Generación automática de casos de test para criterios de código avanzados

Instituto Tecnológico de Buenos Aires

Ingeniería Informática

Federico Homovc (50418)
Esteban Pintos (51048)
Matías De Santi (51051)

Agosto 2014

Generación automática de casos de test para criterios de código avanzados

Abstract

El testing de software es una de las tecnologías más utilizadas en el análisis y validación de sistemas. El mismo consiste en ejecutar el sistema desarrollado sobre inputs particulares, y verificar que los outputs producidos se correspondan con lo esperado.

La generación automática de inputs para testeo de software es un área de investigación y desarrollo muy activa, con conferencias de primer nivel que dedican sesiones a la misma (por ejemplo ICST –International Conference on Software Testing-, ICSE -International Conference on Software Engineering-, ASE –Automated Software Engineering-, lo hacen).

En el marco del testeo de software existen diversos criterios de coberturas de código que sirven para determinar la calidad de un conjunto de inputs de testeo. Por ejemplo, la cobertura de sentencias requiere que cada sentencia del programa sea ejecutada al menos una vez por algún test. La herramienta FAJITA [1] permite al usuario elegir un criterio de cobertura y llevarlo a cabo. Actualmente FAJITA permite utilizar los criterios de cobertura de sentencias, de condiciones y de objetivos, pero existen otros que aún no son soportados. El objetivo de este proyecto es entonces extender el portfolio de criterios de cobertura ofrecidos por dicha herramienta.

Índice general

Abstract II

1. Introducción **1**

1.1. Testing 2

1.2. Fajita 3

1.2.1. Instrumentación de código en Fajita 4

1.3. TACO 6

2. Criterios de cobertura existentes en FAJITA **7**

2.1. Cobertura de objetivos 7

2.2. Cobertura de decisiones 8

3. Criterios de cobertura implementados **10**

3.1. Cobertura de múltiples condiciones 10

3.1.1. Implementación 12

3.2. K-Cobertura de caminos 12

3.2.1. Implementación 14

3.3. Cobertura de pares definición-uso 17

3.3.1. Implementación 18

4. Evaluación y Resultados **20**

4.1. Casos de Estudio: Estructura de Datos 20

4.2. Herramientas Utilizadas 20

4.3. Configuración 21

4.4. Formato de los resultados 21

4.5. Evaluación 21
4.6. Resultados 23
4.6.1. Resultados para cobertura de pares definición-uso 23
4.6.2. Resultados para cobertura de múltiples condiciones 23
4.6.3. Resultados para K-Cobertura de caminos 24
4.6.4. Resultados generales 24

5. Conclusiones **26**
5.1. Conclusiones 26

Bibliografía **27**

A **29**

A. Resultados **29**
A.1. Resultados 29

Índice de figuras

1.1. Esquema de la arquitectura de Fajita 5

Índice de cuadros

A.1. Tabla 1: Resultados para Cobertura de pares definición-uso 30
A.2. Tabla 2: Resultados para Cobertura de múltiples condiciones 31
A.3. Tabla 3: Resultados para K-Cobertura de caminos. 32

Capítulo 1

Introducción

El software es un componente clave en muchos de los dispositivos y sistemas que integran nuestra sociedad. Define el comportamiento de los routers de una red, de las redes financieras, redes de conmutación telefónica, la Web y otras infraestructuras de la vida moderna. El software es un componente esencial de las aplicaciones integradas que controlan sistemas críticos tales como aviones, naves espaciales, y sistemas de control del tráfico aéreo, así como los aparatos cotidianos, tales como relojes, hornos, coches, reproductores de DVD, puertas de garajes, teléfonos celulares, y mandos a distancia.

Actualmente el humano hace uso del software cada vez con más frecuencia. Dichas funciones que están automatizadas pueden tener fallas que, si no están controladas, pueden llevar a situaciones inesperadas que pueden resultar en una catástrofe de gran costo.

A través de los años se fue instalando con mayor firmeza la necesidad de tener el software controlado mediante testing. Es por esto que han surgido diferentes metodologías para llevar a cabo este proceso. La gran diversidad de las mismas surge debido al costo de cada una, basado en diferentes fundamentos. El presente trabajo cubre la implementación de tres nuevos criterios de cobertura para la herramienta Fajita: All Definition-Uses, Multiple Condition Coverage y k-cobertura de caminos.

1.1. Testing

El testing de software es un proceso que tiene como objetivo presentar información sobre la calidad del producto. Es decir, es el proceso de verificar que un programa hace lo que debería hacer. El mismo aporta:

- Calidad
- Disminución de costos
- Reducción de riesgos
- Optimización de recursos

Una limitación fundamental del testing es que mientras un test que falla (retorna un resultado que no se corresponde con lo esperado) es suficiente para determinar que existe una falla en el código bajo análisis, el tener una cantidad de tests que no fallan solo nos permite concluir que el código validado es correcto con respecto a esos inputs, es decir, el testing puede probar la presencia de error pero no la ausencia de ellos.

La generación automática de inputs es fundamental, pues testing es la técnica de validación de software más utilizada y puede llegar a insumir un 40 % del costo del desarrollo de un proyecto de software. Luego, el reemplazar la generación manual de tests por inputs generados automáticamente permite no solo reducir los costos de desarrollo de manera significativa, sino también generar conjuntos de inputs que garantizan la satisfacción de criterios de cobertura.

Entre las herramientas que se destacan en este área se encuentran aquellas que se basan en ejecución simbólica y SMT-solving (por ejemplo PEX [5]), aquellas basadas en ejecución simbólica guiada por model checking, y otras basadas en métodos aleatorios como por ejemplo Autotest [6] o Randoop [7]. También existen herramientas que generan inputs a partir de especificaciones de los invariantes de las clases, como por ejemplo TestEra [8] que lo hace a partir de invariantes descritos de forma declarativa en el lenguaje Alloy [9]), o Korat [10] (que utiliza invariantes de la clase descritos por medio de un método Java).

