Generación Automática de Casos de Prueba Investigación en Variables Enteras

Diego Vallespir Ligugnana
Universidad de la República, Facultad de Ingeniería, Instituto de Computación
Grupo de Ingeniería de Software (Gris)
Montevideo, Uruguay, 11300
dvallesp@fing.edu.uy

Noviembre 2005

Abstract

This paper presents the JFing project, whose goal is to automate the generation of test cases at class level and at class interaction level. For the test cases generation we take as basic strategies the ones presented by McGregor and Sykes for methods and Binder, and their N+ strategy for classes behavior. These strategies are automated and improved through the utilization of equivalence class partition. The state of the project, the obtained results and the future work are presented.

Keywords: Testing, Unit Testing, Automated Testing.

Resumen

Este artículo presenta el proyecto JFing que busca automatizar la generación de casos de prueba a nivel de clases y a nivel de interacción entre clases. Para la generación de los casos de prueba se toman como estrategias básicas las presentadas por McGregor y Sykes para métodos y Binder y su estrategia N+ para comportamiento de clases, las cuales se automatizan y mejoran usando partición en clases de equivalencia. Se presenta el avance del proyecto, los resultados obtenidos y el trabajo a realizar a futuro.

Palabras claves: Testing, Testing Unitario, Testing Automatizado.

1. Introducción

Este artículo presenta los avances del proyecto de investigación *"JFing - Generación Automática de Casos de Prueba"*. Este proyecto es llevado adelante por el Grupo de Ingeniería de Software del Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República [7].

En esta sección se presenta el contexto de trabajo y los objetivos de la primera fase del proyecto. La segunda sección presenta la metodología usada para el proyecto. La tercera sección muestra el avance conseguido en la primera fase del proyecto y la última sección aborda las conclusiones.

1.1. Contexto y Objetivos Generales del Proyecto

Es claro, a esta altura, que es necesario realizar actividades de testing durante todo el proceso de desarrollo de software. Esperar a finalizar el desarrollo del software para luego realizar el testing se constituye en una práctica de alto riesgo [3], [4], [9], [10] y [12]. Esto se debe a que las fallas encontradas al finalizar el desarrollo pueden ser muchas y difíciles de remover, ya que se está testeando todo el sistema y es difícil saber dónde se encuentra el defecto. Por esto se realiza testing a todo nivel, unitario, de integración y de sistema entre otros.

En la literatura [3], [4], [10], [11] y [12] se considera como apropiado que el testing unitario sea realizado por el propio desarrollador de la unidad; esto es debido, entre otras cosas, al conocimiento que el desarrollador tiene de la unidad a testear; sin embargo, existen visiones diferentes. El conocimiento de la unidad a testear permite al desarrollador crear casos de prueba que logren un buen cubrimiento de código y que además detecten fallas, ya que probablemente él mismo sepa donde las puede encontrar. Por otro lado, Myers [9] plantea los problemas de la psicología de las pruebas. Este problema puede llevar al propio desarrollador a "hacerse trampas al solitario", esto quiere decir, que como él realizó el desarrollo de la unidad está ansioso de que ésta sea correcta y quizás genere casos de prueba triviales en los cuales la unidad no falla. Por otro lado, si el testing de la unidad lo realiza otra persona ésta debe interiorizarse del código para poder realizar buenos casos de prueba; se define un buen caso de prueba como aquél que tiene una alta probabilidad de provocar una falla al ser ejecutado. La interiorización en el código de esa unidad tiene su costo, el cual muchas veces no es menor. Como un complemento al planteo de Myers, entendemos que a los desarrolladores normalmente no les gusta realizar pruebas y esto también es debido a la psicología de las personas. Creemos que el testing es básicamente una actividad destructiva mientras que el desarrollo es una actividad constructiva. Esto da dos opciones contrapuestas con sus pros y sus contras. Una primera opción es que el propio desarrollador realice las pruebas, lo cual puede provocar que no se detecten muchas fallas pero que el costo no sea alto. La segunda opción es que sea otra persona quien realice las pruebas, lo que puede llevar a encontrar más fallas que en el caso anterior pero hay un costo más elevado que pagar.

