace un análisis de la eficiencia de este criterio, teniendo en cuenta que, si bien inicialmente su instrumentación requiere más variables proposicionales, las soluciones obtenidas sirven para restringir el espacio de búsqueda de la segunda etapa.

3.6. Exploración iterativa del espacio de estados

La utilización de herramientas de verificación acotada requiere, como ya fue mencionado, el empleo de límites para el scope de análisis. Valores para la cantidad máxima de objetos a ser almacenados en memoria y la longitud máxima de las trazas a ser analizadas pueden ser parametrizados en una ejecución de Fajita, dejando al usuario la capacidad y la responsabilidad de decidir cuán exhaustivo desea realizarse el análisis.

En el capítulo 2 se introdujo el comportamiento de Fajita a grandes rasgos y se mencionó que el scope de análisis puede ser decidido automáticamente por la herramienta. Este comportamiento es agregado con el fin de aumentar la automatización del procedimiento de generación de casos de test. En esta sección se explican los detalles concernientes a esta característica.

Se observa que en muchos casos la longitud de las trazas y la cantidad de objetos necesaria están relacionados, pues una gran cantidad de programas trabajan iterando sobre estructuras de datos. Por consiguiente a medida que aumenta el tamaño de las estructuras subyacentes, suele suceder que también es necesario aumentar la cantidad de iteraciones utilizadas en el análisis. La estrategia de exploración del espacio de estados utiliza este hecho, asumiendo que se trabaja con un programa con las características mencionadas. Su funcionamiento puede ser observado en el Algoritmo 3.3. Para los casos en donde no se verifique este comportamiento puede ser preferible utilizar la configuración manual de los parámetros.

La entrada del algoritmo es el código a ser analizado y una “estrategia de barrido” que indica qué valores del loop unroll deben ser analizados para cada valor del object scope. La estrategia de barrido es parametrizada con el fin de poder adaptar el comportamiento del procedimiento según el contexto de uso, ya que si bien se supone que los parámetros de object scope y loop unroll están relacionados, esta relación puede cambiar según el contexto. En la sección 4.1.6 se proponen distintas estrategias de barrido y se analiza el comportamiento de la herramienta para cada una de ellas.

Primero, el algoritmo infiere la cantidad de objetos necesaria analizando el código fuente (`objectScope`), de forma tal de intentar descubrir una cantidad de objetos suficiente para generar por lo menos una traza válida. Por ejemplo, al trabajar sobre el método de inserción en una estructura cuya precondition requiere que el objeto insertado no sea el objeto nulo, como mínimo son necesarios dos objetos para generar una ejecución válida, una instancia de la clase bajo test y otra para el parámetro a ser utilizado.

```

ExplorarDominio(codigo, estrategiaDeBarrido) : suite
  suite = Vacio
  objectScopeInicial = 1
  loopUnrollInicial = estrategiaDeBarrido.inicio()
  (objectScope, esMaximo) = InferirObjectScope(
    codigo, objectScopeInicial, loopUnrollInicial)
  while (true)
    loopUnroll = loopUnrollInicial
    do
      phi = GenerarModelo(codigo, objectScope, loopUnroll)
      suite += ObtenerCoberturaDeParticiones(phi)
      loopUnroll = estrategiaDeBarrido.incrementar(loopUnroll)
      if (#Objetivos(codigo) == #ObjetivosCubiertos(suite))
        exit
      if (esMaximo)
        exit
      while (unroll < estrategiaDeBarrido.fin())
        objectScope = objectScope + 1

```

Algoritmo 3.3: Exploración iterativa del espacio de estados.

El mecanismo de inferencia del object scope también intenta determinar el tamaño máximo utilizable, es decir, un tamaño para el espacio de búsqueda que sea suficiente para hallar el cubrimiento máximo para el criterio de selección elegido. Para estructuras dinámicas que aceptan una cantidad no acotada de estados no es posible encontrar tal cota, por lo que se retorna el valor inicial y el algoritmo continúa hasta cubrir todas las particiones o alcanzar el tiempo de espera. Sin embargo, en los casos en donde una cota es encontrada ese valor es utilizado en lugar del inicial y el algoritmo no explora más allá de esa cota.

El funcionamiento del procedimiento `InferirObjectScope` puede resumirse de la siguiente manera, dado el programa bajo análisis P , el object scope inicial s y una cota para el despliegue de ciclos u :

1. Sea D el diagrama de clases de P , donde un diagrama de clases es un grafo dirigido cuyos nodos representan tipos de datos y los ejes denotan una composición entre las clases conectadas. Sea N_{a_1}, \dots, N_{a_k} los nodos del diagrama D correspondientes a los tipos de los argumentos a_1, \dots, a_k del método bajo test. El algoritmo enumera la cantidad de caminos entre cada nodo N_{a_i} y cada nodo de D . Si en la enumeración existe un camino con un ciclo se considera que la cantidad de caminos $\#c$ es infinita. Por ejemplo, al trabajar con una lista enlazada donde cada nodo posee una referencia al siguiente, el diagrama de clase presenta un ciclo en la clase nodo.
2. Sea P_u el programa de entrada cuyos ciclos fueron desenrollados utilizando la cota u . Notamos con $\#o$ a la sumatoria de los literales, los objetos generados

por la resolución de expresiones aritméticas y objetos construidos utilizando el operador `new` en P_u . Dado que se considera el programa sin ciclos P_u , el valor de $\#o$ es siempre finito. En otras palabras, se busca contar la cantidad de objetos distintos que utiliza el programa. Puede observarse que este procedimiento retorna una aproximación por exceso de los objetos necesarios, pues se consideran los objetos necesarios para evaluar todos los caminos de ejecución posibles, mientras que la entrada generada desencadena una única traza.

3. Finalmente si $\#c$ es finita, el procedimiento `InferirObjectScope` retorna una tupla cuya primer componente tiene la suma entre $\#c$ y $\#o$ y la segunda componente `true`, lo que indica que el valor es considerado la cota máxima utilizable. En caso contrario retorna la suma entre s y $\#o$ como primer componente y `false` como segundo elemento de la tupla.

Una vez decidido el valor para la cantidad de objetos, el Algoritmo 3.3 realiza un barrido sobre trazas de distintas longitudes siguiendo la estrategia de barrido tomada como parámetro. Luego prosigue invocando al Algoritmo 3.1 (`ObtenerCoberturaDeParticiones`) para cada combinación de variables y removiendo de la instrumentación las clases ya cubiertas en iteraciones anteriores, lo que requiere volver a realizar la instrumentación en el código fuente y generar todos los modelos intermedios. Finalmente, siempre y cuando no se haya alcanzado una cobertura óptima, se incrementa el object scope en 1 y se repite el procedimiento.