En general todas estas herramientas tienen como limitación la dificultad para generar tests para código que utilizan estructuras de datos complejas (árboles balanceados, por ejemplo). Fajita [4], en cambio, es particularmente apropiada en dicho contexto de uso, como muestran los experimentos presentados en [1]. Sin embargo, los criterios de cobertura utilizados por Fajita son pocos y deben ser extendidos.

1.2. Fajita

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 consiste en examinar exhaustivamente todas las posibles ejecuciones dentro de un espacio finito dado por una cota para el tamaño o de la memoria a utilizar y un número máximo de iteraciones. Dentro de este espacio se busca alguna traza de ejecución que viole una especificación que fue dada en un lenguaje formal.

Para poder definir esta especificación se utiliza JML (Java Modeling Language) [13]. JML provee anotaciones que permiten definir contratos e invariantes. Además, se pueden especificar pre y post condiciones para la ejecución de un método. Por ejemplo, utilizando JML puede definirse que cualquier instancia de una lista simplemente encadenada no puede contener ciclos.

Fajita hace un uso intensivo de las herramientas TACO y Alloy.

La forma de trabajo de Fajita, esqumatizada en la figura 1.1, 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. En este paso es en donde se centra este trabajo y será detallado en las siguientes secciones.

2. El resultado de la instrumentación del código es utilizado como entrada de TACO, que traduce el código y especificaciones a un lenguaje formal. Es con esta traducción que se puede aplicar la técnica de verificación acotada. La

salida de TACO es un modelo formal expresado en el lenguaje de modelado Alloy, que puede ser analizado con un SAT-solver.

3. Luego, el modelo Alloy generado por TACO se modifica para poder incluir restricciones adicionales para poder aprovechar las características de los SAT-solvers incrementales.

4. El modelo generado es luego utilizado como entrada para la herramienta Alloy. Esta herramienta retorna una valuación que satisface el modelo o un mensaje (UNSAT) indicando que no se pudo encontrar ninguna valuación para el scope analizado. Es importante destacar que obtener el mensaje UNSAT no quiere decir que el modelo sea insatisfacible, si no que quiere decir que no se pudo encontrar una valuación que lo satisfaga dentro del scope dado. Si el modelo es insatisfacible, entonces es imposible encontrar una valuación que lo satisfaga en un scope infinito.

5. Cada solución encontrada por Alloy es luego interpretada por Fajita, quien genera un caso de test para dicha valuación y agrega restricciónes al modelo para que no vuelva a obtener la misma solución.

1.2.1. Instrumentación de código en Fajita

Como se mencionó anteriormente, Fajita realiza un preprocesamiento del código antes de envíarselo a TACO. Dicho preprocesamiento consiste en la instrumentación del código para poder obtener el criterio de cobertura necesario.

Al instrumentar el código se agregan unas variables booleanas con un nombre especial: `roops_goal_x`, donde x es el número de goal. Son estas variables las que el SAT-Solver intenta hacer verdaderas mediante la obtención de una valuación.

En el capítulo 2 se mostrarán algunos ejemplos de esto para poder ilustrar el funcionamiento.