Nuestro enfoque propone la generación automática de casos de prueba para evitar que ni el desarrollador ni una persona externa sean los que los generen. De esta manera evitamos caer en la psicología de pruebas mencionada por Myers y en los altos costos que trae aparejado involucrar una segunda persona. Por otro lado, se evita también el alto tiempo y esfuerzo que insume generar los casos de prueba de forma manual, acortando de esta manera el tiempo

de salida al mercado del producto y asegurando una mejor calidad.

Los casos de prueba se generaran automáticamente a partir de especificaciones realizadas en el lenguaje unificado UML [5]. A nivel de métodos se usan pre y post condiciones de cada método de una clase como especificación, a nivel de comportamiento de la clase se usa el diagrama de estados de la clase y como testing a nivel de integración de clases se usan los diagramas de secuencia o de colaboración entre clases.

Si bien este enfoque se puede usar con varios paradigmas, nuestro proyecto se basa en el testing de sistemas desarrollados con tecnología orientada a objetos y, en particular en estas primeras fases, con el lenguaje Java.

El proyecto realiza dos actividades en paralelo, a saber, el estudio teórico de la automatización del testing unitario y la construcción de una herramienta para la generación de casos de prueba de forma automática para clases desarrolladas con el lenguaje Java.

El aporte de nuestro trabajo consiste en tres grandes puntos:

1. Automatizar el testing con un enfoque riguroso.

2. Determinar el cubrimiento de código alcanzado al ejecutar los test generados automáticamente.

3. Investigar qué clases de defectos se detectan y qué clases de defectos no son detectados por la herramienta.

1.2. Trabajos Relacionados

En este momento no conocemos trabajos relacionados que abarquen automáticamente todo lo que quiere abarcar nuestro proyecto. De todas formas existen trabajos relacionados que han realizado aportes en el área.

En [6], se usan dos técnicas, Tabu Search y Scatter Search para generar casos de prueba a partir del flujo de control del programa y cumplir con el criterio de cubrimiento de decisión. Este artículo presenta el software diseñado hasta el momento y se relaciona a nuestro trabajo únicamente en la parte de testing de métodos. En nuestro caso no se usa ninguna de estas técnicas para resolver el problema. En el artículo mencionado no se muestran resultados más allá del cubrimiento de código alcanzado, como por ejemplo, tiempo de procesamiento para derivar casos de prueba, orden de complejidad de los algoritmos utilizados o mutant score entre otros. De todas maneras, es el trabajo que más podría servir en el futuro de JFing.

En [2] se presenta la herramienta Cow-Suite. Esta herramienta deriva casos de prueba para diagramas de interacción entre clases; está en una versión prototipo y no hay muchos resultados publicados de cubrimiento o mutant score. Esta herramienta se relaciona con nuestro trabajo en lo que tiene que ver con el testing interclases, que como se verá en la siguiente subsección no es un punto que se ataque en la primer versión de JFing.

Las clases de defectos que se pueden detectar con la estrategia N+ se han descrito en una investigación conducida por Antoniol y otros [1], pero ha sido muy controlado en cuanto al tamaño de la aplicación bajo test. Además los test fueron generados manualmente y no automáticamente. En nuestro trabajo usamos mutantes para descubrir que clases de defectos podemos encontrar con JFing y cuáles no.

1.3. Objetivos del Proyecto JFing 1.0

La primer versión de JFing, liberada en Mayo de este año, es una versión muy reducida de lo planteado como objetivos en la sección anterior.

JFing 1.0 no ataca el testing a nivel de interacción entre objetos, únicamente se encarga del testing a nivel de métodos y el testing a nivel de comportamiento de la clase.

Además solamente trabaja con enteros. Si bien es una herramienta para el lenguaje Java no acepta que una clase tenga como miembro de la misma a otros objetos, vectores por ejemplo. Mismo así, los métodos de la clase solamente pueden devolver enteros y aceptar enteros como parámetros.