Con este acercamiento se logra una mayor automatización de la técnica y además se espera que a medida que se realicen iteraciones se reduzca el espacio de búsqueda al remover las particiones cubiertas. Esto permitiría cubrir rápidamente los casos simples y destinar el poder computacional disponible a validar los casos más complejos. En la sección 4.1.6 se comparan distintas estrategias de barrido y se realiza un análisis de los resultados.

3.7. Especificaciones parciales

Es cierto que en la práctica gran parte del código existente no posee una especificación formal y que escribirlas y mantenerlas es un proceso costoso, al igual que la generación de casos de test. En esta sección se analiza el potencial de Fajita en escenarios con especificaciones reducidas de forma de poder minimizar el esfuerzo requerido en la construcción y mantenimiento del software.

Antes que nada hay que destacar que Fajita produce instancias que respetan el invariante de representación del objeto bajo test, pero no garantiza que tal instancia pueda ser construida mediante la utilización de su interfaz. Es decir, es posible que no exista ninguna secuencia de constructores y métodos de la interfaz con la que se pueda obtener una instancia igual a la generada por Fajita. En estos casos se dice que el invariante provisto no se ajusta correctamente al objeto y es considerado un defecto de la especificación. Por consiguiente para la utilización de Fajita es imprescindible la existencia de un invariante formal que se ajuste lo mejor posible al objeto analizado.

Fajita utiliza las precondiciones de los métodos bajo análisis para construir parámetros consistentes con la especificación. De no existir tales precondiciones se asume que cualquier instancia válida es aceptable, es decir una precondición siempre verdadera. Esta suposición es razonable en gran parte de los programas disponibles. Por lo tanto, si bien la presencia de precondiciones puede mejorar el rendimiento de Fajita, su ausencia no representa una limitación importante.

Por otro lado, si bien es posible utilizar las poscondiciones de los métodos para decidir automáticamente si el resultado de un test es correcto o no, actualmente Fajita no lo implementa. Esto deja la responsabilidad de verificar los resultados de la ejecución de los tests al programador, quien puede hacerlo de forma manual o insertando aserciones en el código. Es claro, entonces, que la falta de poscondiciones tampoco representa un obstáculo para la utilización de Fajita, si bien podría utilizarse para automatizar aún más el procedimiento.

En conclusión esta técnica es apta para escenarios con especificaciones parciales, ya que sólo los invariantes de representación son imprescindibles. Esto representa una ventaja con respecto a la verificación formal de software, que requiere una especificación total. Esta característica de Fajita revela su potencial en combinación con métodos de inferencia de especificaciones como DySy [36], que es capaz de inferir invariantes de clase.

Capítulo 4

Evaluación

En este capítulo se realiza una evaluación de Fajita. Para ello se la compara con los resultados obtenidos con otras herramientas del estado del arte en la sección 4.1. Luego se evalúan otras características propias de Fajita en la sección 4.2.

4.1. Comparación de herramientas

4.1.1. ROOPS

La necesidad de un terreno común para comparar las distintas herramientas de generación de casos de test llevó a la concepción de la iniciativa ROOPS (Reachability in Object-Oriented ProgramS) [37]. Uno de los aportes de ROOPS es un lenguaje de programación con características comunes a Java y C#. El ROOPS Language es provisto junto con herramientas capaces de hacer traducciones a ambos lenguajes. El lenguaje incluye anotaciones que permiten indicar objetivos alcanzables e inalcanzables dentro del código. Otro de los aportes de la iniciativa es la construcción y el mantenimiento de un repositorio de casos de estudio llamado ROOPS benchmark. El benchmark está constituido por distintos programas con características diversas, con el fin de poder evaluar las fortalezas y debilidades de cada técnica. La evaluación se realiza comparando el cubrimiento de los objetivos logrado por las distintas herramientas.

4.1.2. Herramientas

A continuación se presentan las herramientas elegidas para la comparación:

- **Kiasan:** Desarrollado en la universidad del estado de Kansas, Kiasan realiza una verificación de modelos simbólicos con inicialización perezosa [6] y utiliza el Yices SMT-Solver [38] para la resolución de restricciones. Si bien la herramienta posee un soporte básico de anotaciones JML, actualmente no es suficientemente expresivo como para poder describir invariantes de representación complejos, por lo que en su lugar se utilizan métodos `repOK`.

- **PEX:** Desarrollado por Microsoft, PEX usa ejecución simbólica dinámica [7]. Combina ejecuciones de código con técnicas de análisis estático y usa el Z3 SMT-Solver [39] para resolver las restricciones que surgen a partir de las distintas condiciones de ramificación. PEX también hace uso de los métodos `repOK` y en algunos casos requiere la incorporación de *factories* (clases con la capacidad de generar instancias válidas de la clase bajo test).

Los métodos `repOK` representan el invariante de representación del objeto retornando verdadero cuando una instancia cumple el invariante de la clase. Los métodos `repOK` fueron agregados a las clases disponibles en el benchmark intentando no perjudicar la performance de las herramientas. Por este motivo, si bien no es realista afirmar que cada herramienta fue utilizada al máximo de sus posibilidades, se hizo un esfuerzo consciente en esa dirección.

4.1.3. Configuración

Todos los experimentos fueron realizados utilizando el mismo equipo con las siguientes características:

- **Microprocesador:** Intel Core i5-750 (2.67 GHz)
- **Placa madre:** DP55WB
- **Memoria RAM:** 8 GB 1333 MHz DDR3
- **Sistema operativo:** Microsoft Windows 7 en el caso de PEX y Debian GNU/Linux (versión 6 “squeeze”) en todos los otros casos.

4.1.4. Formato de los resultados

Los resultados presentados indican el índice de cubrimiento obtenido con las distintas herramientas y también se reporta el tiempo de análisis entre paréntesis. En todos los casos el tiempo de ejecución es reportado en segundos y el tiempo de espera máximo es fijado en 1 hora. Las entradas donde la ejecución de una herramienta supera el tiempo de espera son marcadas con TO. Los casos en los que se logra un cubrimiento subóptimo son resaltados.

4.1.5. Caso de estudio: Operaciones sobre tipos primitivos

En esta sección se evalúa el comportamiento de las herramientas ante operaciones con tipos primitivos como el manejo de condiciones aritméticas complejas. Las estructuras de los métodos utilizados son simples (no se utilizan ciclos), la complejidad se encuentra en la evaluación de una condición en un bloque `if-then-else`. Estos casos evalúan, principalmente, la potencia de los mecanismos de resolución de restricciones subyacentes. Sólo se reporta el cubrimiento de los objetivos ya que la simplicidad de la estructura del código hace que los otros criterios no resulten interesantes.