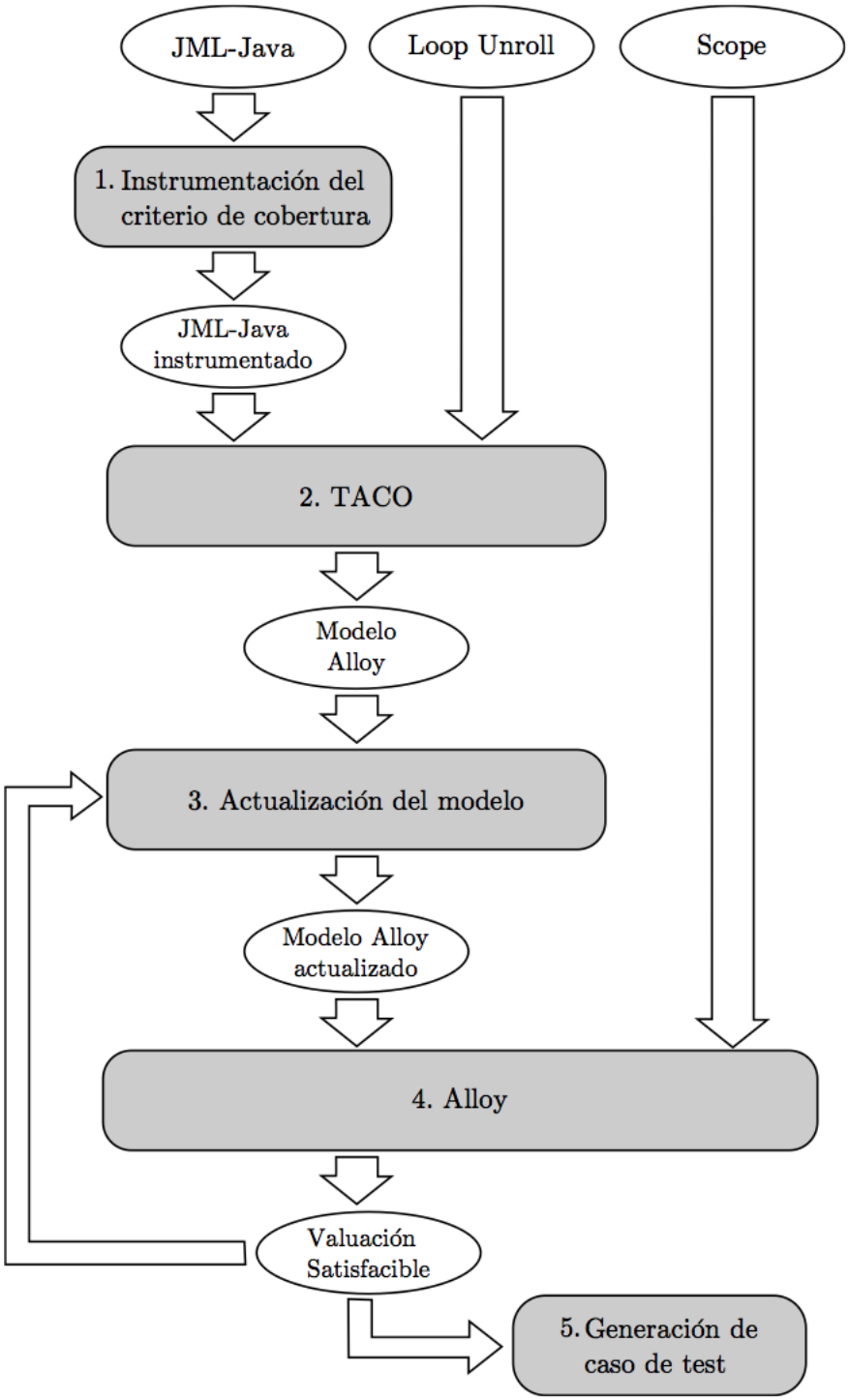

Figura 1.1: Esquema de la arquitectura de Fajita

[fig:arqfajita]

1.3. TACO

TACO (Translation of Annotated COde) es una herramienta que traduce código fuente Java anotado con una especificación JML en un problema SAT. Luego usa Alloy como lenguaje intermedio para expresar el problema SAT.

Para poder lograr la conversión entre el código Java y el modelo Alloy, TACO utiliza la herramienta DynAlloy, una extensión de Alloy. TACO usa una herramienta llamada DynJML que permite pasar de un código Java a un modelo DynAlloy. Luego DynAlloy genera un modelo Alloy, siempre preservando la semántica del programa original.

Para poder manejar los ciclos dentro del programa original, se restringe el tamaño máximo de las trazas de ejecución. Esto se realiza replicando el código de los ciclos la cantidad de veces que se requiera. Esta técnica se conoce como loop unroll.

Finalmente el modelo Alloy es traducido a una fórmula SAT y esta es resuelta por el SAT solver.

Capítulo 2

Criterios de cobertura existentes en FAJITA

Como se mencionó anteriormente, en primera instancia FAJITA instrumenta el código para luego poder restringir las soluciones del SAT-Solver a aquellas que son interesantes para el criterio de cobertura deseado. La instrumentación agregar unas variables especiales llamadas `roops_goal_x` (donde x es el número de goal), que son las que el SAT-Solver intenta cubrir.

Este capítulo muestra los criterios de cobertura existentes en FAJITA al inicio del proyecto, con sus ventajas y desventajas, para luego dar lugar a los implementados en este proyecto.

2.1. Cobertura de objetivos

La cobertura de objetivos mide la cantidad de objetivos que son alcanzados luego de la ejecución de una batería de tests. Cada objetivo se especifica con una anotación especial en el código que no afecta la ejecución del mismo. Si un test del conjunto genera una traza de ejecución que pasa por dicho punto, entonces se considera que fue alcanzado.

Cada uno de estos objetivos es agregado por el programador a mano en el lugar donde quiera que sea ejercitado por los tests. En consecuencia, los tests que se generen van a responder a los objetivos propuestos por el programador y no a un

criterio en particular.

Listing 2.1: Ejemplo de cobertura de objetivos

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

En el ejemplo anterior, se busca generar un test que ejercite el `catch`. Esto podría darse en el caso que el programador se quiera asegurar un buen manejo de excepciones en el código y para eso quiere tener un test.

Como se mencionó en la sección 1.2.1, al instrumentar el código se agregan unas variables llamadas `roops_goal_x`. Para este caso, la instrumentación de dicho código quedaría de la siguiente manera:

Listing 2.2: Ejemplo de cobertura de objetivos

```
try {
    FileReader f = FileReader("prueba.txt");
} catch (Exception e) {
    roops_goal_0 = true;
}
```

Estas variables son inicializadas en false por default.

2.2. Cobertura de decisiones

Esta métrica reporta la cantidad de expresiones booleanas que evalúan tanto verdadero como falso. También puede pensarse como un criterio que cubre las distintas ramas que existen dentro de la traza de ejecución de un código.

Por ejemplo, dado el siguiente código, un posible conjunto de tests que cubra el 100 % de las ramas consistiría en un test que evalúe ambos condicionales en verdadero y otro que evalúe ambos condicionales en falso (el primer condicional define las primeras dos ramas y el segundo condicional define las otras dos).

Listing 2.3: Ejemplo cobertura de decisiones

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

Claro está que este conjunto de tests cubre el 100 % de las ramas. Sin embargo, es fácil ver que una ejecución que haga verdadera la primer condición y falsa la segunda encontraría un error en el código.

Si ahora observamos el siguiente ejemplo, vemos que esta métrica también podría omitir algunas ejecuciones que econtrarían una falla.

Listing 2.4: Ejemplo cobertura de decisiones

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

En este caso, se podría llegar a encontrar un test que no ejecute `s.isEmtpy()` y entonces nunca se encontraría la falla en este código. De todas maneras, obtendría un 100 % de cobertura.