La tercera limitación de JFing 1.0 es que se limita a derivar los casos de prueba a nivel de métodos a partir de las precondiciones y no deriva casos a partir de las poscondiciones.

Si bien las limitaciones mencionadas son grandes, de todas maneras, esta versión nos permite investigar la forma en la cual se van a automatizar los test y las dificultades con las cuales nos podemos topar a futuro.

En definitiva, JFing 1.0 tiene como objetivo automatizar el testing de clases que solo trabajan con el tipo int de Java y cuyos diagramas de estado solo trabajan con guardas del tipo int, derivando automáticamente los casos de prueba únicamente a partir de las precondiciones de los métodos.

2. Metodología

En esta sección se presenta la metodología usada para derivar casos de prueba para la versión 1.0 de la herramienta. Para tener una visión más general de la metodología a ser usada en subsiguientes versiones de JFing ver [13].

Se presentan dos subsecciones que muestran el enfoque usado para el testing tanto a nivel de métodos como a nivel de comportamiento. Queda fuera de este reporte la discusión de estrategias de testing a nivel de interacción entre clases ya que la versión 1.0 de JFing no ataca este tipo de testing. En ambos niveles de testing se usan y complementan dos estrategias conocidas de testing. Ambas estrategias son teóricas y se pueden aplicar en testing manual; nuestro trabajo consiste en automatizarlas y en mejorarlas usando partición en clases de equivalencia.

2.1. A Nivel de Métodos

Cada método tiene su especificación, esta está dada por la definición de las pre y post condiciones del mismo. Estas son descritas mediante el lenguaje Object Constraint Language (OCL) [14].

Para la versión actual de JFing solamente se derivan casos de prueba a partir de las precondiciones del método por lo cual solamente se va a presentar esta estrategia. Además se considera programación por contrato por lo cual los casos de prueba generados siempre cumplen con la precondición del método.

Las pre y post condiciones son restricciones sobre los miembros de la clase, los parámetros del método y el resultado del mismo. En la figura 1 se muestra un ejemplo de lo mencionado para un método que lo que hace es aumentar un acumulador de la clase. El acumulador es la suma del acumulador actual más el parámetro pasado. Además el método aumenta en

uno la cantidad de veces que fue ejecutado el mismo. La cantidad de veces de ejecución del método no puede ser mayor que 1.000 y el parámetro pasado debe ser positivo y menor que 100.

```
/**
*@pre  n >= 0 and n < 100 and elementos < 1000
*@post suma = suma@pre + n and elementos = elementos@pre + 1
*/
void push(int n)
```

Figura 1: Ejemplo de Pre y Post Condiciones de un Método

La derivación de los casos de prueba para realizar este testing se basa en la propuesta de McGregor y Sykes [8]. Esta propuesta es un conjunto de reglas de derivación de casos de prueba a partir de las pre y pos condiciones de los métodos, teniendo en cuenta los conectores (and, or, entre otros), que estén presentes en las mismas. Se muestran las reglas para derivar casos de prueba a partir de las precondiciones en la figura 2. La primer columna de la figura refiere a las precondiciones, y la segunda a los casos de prueba que esa precondición genera siguiendo el método.

Nuestra propuesta de derivación de casos de prueba, a nivel de métodos, usa estas reglas agregando la aplicación de las técnicas de partición en clases de equivalencia y valor límite a las variables referenciadas en las pre y pos condiciones, así como también a las no referenciadas pero que son pasadas como parámetros al método. A modo de fijar ideas, en la figura 3 se muestran los casos de prueba que se pueden generar usando esta combinación de técnicas para lo presentado en la figura 1. Luego veremos en la sección 3 que la herramienta genera una mayor cantidad de casos de prueba para este tipo de especificación.