El ROOPS benchmark provee los siguientes casos útiles para esta evaluación:

1. `LinearWithoutOverflow`: Evaluaciones de condiciones con expresiones aritméticas con enteros de 32 bits de precisión.
2. `LongBenchmark`: Evaluación de expresiones aritméticas con enteros de 64 bits de precisión.
3. `FloatBenchmark`: Evaluación de operaciones de punto flotante con números de 32 bits de precisión (IEEE-754)
4. `IntArrayWithoutExceptions`: Uso de arrays con enteros de 32 bits.
5. `IntArrayWithoutExceptionsWithArrayParameters`: Uso de arrays con enteros de 32 bits y expresiones aritméticas.
6. `Objects`: Operaciones primitivas sobre objetos.

En la Figura 4.2 se muestra un método de la clase `LongBenchmark` para dar una intuición de la complejidad de los métodos. Cabe destacar que, en general, la complejidad de los métodos en cada clase es creciente, por lo que el número del método es un índice de su dificultad relativa dentro de la clase a la que pertenece.

```
public void Method1(float i) {
    if (i == i)
        {@goal(0)}
    else
        {@goal(1)}
}
```

Figura 4.1: Ejemplo de evaluación de condiciones con expresiones de punto flotante.

```
public void Method8(long x1, ..., long x15) {
    if (((x1+x2)/239872938472213L) +
        ((x3+x4)/934898348272837L) +
        ((x5+x6)/8947934794879483948L) +
        ((x7+x8)/343847232124345L) +
        ((x9+x10)/9982728762834283L) +
        ((x11+x12)/3879237429742929387L) +
        ((x13+x14+x15)/(-23842938479229293L))
        == 9223372036854775807L)
        {@goal(0)}
    else
        {@goal(1)}
}
```

Figura 4.2: Ejemplo de evaluación de condiciones con aritmética entera de 64 bits.

A continuación se presentan los resultados para los métodos en los que alguna herramienta alcanzó un cubrimiento subóptimo (28 casos), en el resto de los casos todas las herramientas lograron cubrir todos los objetivos (76 casos).

Método	Objetivos	PEX	Kiasan	Fajita
XTimesCLessThanZ	2	2 (1 s)	0 (2 s)	2 (4 s)
XTimesCLessEqualThanZ	2	2 (1 s)	0 (2 s)	2 (3 s)
XTimesCGreaterThanZ	2	2 (1 s)	0 (2 s)	2 (4 s)
XTimesCGreaterEqualThanZ	2	2 (1 s)	0 (2 s)	2 (6 s)
XTimesCEqualsZ	2	2 (1 s)	0 (2 s)	2 (4 s)
TestLinearEquation2	2	2 (2 s)	0 (2 s)	2 (4 s)

Tabla 4.1: Cobertura de objetivos de la clase `LinearWithoutOverflow`

Método	Objetivos	PEX	Kiasan	Fajita
Method1	2	1 (4 s)	2 (2 s)	2 (8 s)
Method2	2	2 (2 s)	0 (2 s)	2 (35 s)
Method3	2	2 (1 s)	0 (2 s)	2 (30 s)
Method4	2	1 (1 s)	2 (2 s)	2 (8 s)
Method5	2	2 (3 s)	0 (2 s)	2 (117 s)
Method6	2	2 (4 s)	0 (2 s)	2 (391 s)
Method7	2	2 (25 s)	0 (2 s)	2 (544 s)
Method8	2	1 (1 s)	2 (2 s)	1 (1682 s)

Tabla 4.2: Cobertura de objetivos de la clase `LongBenchmark`

Método	Objetivos	PEX	Kiasan	Fajita
Method1	2	1 (2 s)	1 (2 s)	2 (4 s)
Method2	2	1 (1 s)	0 (2 s)	2 (6 s)
Method3	2	1 (1 s)	1 (2 s)	2 (6 s)
Method4	2	1 (1 s)	1 (2 s)	2 (6 s)
Method5	2	1 (1 s)	1 (2 s)	2 (6 s)
Method6	2	1 (1 s)	1 (2 s)	2 (6 s)
Method7	2	2 (1 s)	0 (2 s)	2 (9 s)
Method8	2	2 (1 s)	0 (2 s)	2 (8 s)
Method9	2	2 (1 s)	0 (2 s)	2 (50 s)
Method10	2	2 (1 s)	0 (2 s)	2 (97 s)
Method11	2	2 (1 s)	0 (2 s)	2 (115 s)
Method12	2	1 (1 s)	0 (2 s)	2 (3425 s)
Method13	2	1 (1 s)	1 (2 s)	2 (1942 s)

Tabla 4.3: Cobertura de objetivos de la clase `FloatBenchmark`

Método	Objetivos	PEX	Kiasan	Fajita
BigArray	3	3 (2 s)	0 (TO)	3 (8 s)

Tabla 4.4: Cobertura de objetivos de la clase `IntArrayWithoutExceptions`

Las siguientes particularidades fueron encontradas durante la experimentación:

1. Para poder cubrir el objetivo 0 del método `method1` de la clase `FloatBenchmark` (Fig. 4.1) Fajita usa un valor de punto flotante `NaN` (*Not a Number*), que según el estándar 754 de la IEEE no es igual a sí mismo, tanto Kiasan como PEX ignoran este hecho.
2. El método `method8` de la clase `LongBenchmark` (Fig. 4.2) posee un objetivo alcanzable y uno inalcanzable, sin embargo Kiasan reporta la cobertura de ambos. El caso de test generado por Kiasan para cubrir el objetivo inalcanzable (el número 0) es erróneo.

Se observa que en la mayoría de estos casos las herramientas o bien generan la entrada que cubre el objetivo casi inmediatamente, o bien no logran cubrirlo. Se ve, también, que Fajita ofrece un soporte adecuado ante restricciones matemáticas con los tipos de datos primitivos. Los SMT-Solvers usados por PEX y Kiasan escalan mejor a medida que las restricciones crecen en tamaño, lo que se evidencia con menores tiempos de ejecución, pero fallan en alcanzar una buena cobertura con condiciones de complejidad moderada donde surgen las sutilezas de los tipos de datos.

4.1.6. Caso de estudio: Estructuras de datos

En esta sección se muestran los resultados obtenidos con métodos pertenecientes a estructuras de datos, cuya estructura de control es mucho más compleja que en los casos de la sección anterior. Estos casos representan, además, a un conjunto de programas mucho más amplio. Pues se asemejan a otras aplicaciones con fuertes restricciones sobre las relaciones de los objetos en memoria, como por ejemplo lectores de documentos XML.