Capítulo 3

Criterios de cobertura implementados

Los criterios de cobertura que se implementaron para extender los existentes fueron los siguientes:

- Cobertura de múltiples condiciones
- K-Cobertura de caminos
- Cobertura de pares definición-uso

Esta sección detalla los criterios anteriormente mencionados junto con sus implementaciones.

3.1. Cobertura de múltiples condiciones

Esta métrica se impulsa en la falencia que tiene el criterio de cobertura de decisiones para cubrir todos las subexpresiones de una condición.

Como se pudo evidenciar en el ejemplo 2.3, el criterio de cobertura de decisiones puede pasar por alto casos en los cuales la evaluación de la condición puede arrojar errores. Es por esto que es importante que exista una métrica que pueda cubrir este tipo de casos. En consecuencia, lo que busca el criterio de cobertura de múltiples condiciones es evaluar cada expresión atómica tanto por verdadero como por falso.

Es una variación del criterio de múltiples condiciones explicado en el libro Introducción al testeo de sofware [3]. El criterio explicado en el libro menciona que se debe generar un test por cada posible combinación de valor de verdad de todas las subexpresiones atómicas de la expresión. Es decir, se requieren 2^n tests, donde `n` es la cantidad de subexpresiones atómicas. Esto es practicable para condiciones de unas pocas subexpresiones, pero a medida que `n` crece se convierte en un problema exponencial.

Volviendo al mismo ejemplo que el utilizado en Cobertura de decisiones pero esta vez utilizando el criterio de Cobertura de múltiples condiciones, el objetivo sería el siguiente: obtener una batería de tests que, luego de ejecutados, hayan logrado que:

- `condicion1` haya tomado el valor verdadero por lo menos una vez.
- `condicion1` haya tomado el valor falso por lo menos una vez.
- `condicion2` haya tomado el valor verdadero por lo menos una vez.
- `condicion2` haya tomado el valor falso por lo menos una vez.
- `s.isEmpty()` haya tomado el valor verdadero por lo menos una vez.
- `s.isEmpty()` haya tomado el valor falso por lo menos una vez.

Listing 3.1: Ejemplo cobertura de decisiones

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

En este caso, el criterio de cobertura de múltiples condiciones mostraría que no existe un test que pueda generar una valuación verdadera o falsa de `s.isEmpty()` y por lo tanto reportaría una cobertura inferior al 100 %, ayudando eventualmente al programador a darse cuenta que existe un problema en el código.

3.1.1. Implementación

Para poder lograr este tipo de cobertura se instrumentó el código de la siguiente manera.

Se visitaron todas las expresiones booleanas existentes en el código. Cada una de ellas se las exploró hasta dar con las mínimas expresiones booleanas. Una vez obtenidas dichas expresiones, se genereraron 2 variables del tipo `roops_goal_x` por cada una; la primera es verdadera si la expresión es verdadera, y la otra es verdadera si la expresión es falsa. Volviendo al ejemplo anterior, este sería el código instrumentado

Listing 3.2: Ejemplo cobertura de decisiones instrumentado

```
String s = null;
roops_goal_0 = condicion1;
roops_goal_1 = condicion1 == false;
roops_goal_2 = condicion2;
roops_goal_3 = condicion2 == false;
roops_goal_4 = s.isEmpty();
roops_goal_5 = s.isEmpty() == false;
if (condicion1 && (condicion2 ||  s.isEmpty())) {
    s = 'Verdadero';
} else {
    s = 'Falso';
}
```

De esta manera, se reportará un 100 % de cobertura si se puede generar una batería de tests que logre evaluar `condicion1`, `condicion2` y `s.isEmpty()` tanto verdadero como falso.

3.2. K-Cobertura de caminos

Al igual que el criterio anterior, este también se impulsa en una falencia del criterio de cobertura de decisiones. Sin embargo, esta vez haremos foco en las ramas visitadas y el orden en el cual se visitan y no en la evaluación de las expresiones

booleanas.

Este criterio tiene por objetivo cubrir todos los caminos posibles dentro de un código. Como se puede intuir, el problema tiene ciertas limitaciones. La cantidad de caminos que existen dentro de un código pueden variar mucho dependiendo de la cantidad de veces que los ciclos sean ejecutados. Por ejemplo, la cantidad de caminos posibles en el siguiente código que busca un elemento dentro de un árbol binario de búsqueda depende de la cantidad de veces que pueda ser ejecutado el `while`:

```
public boolean contains(int elem) {
    Node current = root;
    while (current != null && current.getElem() != elem) {
        if (current.getElem() > elem) {
            current = current.getRight();
        } else {
            current = current.getLeft();
        }
    }
    if (current == null) {
        return false;
    }
    return true;
}
```

Veamos por un instante cuáles son los posibles caminos para este fragmento de código:

1. La raíz es `null` y por lo tanto no entra en ningún momento al `while`.
2. El árbol tiene un único elemento y no es igual al buscado. En este caso entra una única vez al `while`, entra al `else` y al salir del ciclo entra al `if`.
3. El árbol tiene un único elemento y es igual al buscado. En este caso no entra al `while` y tampoco entra al condicional a la salida del mismo.
4. ...

Y así se podrían enumerar una cantidad infinita de caminos para este fragmento de código. En consecuencia, es evidente que generar inputs para cubrir todos los caminos posibles es impracticable.

Por lo tanto, para este criterio de cobertura se eligió desarmar los ciclos, repitiendo el código del mismo una cierta cantidad de veces `k` (de allí que el nombre del criterio de cobertura sea k-cobertura de caminos y no cobertura de caminos). De esta manera, se limita la cantidad de veces que el ciclo puede ser visitado y la cantidad de caminos posibles queda acotada.