Para aclarar mejor nuestro método presentamos este segundo ejemplo, supongamos que tenemos la siguiente precondición y poscondición:

Pre: $a \geq 0$ or $b > 0$

Post: $c = a + b$

Aplicando la tabla de la figura 2 obtenemos los siguientes meta-casos de prueba:

Entrada (pre)	**Salida (post)**
$a \geq 0$	$c = a + b$
$b > 0$	$c = a + b$
$a \geq 0$	$c = a + b$

Aplicando partición en clases de equivalencia y valor límite obtenemos los siguientes casos de prueba para la opcin que considera solo $a \geq 0$ (primera opción); y con JFing de forma automática:

Datos de Prueba	**Resultado Esperado**
$a = 0$ and $b = B$ cualquier entero	$c = B$
$a = 1$ and $b = B$ cualquier entero	$c = B + 1$
$a = MaxInt$, $b = B$ cualquier entero menor o igual a cero	$c = MaxInt + B$

Para el caso ($b > 0$) y el caso ($a \geq 0$ and $b > 0$) se aplica un procedimiento igual.

Expresión Lógica	Derivación de Casos de Prueba
true	(true, post)
1	(**1** , post)
not **1**	(not **1** , post)
1 and **2**	(**1** and **2** , post)
1 or **2**	(**1** , post) (**2** , post) (**1** and **2** , post)
1 xor **2**	(**1** and not **2** , post) (not **1** and **2** , post)
1 implies **2**	(not **1** , post) (**2** , post) (not **1** and **2** , post)
if **1** then **2** else **3** endif	(**1** and **2** , post) (not **1** and **3** , post)

Nota:
1, **2** y **3** representan componentes en una expresión OCL

Figura 2: Propuesta de McGregor y Sykes para Derivar Casos de Prueba desde las Precondiciones

Datos del Caso de Prueba	Resultado Esperado
(n=0, elementos=999)	(suma=suma@pre, elementos=elementos@pre+1)
(n=99, elementos=999)	(suma=suma@pre+99, elementos=elementos@pre+1)

Figura 3: Ejemplo de Aplicación de la Metodología

2.2. A Nivel de Comportamiento

El comportamiento de la clase se especifica mediante el diagrama de estados de la clase usando el lenguaje UML.

Para la versión JFing 1.0 no se considera que las transiciones entre estados puedan tener acciones, las mismas solamente pueden tener eventos y guardas para esos eventos. Como ya se mencionó las guardas pueden contener solamente enteros.

Otra restricción para los diagramas de clase es que para dos estados diferentes no pueden existir dos transiciones diferentes de un estado al otro.

La derivación de los casos de prueba a partir del diagrama de estados de clase se basa en la propuesta N+ de Binder [4]. Además se agrega partición en clases de equivalencia en los casos en que esto es posible. Esta estrategia considera al diagrama de estados como un grafo, donde los estados son los nodos y las transiciones son las aristas. Este grafo se recorre en amplitud o en profundidad y la recorrida se detiene al llegar a un nodo ya visitado; este último igual se agrega al árbol que se forma al realizar la recorrida. Cada camino que va desde la raíz hasta una hoja del árbol se constituye en un caso de prueba. Para cada caso de prueba se van realizando distintos chequeos al momento de ejecución, tales como chequeo de la precondición del método que ejecuta la transición y que el objeto se encuentre en el estado que se debería encontrar. En la sección 3 se muestra un ejemplo sencillo donde se usa esta estrategia para derivar casos de prueba.

3. La Herramienta - JFing 1.0

En esta sección se presenta la herramienta JFing 1.0. No se presentan detalles de implementación de la herramienta, sino que se presenta el funcionamiento de la misma a través de un ejemplo y se muestran los resultados obtenidos.

3.1. Ejemplo

Se presenta un ejemplo sencillo que cumple con la especificación de JFing 1.0 y con el cual se muestra cómo genera JFing los casos de prueba.

La clase del ejemplo se llama PilaEntera y es una implementación particular de una pila. La misma tiene dos métodos, un método push, para agregar elementos a la pila, y un método pop para sacar un elemento de la pila.

El método push sólo puede ingresar un entero mayor o igual que cero y menor que cien. Este método hace que la pila aumente en uno la cantidad de elementos que contiene y que aumente el miembro de la clase *suma*, que se inicializa en cero, en una cantidad igual al entero pasado.