Se consideraron las siguientes clases pertenecientes al benchmark ROOPS:

1. `SinglyLinkedList`: Una implementación de una lista simplemente enlazada.
2. `DoublyLinkedList`: Una lista circular doblemente enlazada.
3. `NodeCachingLinkedList`: Una lista circular doblemente enlazada con almacenamiento de nodos para reducir el tiempo de asignación y “garbage collection”.
4. `BinarySearchTree`: Una implementación de un árbol de búsqueda binario.
5. `AVLTree`: Un árbol de búsqueda binario tipo AVL que se mantiene balanceado.
6. `BinomialHeap`: Una implementación de una estructura similar a un binary heap que permite realizar una unión más eficientemente.
7. `FibonacciHeap`: Una implementación de una estructura similar a un heap conformado por conjuntos de árboles, que logra un mejor rendimiento amortizado que un binomial heap.
8. `IntRedBlackTreeMap`: Una implementación de un diccionario basado en un red black tree usando para las claves el tipo primitivo `int`.

Como se mencionó en la sección 3.6, para lograr una mayor automatización del proceso es deseable relevar al usuario de la responsabilidad de decidir el scope de análisis a ser utilizado. Para poder utilizar el mecanismo de exploración iterativa del espacio de búsqueda provisto por Fajita, primero es necesario definir la estrategia de exploración. Por lo que antes de comparar a Fajita con las otras herramientas es necesario evaluar distintas estrategias y seleccionar la más eficiente.

La estrategia de exploración se basa en realizar la instrumentación para cada combinación de las variables. Se asigna inicialmente el valor 1 a la variable `objectScope` y se la incrementa en 1 luego de barrer un rango de valores para la variable `loopUnroll`. Se consideran distintas formas de barrido del loop unroll con el objetivo de poder efectuar un análisis comparando los resultados obtenidos. El algoritmo se detiene cuando logra cubrir exhaustivamente el caso de análisis o supera el tiempo de espera.

Se consideran las siguientes estrategias de barrido para cada object scope s :

1. **Barrido completo:** Se utilizan todos los valores entre 1 y s .
2. **Máximo:** Sólo se utiliza el valor s .
3. **Medio:** Sólo se utiliza el valor $\frac{s}{2}$.

Intuitivamente la estrategia de barrido completo parece ser la más exhaustiva. Sin embargo, debemos recordar que los casos con un loop unroll mayor incluyen a los menores y que para cada valor de loop unroll es necesario realizar una instrumentación completa. La ventaja de realizar la exploración paulatinamente es que los “objetivos” cubiertos por iteraciones anteriores son removidos dejando un modelo más simple. Pero por otro lado el costo de realizar la instrumentación para cada valor puede resultar demasiado grande.

Para evitar el problema de la estrategia anterior se propone la estrategia que sólo contempla como valor para el loop unroll el mismo valor que para el object scope. Este acercamiento permite explorar las dos dimensiones simultáneamente sin el costo de las etapas intermedias requeridas por el mecanismo anterior. No obstante, el crecimiento del modelo (en variables proposicionales) al aumentar ambos valores puede impactar en la eficiencia del SAT-Solver.

Durante el período de prueba de la herramienta se observó que muy frecuentemente los valores óptimos se obtenían considerando un loop unroll al rededor de la mitad del object scope. Para verificar esta observación se incluye la tercer estrategia de barrido.

La Tabla 4.5 reporta para cada método del benchmark el tiempo insumido por cada estrategia de barrido. Si bien el tiempo de precomputar las cotas superiores ajustadas no está incluido en el tiempo reportado, para estos experimentos es un promedio de 70 segundos (el detalle puede encontrarse en [40]). Se resaltan los casos donde se obtiene un desempeño menor que el mejor caso.

	Completo	Máximo	Medio
SinglyLinkedList			
contains	2 s	2 s	2 s
insertBack	3 s	2 s	2 s
remove	4 s	5 s	4 s
DoublyLinkedList			
contains	2 s	2 s	2 s
addLast	3 s	3 s	3 s
remove	5 s	4 s	3 s
NodeCachingLinkedList			
contains	2 s	3 s	2 s
setMaxCache	TO	TO	TO
addLast	2 s	2 s	2 s
removeIndex	TO	TO	TO
SearchTree			
find	3 s	4 s	3 s
add	5 s	4 s	7 s
remove	TO	TO	TO
AVLTree			
findNode	10 s	6 s	4 s
findMax	11 s	6 s	6 s
findMin	9 s	7 s	6 s
BinomialHeap			
findMin	12 s	6 s	8 s
decreaseKey	11 s	7 s	6 s
extractMin	216 s	120 s	64 s
insert	18 s	16 s	10 s
FibonacciHeap			
minimum	6 s	4 s	4 s
insertNode	6 s	5 s	3 s
removeMin	TO	TO	TO
IntRedBlackTreeMap			
put	6 s	5 s	4 s
remove	TO	TO	TO

Tabla 4.5: Resultados sobre el ROOPS benchmark usando exploración iterativa.

En los resultados obtenidos puede verse que la estrategia menos eficiente es la de exploración completa, lo que puede deberse a que posee muchas más etapas intermedias que las demás. Tal como se había observado al utilizar el valor medio del intervalo se logran, en general, los mejores resultados. Sin embargo, esto puede deberse a que

los algoritmos analizados tienen, en muchos casos, un orden de complejidad menor que lineal, por lo que rara vez resulta necesario realizar una cantidad de iteraciones igual a la cantidad de objetos contenidos en las estructuras. En los siguientes experimentos se utiliza Fajita con la estrategia de exploración que utiliza el valor medio.

Cabe destacar que los casos en los que la herramienta agotó el tiempo de espera se determinó, mediante una inspección manual, que los métodos poseían objetivos inalcanzables. Para poder hacer una evaluación más completa en el siguiente experimento se reporta el tiempo insumido en cubrir los objetivos alcanzables.

Tanto Kiasan como PEX poseen un menor grado de automatización que Fajita. En ambos casos fue necesario hacer un ajuste manual de distintos parámetros para intentar utilizar la herramienta en las mejores condiciones.

- Para cada experimento realizado con Kiasan se buscó la combinación de parámetros (cantidad de iteraciones y llamadas a funciones) que produjera la mejor cobertura al mismo tiempo que minimizara el tiempo de análisis. Sólo se considera el tiempo de la configuración más exitosa.
- Al utilizar PEX se configuraron los parámetros de tiempo máximo de resolución de restricciones en 1800s y de tiempo de espera total en 3600s. En el caso en el que una cota (como la cantidad máxima de ramificaciones) hubiera sido excedida antes que todos los objetivos y ramificaciones hayan sido cubiertas, estas fueron incrementadas y el experimento repetido. Sólo se reportan las ejecuciones con los mejores resultados.