3.2.1. Implementación

Lo primero que se realizó para la implementación de este criterio fue desarrollar un código que permitiera desarmar los ciclos. Por cada ciclo, replicó el código interno de cada ciclo `K` veces. Cada una de estas copias del código se encuentra dentro de un condicional, cuya guarda es la condición que originalmente tenía el ciclo. De esta manera, si aplicamos este procedimiento con `K = 2` al código recién visto, este quedaría como se muestra a continuación:

```
public boolean contains(int elem) {
    Node current = root;
    if (current != null && current.getElem() != elem) {
        if (current.getElem() > elem) {
            current = current.getRight();
        } else {
            current = current.getLeft();
        }
    }
    if (current != null && current.getElem() != elem) {
        if (current.getElem() > elem) {
            current = current.getRight();
        } else {
            current = current.getLeft();
        }
```

```
    }
    if (current == null) {
        return false;
    }
    return true;
}
```

Una vez que se tiene este código, el siguiente paso es instrumentar de la misma manera que se hace en el Criterio de cobertura de decisiones. Es decir, se busca generar un `roops_goal_x` por cada rama existente. El codigo instrumentado quedaría de la siguiente manera:

```
public boolean contains(int elem) {
    Node current = root;
    if (current != null && current.getElem() != elem) {
        roops_goal_0 = true;
        if (current.getElem() > elem) {
            roops_goal_2 = true;
            current = current.getRight();
        } else {
            roops_goal_3 = true;
            current = current.getLeft();
        }
    } else {
        roops_goal_1 = true;
    }
    if (current != null && current.getElem() != elem) {
        roops_goal_4 = true;
        if (current.getElem() > elem) {
            roops_goal_6 = true;
            current = current.getRight();
        } else {
            roops_goal_7 = true;
```

```
                current = current.getLeft();
            }
        } else {
            roops_goal_5 = true;
        }
        if (current == null) {
            roops_goal_8 = true;
            return false;
        } else {
            roops_goal_9 = true;
        }
        return true;
}
```

Dado que el SAT-Solver trata de hacer verdadera todas las variables del tipo `roops_goal_x` individualmente, si el código se dejara así y se enviara a TACO, entonces tendríamos no más que una versión modificada de cobertura de condiciones. Para que funcione como cobertura de caminos, es necesario indicarle al SAT-Solver que una vez cubiertas una secuencia de variables, no las vuelva a cubrir todas a la vez. Es decir, si el SAT-Solver retorna una solución que hace verdaderas a las variables 0, 2, 5 y 9, entonces se agrega una claúsula indicando que dichas variables no pueden ser verdaderas al mismo tiempo.

Por ejemplo, si las variables que utilizar el SAT-Solver para representar a los objetivos 0, 2, 5 y 9 son `roops_goal_0`, `roops_goal_2`, `roops_goal_5` y `roops_goal_9` respectivamente entonces la cláusula agregada al SAT-Solver es de la siguiente forma:

$$\neg\texttt{roops\_goal\_0} \vee \neg\texttt{roops\_goal\_2} \vee \neg\texttt{roops\_goal\_5} \vee \neg\texttt{roops\_goal\_9} \quad (3.2.1)$$

Dicha fórmula es equivalente a:

$$\neg(\texttt{roops\_goal\_0} \wedge \texttt{roops\_goal\_2} \wedge \texttt{roops\_goal\_5} \wedge \texttt{roops\_goal\_9}) \quad (3.2.2)$$

Por lo tanto, en el momento que el SAT-Solver intente generar una nueva solución, esta restricción evitará que las variables recién menciadas sean verdaderas todas a la vez, forzando entonces a que el camino recorrido por el nuevo tests sea distinto al del los tests anteriores.

Un problema conocido con este criterio de cobertura es que no se sabe qué porcentaje del total de caminos fueron cubiertos, dado que contar la cantidad de caminos existentes en un código no es tarea sencilla.

3.3. Cobertura de pares definición-uso

Este critero ya no está relacionado con los existentes en Fajita y trata de cubrir un aspecto del código totalmente diferente. Este criterio se impulsa en la asunción que para testear un software de manera adecuada, es necesario hacer foco en los flujos de los datos en las variables. Específicamente, debería asegurarse que los valores creados en un punto del programa son creados y usados de manera correcta. Para hacer esto, es necesario focalizarse tanto en las definiciones de las variables como en los usos.

Una `definición` es una ubicación en el código en el cual un valor de una variable es guardado en memoria. Un `uso` es una ubicación en la cual la variable es accedida. Los criterios de testing basados en el flujo de los datos usan el hecho que los valores son llevados desde las definiciones hasta los usos. Esto se considera un par definición-uso. [2]

Por ejemplo, para el siguiente código existen 3 definiciones de la variable `s1` y 2 usos de la misma. En cuanto a pares definición-uso, existen 3. El que comprende la definición en la línea 1 y el uso en la línea 3, otro que comprende la definición en la línea 1 y el uso en la línea 5 y por último el que comprende la definición en la línea 3 y el uso en la 5.

```
1. String s1 = "string";
2. if (condicion1) {
3.     s1 = s1.split('s');
4. }
```

```
5. s1 = s1.trim();
```

3.3.1. Implementación

Como primer paso para poder implementar este criterio, es importante poder descubrir todas las definiciones de variables dentro del código y de alguna manera dejar registro de las mismas para luego, ante un uso, poder saber cuáles definiciones son válidas.

Luego, cuando se encuentra un uso, se agrega una variable del tipo `roops_goal_x` por cada definición de la variable en uso. De esta manera, un goal toma el valor verdadero si el uso de la variable corresponde a una definición específica de la misma.