El método pop devuelve el entero cien en caso que el acumulador sea mayor o igual que cien y en caso contrario devuelve el valor del acumulador. El método, además, resta al acumulador el entero devuelto. En la figura 4 se muestran las pre y post condiciones de los dos métodos que componen la clase.

El comportamiento de la clase depende de la máxima cantidad de elementos que la pila acepta, que para este ejemplo es de 1.000. La pila se inicializa en un estado llamado *Vacio*, en este estado la pila no tiene elementos, por lo que el miembro elementos es igual a cero. Además en este estado el acumulador se inicializa también en cero, con lo que el miembro

```
/**
*@pre  n >= 0 and n < 100 and elementos < 1000
*@post suma = suma@pre + n and elementos = elementos@pre + 1
*/
void push(int n)

/**
*@pre  elementos >= 1
*@post if suma@pre >= 100 then (result = 100 and suma = suma@pre - 100)
          else (result = suma@pre and suma = 0) endif
          and elementos = elementos@pre - 1
*/
int pop()
```

Figura 4: Pre y Post Condiciones de los Métodos push y pop

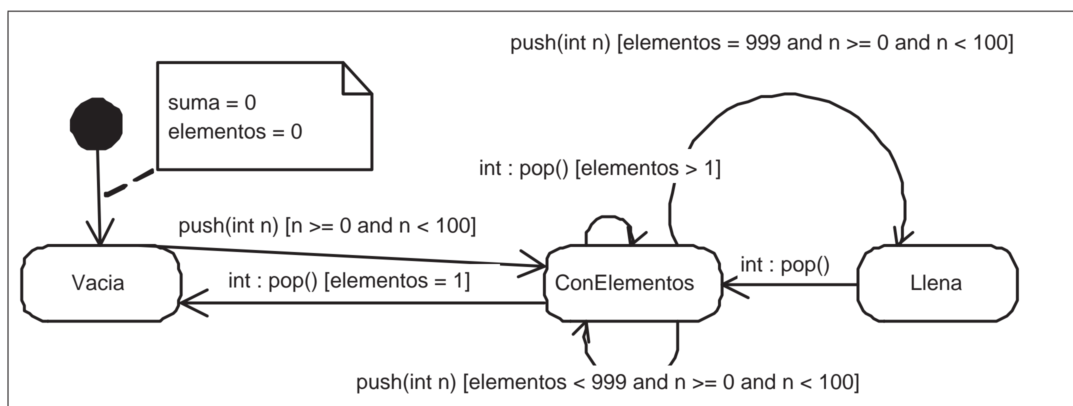

Figura 5: Diagrama de Estados de la Clase PilaEntera

suma está en cero. Cuando la pila tiene entre 1 y 999 elementos se encuentra en el estado *ConElementos*. Cuando la pila tiene 1.000 elementos est en el estado *Llena*. El diagrama de estados de la clase PilaEntera se encuentra en la figura 5.

Un invariante de la clase es: elementos ≥ 0 y elementos < 1000 y suma ≥ 0. La clase, escrita en su conjunto, escrita en su conjunto, se encuentra en la figura 6. Como se ve en la figura se encuentran otros métodos, los métodos get y set de cada miembro de la clase y los métodos que definen en qué estado se encuentra la clase llamados esEstado*NombreDelEstado*.

3.2. A Nivel de Métodos

En esta sección mostramos los casos de prueba generados por JFing a nivel de métodos.

La clase PilaEnteros tiene dos métodos a testear, a saber, push y pop [1]. Ambos métodos tienen sus precondiciones únicamente con conectores OCL *and*. Siguiendo la tabla presentada en la figura 2, tendríamos un número muy acotado de casos de prueba, pero al usar partición en clases de equivalencia y valores límite JFing genera un número importante de casos.