Los datos de la comparación de las herramientas se presentan en una tabla con el siguiente formato:

- La columna $\#O$ indica la cantidad de objetivos marcados en el código. En el caso de que haya objetivos inalcanzables se indican los alcanzables sobre los totales.
- La columna $\#D$ indica la cantidad de decisiones en el código. En el caso de que haya ejes inalcanzables se indican los alcanzables sobre los totales.
- La columna *Kiasan* y *PEX* reportan la cantidad de objetivos cubiertos (O), la cantidad de ramas cubiertas (D) y el tiempo requerido por el experimento (T) con el siguiente formato: $O, D (T)$.
- Los resultados obtenidos por *Fajita* están presentados en dos columnas, la primera reporta los objetivos cubiertos (O) y el tiempo (T), mientras que la segunda reporta las ramificaciones cubiertas (D) y el tiempo correspondiente a este caso (T). En los casos que haya objetivos inalcanzables se muestra el tiempo insumido en alcanzar el máximo cubrimiento obtenido y se lo resalta en negrita ya que en realidad la herramienta continua hasta agotar el tiempo de espera.

A continuación, en la Tabla 4.6 se presentan los resultados obtenidos en este experimento. Luego se muestra un resumen del porcentaje de cobertura logrado por cada herramienta en las Figuras 4.3, 4.4, 4.5 y 4.6.

	#O	#D	PEX O, D (T)	Kiasan O, D (T)	Fajita	
					O (T)	D (T)
SinglyLinkedList						
contains	7	12	1, 3 (7 s)	7, 12 (4 s)	7 (2 s)	12 (2 s)
insertBack	4	6	2, 3 (1 s)	4, 6 (3 s)	4 (2 s)	6 (2 s)
remove	7	10	2, 4 (1 s)	7, 10 (3 s)	7 (4 s)	10 (6 s)
DoblyLinkedList						
contains	3	10	0, 1 (6 s)	3, 10 (3 s)	3 (2 s)	10 (3 s)
addLast	3	2	0, 1 (9 s)	3, 2 (3 s)	3 (3 s)	2 (3 s)
remove	10	14	10, 13 (6 s)	8, 10 (3 s)	10 (3 s)	14 (2 s)
NodeCachingLinkedList						
contains	4	10	4, 3 (1 s)	4, 10 (9 s)	4 (2 s)	10 (2 s)
setMaxCache	4/5	7/8	5, 8 (9 s)	5, 7 (4 s)	4 (20 s)	7 (22 s)
addLast	4	6	4, 6 (12 s)	4, 6 (4 s)	4 (2 s)	6 (2 s)
removeIndex	10/11	19/20	7, 7 (9 s)	3, 5 (3 s)	10 (126 s)	19 (117 s)
SearchTree						
find	5	8	5, 8 (1 s)	5, 8 (3 s)	5 (3 s)	8 (2 s)
add	9	12	9, 12 (4 s)	9, 12 (3 s)	9 (7 s)	12 (11 s)
remove	15/17	17/20	15, 17 (3515 s)	15, 17 (6 s)	15 (9 s)	17 (145 s)
AVLTree						
findNode	5	8	5, 8 (5 s)	5, 8 (3 s)	5 (4 s)	8 (4 s)
findMax	3	6	2, 5 (216 s)	3, 6 (3 s)	3 (6 s)	6 (5 s)
findMin	3	6	2, 5 (93 s)	3, 6 (3 s)	3 (6 s)	6 (6 s)
BinomialHeap						
findMin	3	6	2, 5 (5 s)	3, 6 (3 s)	3 (8 s)	6 (9 s)
decreaseKey	3	6	3, 6 (6 s)	3, 6 (3 s)	3 (6 s)	6 (6 s)
extractMin	10	46	6, 16 (8 s)	10, 46 (6 s)	10 (64 s)	46 (81 s)
insert	22	28	18, 23 (TO)	18, 23 (TO)	22 (10 s)	28 (16 s)
FibonacciHeap						
minimum	1	2	1, 2 (1 s)	1, 2 (3 s)	1 (4 s)	2 (2 s)
insertNode	5	6	5, 6 (4 s)	5, 6 (3 s)	5 (3 s)	6 (4 s)
removeMin	20/25	33/38	20, 33 (TO)	20, 33 (4 s)	20 (565 s)	33 (609 s)
IntRedBlackTreeMap						
put	24	38	24, 38 (3 s)	22, 34 (TO)	24 (4 s)	38 (5 s)
remove	27/28	65/66	27, 65 (147 s)	21, 54 (875 s)	27 (39 s)	65 (45 s)

Tabla 4.6: Resultados obtenidos para el caso de estudio de estructuras de datos.

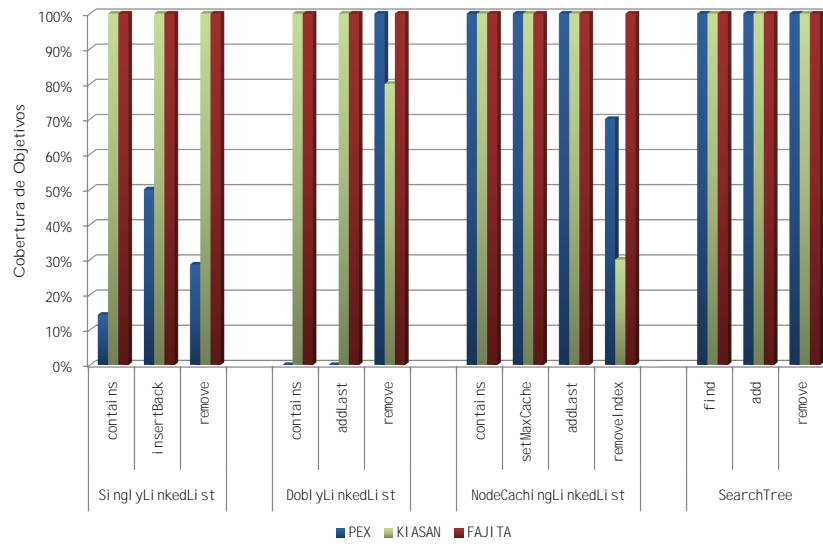


Figura 4.3: Cobertura de objetivos (1/2)