Volviendo al ejemplo de código anterior, la instrumentación del mismo daría el siguiente resultado:

```
1.  String s1 = "string";
2.  variable_definition_0 = true;
3.  if (condicion1) {
4.      s1 = s1.split('s');
5.      roops_goal_0 = variable_definition_0;
6.      variable_definition_1 = true;
7.      variable_definition_0 = false;
8.  }
9.  s1 = s1.trim();
10. roops_goal_1 = variable_definition_0;
11. roops_goal_2 = variable_definition_1;
12. variable_definition_2 = true;
```

Como se puede observar, las definiciones de variables están registradas poniendo en `true` las variables llamadas `variable_definition_x`. Cuando se agrega una definición para una variable que ya tenía una definición previa, es crucial indicar que la definición anterior ya no es válida. Un ejemplo de esto puede verse en las líneas 6 y 7. `s1` ya tenía una definición en la línea 1 y en la línea 4 se la vuelve a definir.

Entonces, una batería de tests que reporte una cobertura del 100 % debería cubrir

los siguientes casos:

1. Hacer válida la `condicion1` y de esta manera ejercitar el par definición uso comprendido entre las líneas 2 y 4 y luego el par definición uso comprendido entre las líneas 4 y 9.

2. Hacer que `condicion1` tome el valor `false` y en consecuencia hacer que se ejercite el par definición uso comprendido entre las líneas 2 y 9.

Basta cubrir todos los goals para poder decir que existe un 100 % de cobertura.

Capítulo 4

Evaluación y Resultados

Este capítulo realiza una comparación de los nuevos criterios implementados en FAJITA con algunas de las herramientas mecionadas en el capítulo 1.

4.1. Casos de Estudio: Estructura de Datos

En esta sección se describen las diferentes Estructura de Datos que se utilizaron para poder testear las diferentes herramientas y generar resultados comparativos. Las mismas fueron una adaptación del benchmark de `ROOPS`.

1. `Singly Linked List`: Implementación de una lista simplemente encadenada.
2. `Double Linked List`: Implementación de una lista circular doblemente encadenada.
3. `Binary Search Tree`: Implementación de un árbol de búsqueda binara.
4. `AVL Tree`: Implementación de un árbol de búsqueda binaria del tipo AVL balanceado.

4.2. Herramientas Utilizadas

Para poder comparar la cobertura obtenida con FAJITA para los nuevos criterios de cobertura se utilizaron las siguientes herramientas:

1. PEX: Desarrollado por Microsoft, PEX es una herramienta para generación de inputs para tests. PEX 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).

2. Randoop: Genera JUnits utilizando feedback de la ejecución de los mismos a medida que los construye para evitar generar inputs inválidos o redundantes. Las instrucciones que ejecuta durante los tests las selecciona de manera aleatoria.

3. EvoSuite: Genera tests para código escrito en Java, optimizando de acuerdo al criterio de cobertura. Actualmente esta herramienta provee cobertura de ramas.

Los métodos repOK representan el invariante de representación del objeto retornando verdadero cuando una instancia cumple el invariante de la clase.

4.3. Configuración

4.4. Formato de los resultados

Los resultados se encuentran en el Apéndice A en forma de tablas. Cada tabla contiene la clase y los métodos que fueron testeados. Las restantes columnas pertenecen a las distintas herramientas y muestran el porcentaje de cobertura sobre el total. Entre paréntesis se encuentra el tiempo de ejecución. Aquellos tiempos que figuren como TO quieren decir que la ejecución de la herramienta terminó por timeout.

4.5. Evaluación

Para realizar la evaluación de todos los criterios de cobertura para todas las herramientas, se llevaron a cabo los siguientes procedimientos:

1. Se instrumentaron todas las clases y los métodos a testear con Fajita para poder tener el código con los objetivos.

2. Se corrió Fajita para poder obtener la cobertura de la herramienta frente a los diferentes criterios de cobertura.

3. Se generaron tests con las herramientas mencionadas anteriormente.

4. Se corrieron los tests uno por uno y se inspeccionaron las clases para poder determinar cuáles objetivos habían sido cubiertos y cuáles no. Con estos datos se pudo determinar el porcentaje de cobertura de las herramientas con respecto al criterio de cobertura deseado.

Los mismos se llevaron a cabo en un hardware y software con las siguientes características:

- Procesador Intel Core i5-460 2.53 GHz
- 4GB 1067 Mhz DDR3 de memoria RAM
- Windows 7 Professional para la herramienta PEX ya que la misma no corre sobre distribuciones Unix, y Ubuntu 12.04 para las demás.

Dado que las herramientas utilizadas no poseen soporte para los criterios de cobertura implementados, se tomó la cantidad de objetivos totales reportados por FAJTA como cantidad a cubrir. En el caso de K-Cobertura de caminos, como no se cuenta con la cantidad de caminos posibles, simplemente se comparó la cantidad de caminos cubiertos por las herramientas. Todos los ciclos para este criterio de cobertura fueron replicados 2 veces. Es decir, se corrió K-Cobertura de caminos con `K` igual a 2.

Para realizar la comparación, a las herramientas se les dio un tiempo máximo de ejecución igual al que FAJTA tardaba en finalizar su ejecución. Es decir, si Fajita requería de 40 segundos para terminar de ejecutar, las otras herramientas tenían un tiempo máximo de ejecución de 40 segundos. Para aquellos casos en los que Fajita no llegó a completar un 100 % de cobertura, se tomó un tiempo máximo de 5 minutos para la generación de tests. En los resultados se puede observar que en algunos casos el tiempo fue levemente mayor a 5 minutos, esto se debe por el overhead al instrumentar, por lo tanto para las herramientas se tomó ese mismo tiempo de timeout.