Al momento de testear un método JFing genera lo siguiente:

[1] JFing también realiza test de los métodos get y set pero no es de interés entrar en detalle en esto.

```
public class PilaEnteros {

    private int suma = 0;
    private int elementos = 0;

    public PilaEnteros() {}

    /**
    *@inv: elementos >= 0 and elementos <= 1000 and suma >= 0
    */

    public int getSuma(){return suma;}
    public int getElementos(){return elementos;}
    public void setSuma(int s){suma = s;}
    public void setElementos(int e){ elementos = e;}

    /**
    *@pre   n >= 0 and n < 100 and elementos <=1000
    *@post suma = suma@pre + n and elementos = elementos@pre + 1
    */
    void push(int n) {
        suma = suma + n;
        elementos = elementos + 1;
    }

    /**
    *@pre: elementos >= 1
    *@post: if suma@pre >= 100 then (result = 100 and suma = suma@pre - 100)
    */
     else (result = suma@pre and suma = 0) endif and elementos = elementos@pre - 1
    int pop() {
        int result;
        if (suma >= 100) {
            suma = suma - 100;
            result = 100;
        }
        else {
            result = suma;
            suma = 0;
        }
        elementos = elementos - 1;
        return result;
    }

     /**
     * @srm elementos = 0;
     */
    public boolean esEstadoVacia(){return elementos==0;}

    /**
     * @srm elementos > 0 and elementos < 1000;
     */
    public boolean esEstadoConElementos(){return (elementos>0) && ementos<1000);}

    /**
     * @srm elementos = 1000;
     */
    public boolean esEstadoLlena(){return elementos==1000;}

}
```

Figura 6: Clase PilaEntera en Lenguaje Java

	Datos de Entrada			Resultado Esperado	
Nro. Caso	**suma**	**elementos**	**n**	**suma**	**elementos**
1	0	0	0	0	1
2	0	0	1	1	1
3	0	0	99	99	1
4	1	0	0	1	1
5	1	0	1	2	1
6	1	0	99	100	1
7	100000	0	0	100000	1
8	100000	0	1	100001	1
9	100000	0	99	100099	1
10	0	1	0	0	2
11	0	1	1	1	2
12	0	1	99	99	2
13	1	1	0	1	2
14	1	1	1	2	2
15	1	1	99	100	2
16	100000	1	0	100000	2
17	100000	1	1	100001	2
18	100000	1	99	100099	2
19	0	999	0	0	1000
20	0	999	1	1	1000
21	0	999	99	99	1000
22	1	999	0	1	1000
23	1	999	1	2	1000
24	1	999	99	100	1000
25	100000	999	0	100000	1000
26	100000	999	1	100001	1000
27	100000	999	99	100099	1000

Tabla 1: Casos de Prueba para el Método push

1. Datos de prueba para testear el método
2. Llamado al método a testear
3. Chequeo del invariante de la clase
4. Chequeo de las poscondiciones del método

La clase generada se puede ejecutar con una herramienta de ejecución automática de casos de prueba open source.

En el cuadro 1 se presenta, en un formato amigable, lo generado por JFing para el método push. El cuadro muestra el resultado esperado, JFing calcula el mismo a partir de la poscondición al momento de ejecutar la prueba. En cambio, los datos de prueba son