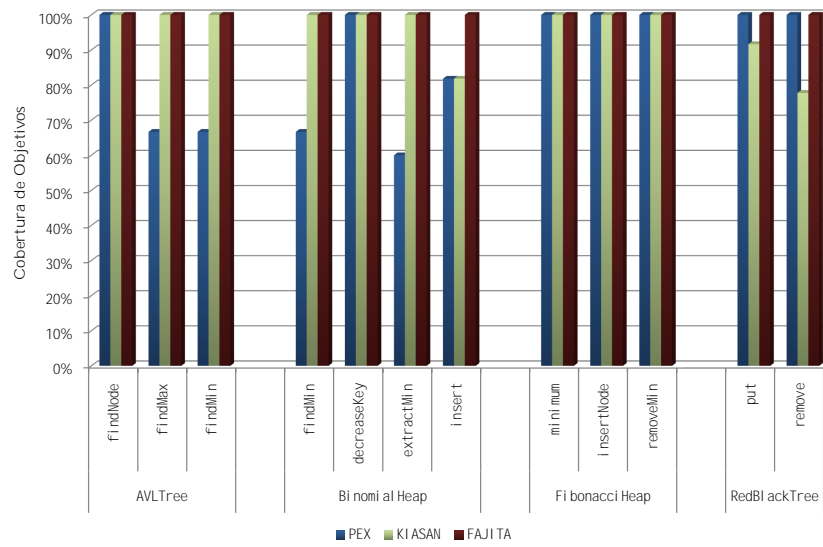


Figura 4.4: Cobertura de objetivos (2/2)

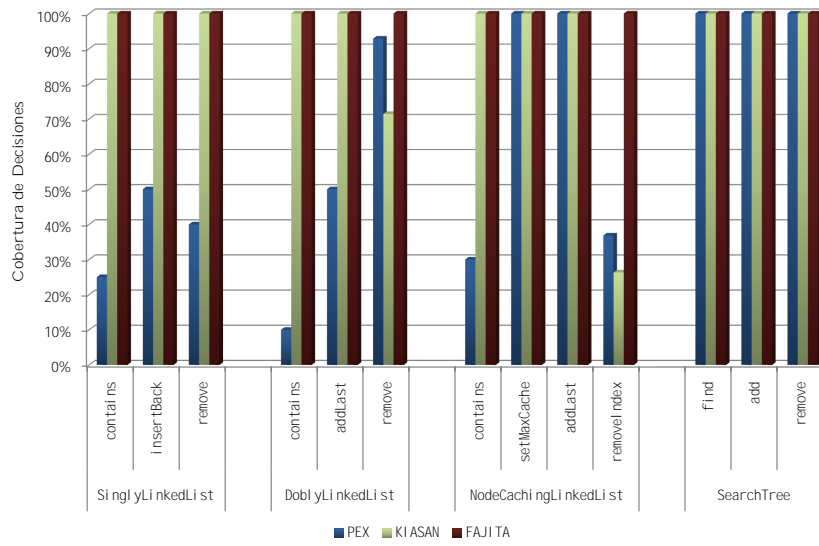


Figura 4.5: Cobertura de decisiones (1/2)

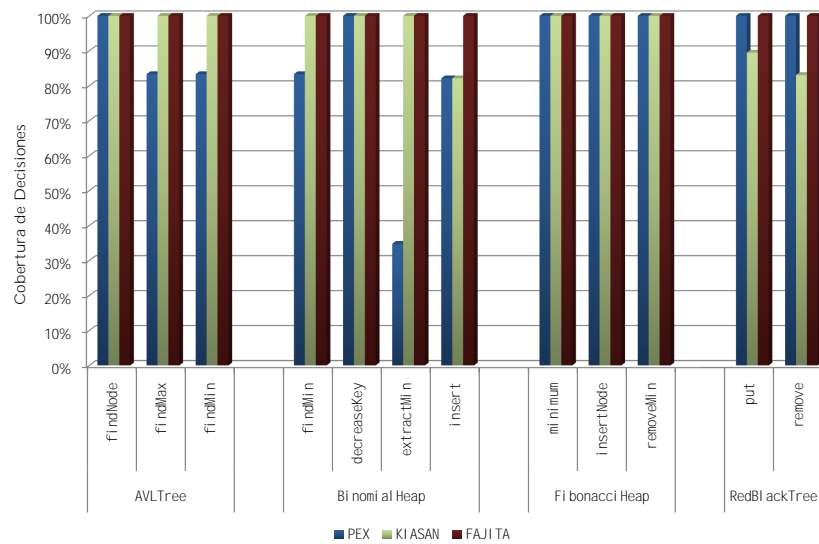


Figura 4.6: Cobertura de decisiones (2/2)

En este caso debe mencionarse que en el método `setMaxCacheSize` de la clase `NodeCachingLinkedList` tanto PEX como Kiasan reportan 5 objetivos cubiertos, mientras que Fajita 4. Una inspección manual permitió confirmar que el objetivo que Fajita no logra cubrir se encuentra dentro de un ciclo infinito. Si bien Kiasan reporta el objetivo como cubierto no genera un caso de test para demostrarlo. Por el contrario, PEX genera una entrada que lo hace. Es discutible si tal entrada debe formar parte de un test suite. En este trabajo se considera que no, dado que al no terminar no permite verificar la correctitud del resultado. La traducción del código a una fórmula proposicional utilizada por Fajita previene explícitamente la generación de este tipo de casos.

Los resultados muestran que Fajita es la única herramienta que logró generar un conjunto de casos de test de cobertura óptima tanto de objetivos como de decisiones. De los 25 métodos bajo análisis Kiasan no logró el cubrimiento óptimo en 6 casos y PEX en 16.

Al mismo tiempo, se observa que en algunos casos Fajita requiere un tiempo mayor que las otras herramientas. Es razonable suponer que el crecimiento de la fórmula proposicional utilizada por el SAT-Solver en cada paso es la causa del problema en la escalabilidad. Sin embargo, una inspección minuciosa de los casos de mayor duración reveló que gran parte del tiempo se utiliza para la traducción del modelo Alloy a Kodkod. Esto se debe a que al cambiar el valor del `object scope` o del `loop unroll` es necesario realizar nuevamente todas las traducciones desde el código fuente Java, por lo que no se reutilizan las traducciones anteriores sino que se regeneran desde cero. La técnica de exploración iterativa incrementa las capacidades de Fajita, pero para escalar a casos de mayor tamaño es necesario disponer de herramientas que permitan extender eficientemente las traducciones realizadas en iteraciones anteriores, actualizando los valores de `object scope` y `loop unroll`.

Observando los casos de test generados puede verse que los métodos para los que la herramienta termina rápidamente son cubiertos con tests simples, conformados por menos de 3 objetos. Por otro lado, los casos en donde es necesario un tiempo mayor pueden encontrarse casos de tests complejos formados por hasta 22 objetos. Resulta evidente que el mecanismo de exploración del `scope` de análisis resulta costoso para los casos en donde es necesario considerar valores grandes.