Para el caso de K-Cobertura de caminos, dado que no se sabe cuántos caminos hay por cubrir, a todas las herramientas se les dio un tiempo máximo de ejecución igual al que FAJITA tardó en cubrir la máxima cantidad de paths. Tanto Pex como EvoSuite no llegaron a utilizar dicho tiempo de ejecución. Randoop, dado que genera tests y su criterio de corte es el tiempo máximo, sí utilizó esta cota. Por último, Fajita también utilizó esta cota de tiempo pero se reporta el tiempo en el cual llegó a su máxima cobertura.

Para la experimentacón con FAJTA, se seteó la cantidad de unrolls en 1 y se habilitó la opción de unroll incremental, para así poder extender el dominio de búsqueda. Esto quiere decir que en cada nueva corrida, aumenta en 1 la cantidad de unrolls llevados a cabo.

4.6. Resultados

4.6.1. Resultados para cobertura de pares definición-uso

En la tabla 1 se muestran los resultados para el criterio de cobertura de pares definición-uso. Como se puede observar en el promedio de objetivos cubiertos, Fajita fue superior al resto de las herramientas. En gran parte esto se debe a que Fajta tiene implementado el criterio de cobertura sobre el cual se trabajó y el resto de las herramientas no.

Una particularidad que merece ser destacada es la siguiente. Randoop no generó una lista válida en el caso de Double Linked List y es por eso que para los métodos `contains` y `addLast` la cobertura fue 0 %.

Dado que este criterio es ciertamente particular, es natural que las herramientas que no lo tienen incorporado hayan reportado porcentajes de cobertura relativamente bajos, mientras que Fajita obtuvo porcentajes, en promedio, cercanos al 87 %.

4.6.2. Resultados para cobertura de múltiples condiciones

En la Tabla 2 pueden observarse los resultados para este criterio de cobertura.

Los resultados para este criterio de cobertura no tienen grandes diferencias con

aquellos reportados en la sección anterior. Fajita tuvo un rendimiento ligeramente superior mientras que las otras herramientas bajaron su rendimiento. Esto puede explicarse nuevamente con el hecho que Pex, Randoop y EvoSuite no tienen implementado el criterio de cobertura puesto a prueba y Fajita sí.

Otra razón por la cual las herramientas comparadas no tienen un buen desempeño en este criterio en particular es porque, como la mayoría se basan en cobertura de ramas, simplemente buscan evaluar los condicionales tanto por verdadero y por falso. No tienen en cuenta las subexpresiones dentro de los condicionales y por lo tanto no exploran otros inputs que hacen que el condicional sea verdadero.

4.6.3. Resultados para K-Cobertura de caminos

Los resultados para este criterio de cobertura fueron realmente llamativos. Es notable como Fajita cubre una cantidad considerablemente mayor de caminos que las otras herramientas. Por ejemplo, si se observa la Tabla3, en el método `remove` del Binary Search Tree Fajita encuentra 31 caminos mientras que las otras herramientas logran encontrar como mucho 5.

En el total, Fajita cubrió más del doble de caminos que Randoop y EvoSuite y casi 5 veces más que Pex.

4.6.4. Resultados generales

En general se puede observar que Fajita tuvo un mejor rendimiento que Pex, Randoop y EvoSuite. Era un resultado esperable ya que ninguna de las últimas tiene implementados los criterios de cobertura que fueron analizados.

De todas maneras, algo destacable de Fajita es que la cobertura que obtiene la hace con un conjunto minimal de tests. Si por el contrario se toma Randoop como contraejemplo, esta herramienta genera en el orden de los cientos tests en unos pocos segundos. Esto hace que el análisis de los mismos para evaluar la cobertura obtenida no sea tarea sencilla.

Adicionalmente, Pex, EvoSuite y Randoop no se focalizan en un método en particular a la hora de generar tests, por lo que se tiene que tener extremo cuidado

en que los métodos a los que llama no den por cubiertos objetivos que no deberían ser cubiertos por ese método.

Capítulo 5

Conclusiones

5.1. Conclusiones

En este trabajo se mostró la flexibilidad que presenta Fajita a la hora de implementar criterios de cobertura nuevos basados en objetivos. Esta flexibilidad hace que Fajita esté por delante de otras herramientas de generación automática de inputs para tests. En general, las herramientas existentes tienen cobertura de ramas y por lo tanto no logran buenos desempeños cuando se trata de otro tipo de coberturas. Fajita, al tratarse de una herramienta que trabaja con objetivos logra buenos desempeños cuando el criterio de cobertura se basa en alguno de estos.

Sin embargo, la tarea de instrumentar código puede no ser trivial. Hay que tener en cuenta que existen diferentes maneras de escribir código equivalente y no todas son igualmente sencillas. Asimismo, los criterios de cobertura pueden variar en complejidad. Si los objetivos dependen de variables dentro del código o dependen entre si, entonces la instrumentación puede tener una complejidad elevada.

Por otra parte, el obtener un conjunto minimal de tests es una caracerística que diferencia a Fajita de las otras herramientas. Con una menor cantidad de tests se pueden cubrir la misma cantidad o más goals que con las otras herramientas. Esto facilita el análisis de los objetivos cubiertos.

Bibliografía

[1] Pablo Abad, Nazareno Aguirre, Valeria Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan Galeotti, Tom Maibaum, Mariano Moscato, Nicolas Rosner, Ignacio Vissani (2012), *Tight Bounds + Incremental SAT = Better Test Generation under Rich Contracts*. Buenos Aires, Argentina.

[2] Paul Ammann and Jeff Offutt (February 2008). *Definición-Uso*, Introduction to Software Testing. pp44, Cambridge, UK.