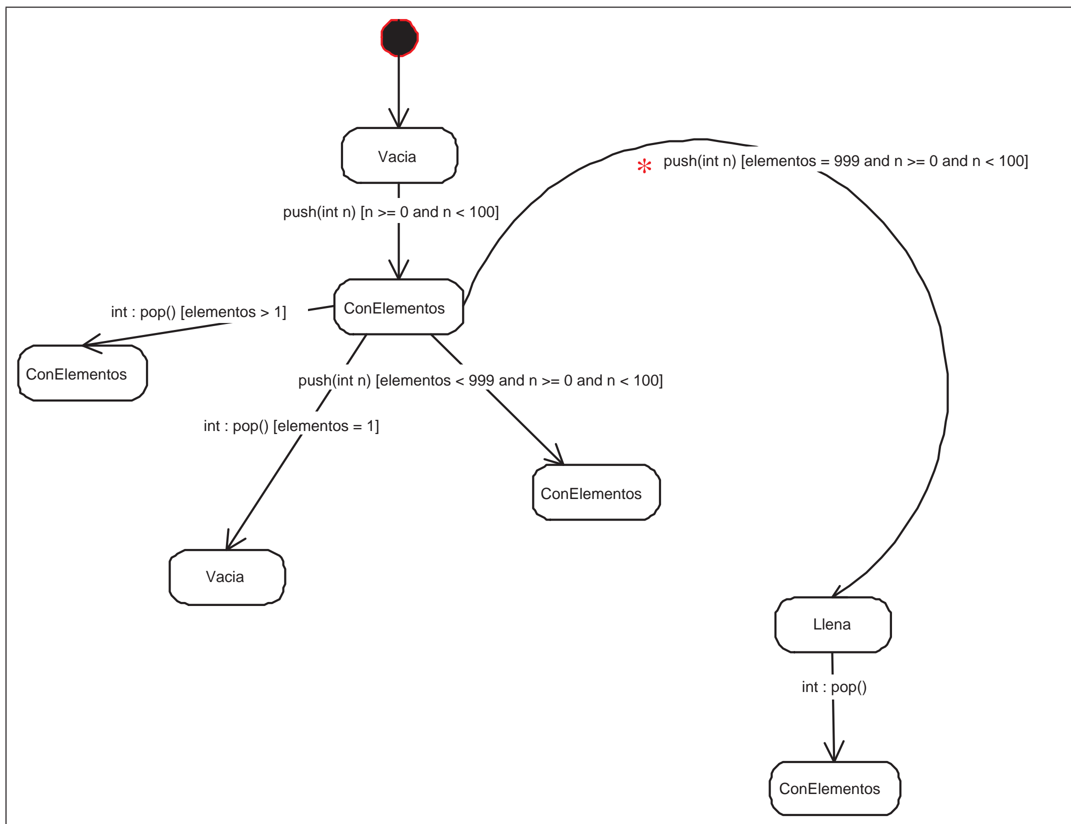

Figura 7: Árbol Derivado del Diagrama de Estados para Estrategia N+

generados al ejecutar JFing. La tabla se muestra de esta manera para que al lector le sea más sencillo entender lo realizado por JFing.

La cantidad de casos presentados muestra la *explosión* de casos de prueba que logra JFing. Para un método sencillo como el push presentado se obtienen 27 casos de prueba. Esto lo tomamos como una ventaja de JFing, más allá del tiempo de ejecución que puedan tener los casos de prueba, ya que la tasa de defectos que se puede detectar es alta.

Como se ve en la tabla todos los datos de prueba cumplen con la precondición del método push, lo cual se debe a que JFing genera casos de prueba considerando programación por contrato. Por tanto, no tiene sentido generar datos de prueba que no cumplen con la precondición. Además debe notarse la aplicación de la tabla presentada en la figura 2. En caso de tener un *or* en lugar de *and*, la cantidad de casos de prueba sería mucho mayor.

Luego que se generan los test de métodos JFing genera los test de comportamiento de la clase. Esto se presenta en la siguiente subseccioón.

3.3. A Nivel de Comportamiento

A nivel de comportamiento, como ya se mencionó, se automatiza la estrategia N+, ampliada con clases de equivalencia. La figura 7 muestra uno de los árboles posibles que se derivan a partir del diagrama de estados de la clase. Este árbol surge de una recorrida en amplitud de dicho diagrama.

Cada camino desde la raíz hasta una de las hojas es un único caso de prueba de comportamiento. Para el ejemplo presentado tenemos, entonces, cuatro casos de prueba. Uno de

los casos de prueba es especial, y es el único camino que va desde la raíz y que pasa por el estado *Llena*. Esto se debe a que la transición push desde el estado *conElementos* debe ser ejecutada más de una vez para llegar al estado *Llena*. Esta transición aparece marcada con un asterisco en la figura. Si se mira con atención el diagrama se nota que uno de los caminos es imposible de realizar, este es el camino de más a la izquierda en la figura. No se puede realizar un único push y luego un pop y que el estado sea *conElementos*.