4.2. Cobertura de caja gris

En esta sección se comparan los resultados obtenidos con criterios de caja cerrada y de caja abierta con la propuesta de la sección 3.5, que resulta de una combinación de ambos acercamientos denominado normalmente cobertura de caja gris. Recordemos que el criterio de caja cerrada utilizado se basa en la suposición que en una clase bien diseñada los métodos booleanos sin argumentos dan suficiente información como para lograr una buena partición del espacio de estados. Desgraciadamente las clases Java disponibles en el benchmark ROOPS no respetan este principio, exponiendo en la mayoría de los casos ningún método booleano. Afortunadamente la clase `NodeCachingLinkedList` posee las características necesarias para efectuar las pruebas.

El experimento consta de la ejecución de Fajita con el criterio de métodos booleanos junto con la instrumentación del criterio de decisiones. De esta forma al ejecutar el criterio de caja cerrada se obtiene un conjunto de tests del cual se sabe, además, su índice de cobertura según el criterio de decisiones. Luego se ejecuta cada uno de los métodos bajo análisis removiendo los objetivos correspondientes a los ejes cubiertos por la primer iteración. Cabe destacar que la cobertura de métodos booleanos es ejecutada una única vez, por lo que es amortizado al utilizar su resultado en las iteraciones a caja abierta.

La clase `NodeCachingLinkedList` contiene la implementación de una lista circular doblemente enlazada con almacenamiento de nodos en una memoria cache para reducir el tiempo de aloación y “garbage collection”. Para simplificar el experimento se considerará únicamente una lista cuyo tamaño de cache sea 5. La interfaz de esta clase posee tres métodos booleanos:

- `isEmpty()` que dice si la lista no contiene ningún elemento.
- `isCacheEmpty()` que indica si no hay nodos almacenados en cache.
- `isCacheFull()` que indica si se tiene guardada la cantidad máxima de nodos.

La Tabla 4.7 muestra la cobertura de métodos booleanos para los distintos tamaños de scope (los casos no mencionados no aumentan la cobertura con respecto al caso anterior).

Scope	Particiones cubiertas
1	2 (3 s)
2	4 (3 s)
5	8 (5 s)

Tabla 4.7: Cobertura de métodos booleanos

Se ve que basta utilizar un object scope de 5 para cubrir las 8 clases de equivalencia. Por cada clase se genera un caso de test, la Tabla 4.8 muestra las ramificaciones cubiertas por este conjunto de tests respetando el siguiente formato:

- La columna *Método* muestra el nombre del método del cual se reporta el cubrimiento de decisiones de la clase `NodeCachingLinkedList`.
- La columna *#Ramas* indica la cantidad total de ramificaciones alcanzables por las distintas ejecuciones del método.
- La columna *CMB* reporta la cantidad de ramas cubiertas por los casos de test generados con el criterio de métodos booleanos.

- La columna *CD* reporta la cantidad de ramas adicionales cubiertas siguiendo el criterio de decisiones, teniendo en cuenta que a diferencia de lo que sucede en la sección 4.1.6, algunos ejes se consideran ya cubiertos por la iteración realizada a caja cerrada. Se indica el tiempo requerido por la ejecución del criterio de decisiones y el total que surge a partir de adicionar el tiempo de ejecución del criterio de caja cerrada.
- La columna *D* muestra los tiempos obtenidos en el experimento anterior, en donde se buscaba cubrir todos las decisiones utilizando únicamente el criterio de caja abierta, con el objetivo de que puedan comprarse los resultados rápidamente.
- Finalmente la última fila muestra el tiempo total requerido para la generación de casos de test para los tres métodos, considerando amortizado entre los tres casos el tiempo requerido por la ejecución del criterio de caja cerrada.

Método	#Ramas	CMB	CD	D
contains	10	9	1 (2 s / 7 s)	2 s
addLast	6	6	-	2 s
removeIndex	19	10	9 (94 s / 99 s)	117 s
		Total	102 s	121 s

Tabla 4.8: Decisiones cubiertas por la iteración de caja cerrada

A partir de los resultados se puede apreciar que a pesar de lograr una cobertura de métodos booleanos total, la cobertura de decisiones no siempre es exhaustiva, por lo que el criterio de caja gris recae en el mecanismo de caja abierta. La segunda iteración resulta más rápida que la ejecución del método de caja abierta del experimento anterior, dado que algunas ramas se consideran cubiertas. Sin embargo, al agregar el tiempo de la iteración de caja cerrada el tiempo total resulta superior. No obstante, dado que el tiempo de la ejecución del criterio de caja cerrada puede amortizarse, los casos en donde se analizan múltiples métodos pertenecientes a la misma clase el tiempo total disminuye.

Por otro lado, los tests generados por la iteración de caja cerrada resultan más fáciles de entender, dado que se pueden expresar en términos de la interfaz definida por el programador. Al ser un conjunto de casos de test común a todos los métodos, la cantidad total de casos necesarios para analizar una clase entera puede verse disminuida al utilizar este acercamiento.

Podría agregarse a la arquitectura de Fajita un repositorio de casos de test obtenidos mediante el mecanismo de caja cerrada, de forma tal que cada vez que se requiera generar un conjunto de tests para una cobertura de decisiones puedan reutilizarse los resultados allí almacenados. Esto también podría utilizarse en combinación con otros criterios de caja abierta. Sin embargo, si bien la cobertura de métodos booleanos no cambia mientras la interfaz de la clase y su invariante no sea alterado, los objetivos de la cobertura de caja abierta sí pueden cambiar junto con el código. Por lo tanto, para

que tal repositorio sea útil es necesario introducir, también, mecanismos que permitan decidir qué objetivos son cubiertos por el conjunto de tests base a medida que el código cambia. Estos mecanismos no necesariamente incluyen una instrumentación como la evaluada en esta sección, sino que pueden estar basados en los resultados de las ejecuciones de la batería de tests.

En conclusión, este criterio logra la misma cobertura que el criterio de decisiones pero potencialmente generando un conjunto de tests más claro para el programador, aunque para lograrlo puede requerir más tiempo. El tiempo adicional requerido por este acercamiento puede amortizarse si se desea generar casos de test para varios métodos de la clase, pudiendo mejorar el desempeño global. Esto podría reducir, además, el esfuerzo realizado por el operador en determinar si los resultados de los tests son correctos o no.

Capítulo 5

Conclusiones

En este trabajo se ha demostrado empíricamente que el uso de SAT-Solving incremental representa una técnica valiosa, capaz de enfrentar eficientemente el problema de la generación automática de casos de test.

Las herramientas de generación de casos de test basadas en verificación de modelos o ejecución simbólica están sujetas al problema de explosión exponencial de caminos. Por lo tanto, la exploración eficiente del espacio de estados inducido por la estructura de un programa es esencial. La combinación de las cotas superiores ajustadas y la incorporación de cláusulas en el proceso de SAT-Solving de forma incremental, permite reducir el espacio de búsqueda concentrándose en el criterio de selección de casos de test elegido.