[3] Paul Ammann and Jeff Offutt (February 2008). *Cobertura de múltiples condiciones*, Introduction to Software Testing. pp 107, Cambridge, UK.

[4] Daniel Alfredo Ciolek (March 2012), *Fajita: Generación automática de casos de test basada en verificación acotada*. Buenos Aires, Argentina.

[5] Nikolai Tillmann and Jonathan de Halleux. *Pex-white box test generation for .net. TAP'08, pp. 134–153, 2008*

[6] *Autotest.* `http://autotest.readthedocs.org/en/latest/`

[7] *Randoop.* `http://randoop.googlecode.com/hg/doc/index.html`

[8] *TestEra.* `http://cs.txstate.edu/g_y10/testera/`

[9] *Alloy.* `http://alloy.mit.edu/alloy/`

[10] *Korat.* `http://korat.sourceforge.net/`

[11] Carlos Pacheco and Michael D. Ernst. *Randoop: Feedback-Directed Random Testing for Java.* `http://people.csail.mit.edu/cpacheco/publications/randoopjava.pdf`

[12] Gordon Fraser y Andrea Arcuri *EvoSuite: automatic test suite generation for object-oriented software. New York, NY, USA, 2011*

[13] Gordon Fraser y Andrea Arcuri *JML.* `http://www.eecs.ucf.edu/ leavens/JML/index.s`

Apéndice A

Resultados

A.1. Resultados

Estructura	# Goals	Pex	EvoSuite	Randoop	Fajita
Singly Linked List					
contains	14	92 % (12s)	92 % (11s)	100 % (TO)	100 % (71s)
insertBack	12	66 % (11s)	75 % (11s)	83 % (TO)	83 % (5m30s)
remove	19	0 % (15s)	89 % (11s)	94 % (TO)	89 % (5m7s)
Double Linked List					
contains	21	31 % (11s)	95 % (81s)	0 % (TO)	95 % (5m33s)
addLast	16	100 % (8s)	100 % (TO)	0 % (TO)	100 % (30s)
removeIndex	26	48 % (13s)	60 % (TO)	12 % (TO)	61 % (5m41s)
Binary Search Tree					
contains	10	70 % (14s)	70 % (TO)	100 % (TO)	100 % (30s)
add	27	33 % (12s)	51 % (TO)	70 % (TO)	70 % (83s)
remove	80	3 % (11s)	35 % (84s)	20 % (TO)	63 % (95s)
AVL Tree					
searchNode	13	61 % (10s)	76 % (TO)	53 % (TO)	100 % (27s)
searchMax	10	80 % (10s)	80 % (81s)	60 % (TO)	90 % (5m19s)
searchMin	10	80 % (12s)	90 % (81s)	60 % (TO)	90 % (5m46s)
Promedio Cubierto	-	55.33 %	76.08 %	54.33 %	86.75 %

A.1: Tabla 1: Resultados para Cobertura de pares definición-uso

Estructura	# Goals	Pex	EvoSuite	Randoop	Fajita
Singly Linked List					
contains	10	80 % (12s)	70 % (11s)	80 % (TO)	90 % (5m16s)
insertBack	6	66 % (TO)	41 % (TO)	83 % (TO)	100 % (10s)
remove	14	14 % (15s)	78 % (11s)	85 % (TO)	92 % (5m44s)
Double Linked List					
contains	12	25 % (11s)	91 % (TO)	0 % (TO)	100 % (26s)
addLast	2	50 % (8s)	50 % (TO)	0 % (TO)	100 % (13s)
removeIndex	16	43 % (13s)	68 % (TO)	43 % (TO)	81 % (5m10s)
Binary Search Tree					
contains	8	50 % (14s)	50 % (84s)	87 % (TO)	87 % (5m12s)
add	12	33 % (12s)	66 % (TO)	75 % (TO)	91 % (5m35s)
remove	26	7 % (11s)	61 % (84s)	30 % (TO)	80 % (5m33s)
AVL Tree					
searchNode	8	62 % (10s)	87 % (81s)	25 % (TO)	87 % (5m17s)
searchMax	6	66 % (10s)	83 % (81s)	33 % (TO)	83 % (6m14s)
searchMin	6	66 % (12s)	83 % (81s)	33 % (TO)	83 % (6m10s)
Promedio Cubierto	-	44 %	69 %	45 %	89 %

A.2: Tabla 2: Resultados para Cobertura de múltiples condiciones

Estructura	Pex	EvoSuite	Randoop	Fajita
Singly Linked List				
contains	3 (12s)	3 (11s)	5 (TO)	6 (17s)
insertBack	2 (11s)	4 (11s)	5 (TO)	4 (14s)
remove	1 (15s)	5 (11s)	7 (TO)	7 (16s)
Double Linked List				
contains	1 (11s)	4 (81s)	0 (TO)	7 (17s)
addLast	1 (8s)	1 (81s)	0 (TO)	1 (13s)
removeIndex	1 (13s)	4 (81s)	4 (TO)	6 (107s)
Binary Search Tree				
contains	2 (14s)	2 (84s)	8 (TO)	8 (18s)
add	2 (12s)	3 (84s)	8 (TO)	12 (25s)
remove	1 (11s)	5 (84s)	2 (TO)	31 (26s)
AVL Tree				
searchNode	2 (10s)	4 (81s)	2 (TO)	6 (24s)
searchMax	2 (10s)	3 (81s)	0 (TO)	3 (25s)
searchMin	2 (12s)	3 (81s)	2 (TO)	3 (26s)
Total Path Cubiertos	20	41	43	94

A.3: Tabla 3: Resultados para K-Cobertura de caminos.