Cabe destacar que la sistematización de la exploración del scope de análisis es un problema recurrente en la literatura de la generación automática de casos de test. Esto se debe principalmente a que resulta difícil lograr una estrategia automática que escale a medida que crece el tamaño de los problemas analizados. Gracias a los mecanismos de exploración iterativa del espacio de búsqueda Fajita ofrece un alto grado de automatización, que si bien puede resultar menos eficiente que la elección de parámetros mediante un análisis minucioso, representa una importante contribución.

Una característica distintiva de Fajita es el amplio espectro de escenarios donde logra un buen desempeño. Las herramientas basadas en SMT-Solving [7], pueden generar casos de test para programas con aritmética entera y estructuras simples, pero el mecanismo subyacente no es capaz de manejar estructuras de datos complejas eficientemente. Otras técnicas, como las utilizadas por Kiasan [6] y Korat [8] logran una generación eficiente de casos de test para estructuras de datos complejas, pero dependen de código adicional con descripciones operacionales para la construcción y validación de instancias (por ejemplo mediante el uso de predicados repOK). Fajita logra manejar una familia amplia de tipos de datos numéricos y genera estructuras válidas bajo restricciones complejas utilizando especificaciones declarativas.

Otro aspecto destacable de Fajita es su flexibilidad, reflejado en el hecho de que admite múltiples criterios tanto provenientes de un acercamiento de testing de caja

cerrada como de caja abierta. Muchas herramientas, en especial las basadas en verificación de modelos como TestEra [41], sólo pueden manejar criterios de caja abierta, dado que se basan fuertemente en el código analizado.

Se puede observar que actualmente los SMT-Solvers logran resolver expresiones de aritmética entera muy rápidamente. No obstante, no sucede lo mismo para las expresiones de punto flotante. El acercamiento de Fajita, si bien logra una performance menor, es capaz de contemplar ambos casos por igual. La eficiencia de Fajita en este aspecto podría mejorarse con una implementación de un SAT-Solver consciente de las operaciones matemáticas primitivas, lo que representa una nueva línea de investigación dado que no existen implementaciones disponibles. Otra optimización que puede hacerse a este nivel es la reutilización de las representaciones intermedias obtenidas en las distintas iteraciones realizadas al explorar el espacio de estados, mejorando así la estrategia de exploración iterativa propuesta.

Si bien las optimizaciones realizadas tanto en Fajita como en TACO logran reducir significativamente el espacio de búsqueda, el acercamiento basado en exploración exhaustiva mantiene una progresión exponencial en peor caso. Gracias al tamaño acotado de los casos de test de unidad, puede apreciarse su potencial. Sin embargo, como ya fue mencionado, este acercamiento no escala a la generación de tests de sistema. Al ser una técnica novedosa es necesaria una mayor investigación para trasladar la técnica a los tamaños de los grandes sistemas de software. Para ello también es necesario considerar ejecuciones concurrentes y otras características comunes de los sistemas distribuidos. Si bien estos problemas no son triviales la técnica presentada posee el potencial para, a futuro, resultar exitosa en estos escenarios.

Bibliografía

- [1] Myers and Glenford J. *The Art of Software Testing*. Wiley, New York, 1979.
- [2] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pp. 425–434, New York, NY, USA, 2007. ACM.
- [3] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pp. 75–84, Minneapolis, MN, USA, May 23–25, 2007.
- [4] Lisa Ling Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *Proceedings of the 1st international conference on Tests and proofs*, TAP'07, pp. 114–130, Berlin, Heidelberg, 2007. Springer-Verlag.
- [5] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in uditá. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 225–234, New York, NY, USA, 2010. ACM.
- [6] Xianghua Deng, Robby, and John Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp. 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
- [7] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. TAP'08, pp. 134–153, 2008.
- [8] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pp. 123–133, New York, NY, USA, 2002. ACM.
- [9] William E. Perry. *A standard for testing application software*. Auerbach, 1991.

-
- [10] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.
- [11] W. E. Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*. McGrall-Hill Book Co, New York, 1987.
- [12] David Griffioen and Marieke Huisman. A comparison of pvs and isabelle/hol. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pp. 123–142, London, UK, UK, 1998. Springer-Verlag.
- [13] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.
- [14] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pp. 130–145. Springer, 2008.
- [15] Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'03*, pp. 505–520, Berlin, Heidelberg, 2003. Springer-Verlag.
- [16] Mana Taghdiri. *Automating modular program verification by refining specifications*. PhD thesis, Cambridge, MA, USA, 2008. Adviser-Jackson, Daniel N.
- [17] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07*, pp. 195–204, New York, NY, USA, 2007. ACM.
- [18] G.D. Dennis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, and Computer Science. *A relational framework for bounded program verification*. PhD thesis, 2009.
- [19] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing, STOC '71*, pp. 151–158, New York, NY, USA, 1971. ACM.
- [20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02*, pp. 234–245, New York, NY, USA, 2002. ACM.
- [21] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, 2nd edition, 2002.
- [22] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

-
- [23] Michal Young and Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.
- [24] Juan Pablo Galeotti. *Verificación de Software usando Alloy*. PhD thesis, Buenos Aires, Argentina, 2010. Director-Frias, Marcelo Fabián.
- [25] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.
- [26] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. Dynalloy: upgrading alloy with actions. In *Proceedings of the 27th international conference on Software engineering, ICSE '05*, pp. 442–451, New York, NY, USA, 2005. ACM.
- [27] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic logic*. Foundations of computing. MIT Press, 2000.
- [28] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *Proceedings of the 2006 international symposium on Software testing and analysis, ISSA '06*, pp. 109–120, New York, NY, USA, 2006. ACM.
- [29] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, New York, NY, USA, 1988.
- [30] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer, 2003.
- [31] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference, DAC '01*, pp. 530–535, New York, NY, USA, 2001. ACM.
- [32] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55:101–122, February 2009.
- [33] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'07*, pp. 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.
- [34] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, ISSA '00*, pp. 14–25, New York, NY, USA, 2000. ACM.
- [35] A. Rtc. Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B*, 1992.
- [36] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 281–290. ACM, May 2008.

- [37] <http://code.google.com/p/roops/>.
- [38] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pp. 81–94, 2006.
- [39] Leonardo De Moura and Nikolaj Björner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08/ETAPS'08*, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [40] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nicolas Rosner. Analysis of invariants for efficient bounded verification. In *Proceedings of ISSTA 2010*, pp. 25–36, New York, NY, USA, 2010. ACM.
- [41] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE international conference on Automated software engineering, ASE '01*, pp. 22–, Washington, DC, USA, 2001. IEEE Computer Society.