JFing genera automáticamente casos de prueba para los cuatro caminos posibles con una implementación particular para el caso particular del camino que pasa por el estado *Llena*. JFing va recorriendo el árbol y generando los casos de prueba, pero como sólo ejecuta una vez una transición de un estado a otro se toma una decisión de cómo ejecutar el caso de prueba particular. La estrategia que se usa es poner a la clase en un estado tal que al ejecutar la transición se llegue al estado deseado. En este caso se pone al objeto con 999 elementos y se ejecuta el método push y luego se sigue recorriendo el árbol de la misma forma que si fuera un caso *"normal"*. El caso de estar en el estado *conElementos* y quedar en el mismo mediante un pop se resuelve de la misma manera que el caso recién descrito.

Como JFing utiliza partición en clases de equivalencia se generan más de cuatro casos de prueba, en definitiva, se generan varios casos de prueba para cada camino.

En el ejemplo mostrado se usan dos herramientas open-source para poder evaluar la calidad de los casos de prueba generados. Una de ellas mide cuánto se ha cumplido con el criterio de cubrimiento de decisión, logrando un 100 % del criterio cumplido. La otra herramienta genera mutantes automáticamente y calcula el mutant score, que en este caso tambieén fue del 100 %.

Si bien el ejemplo es sencillo estos resultados son indicadores positivos de la investigación que se está llevando a cabo.

4. Conclusiones

Si bien JFing tiene restricciones mayúsculas en este momento, de todas maneras nos atrevemos a afirmar que se va por el camino adecuado. Esto se desprende del cubrimiento alcanzado por las distintas pruebas realizadas y del Mutant Score obtenido.

Existe una dificultad seria para derivar casos de prueba a partir de las poscondiciones. Esto puede llegar a requerir recorrer el código en forma inversa y no puede ser totalmente automatizado. Se van a estudiar distintas heurísticas para determinar si se pueden derivar casos de prueba a partir de las poscondiciones de forma relativamente sencilla. Otra solución a investigar es el uso de lo propuesto por Basanieri [2] y otros y ver si se alcanzan todas las poscondiciones, sin preocuparse de derivar los casos a partir de estas.

En lo que respecta puramente a los enteros se tienen restricciones a nivel de cálculos multidimensionales. Para derivar casos de prueba cuando se presentan multidimensiones hay que resolver ecuaciones diferenciales y esto JFing 1.0 no lo soporta.

A JFing lo entendemos como una pequeña contribución en la línea de mejorar el testing, desde tres puntos de vista diferentes: que el mismo sea generado y ejecutado de forma automática, que se pueda realizar en etapas muy tempranas del desarrollo de software y que se ataque de una manera formal, obligando esto último a tener mejores especificaciones de lo desarrollado o que está en desarrollo.

4.1. Trabajo a Futuro

Lo más importante para comentar es el orden en el cual se va a encarar el futuro desarrollo de la herramienta.

1. Agregar testing a nivel de interacción de clases. Los miembros de las clases siguen siendo enteros pero un método puede crear una clase y enviar un método a la misma.
2. Agregar objetos simples a los miembros de las clases, parámetros de los métodos y resultados devueltos por los métodos.
3. Agregar, en los mismos casos que el punto anterior, colecciones de objetos.
4. Derivar Casos de Prueba a partir de postcondiciones.
5. Derivar Casos de Prueba cuando existen multidimensiones en las condiciones.

Referencias

[1] Antoniol, G.; Briand, L. C.; Di Penta, M.; Labiche, Y. A Case Study Using The Round-Trip Strategy for State-Based Class Testing. *Carleton University TR SCE-01-08*, Abril 2002.

[2] Basanieri, F.; Bertolino, A.; Lombardi, G.; Nucera, G.; Marchetti, E.; Ribolini, A. Cow-Suite: A UML-Based Tool for Test-Suite Planning and Drivation *Ercim News, Number 58*, Julio de 2004.

[3] Beizer, B. Software Testing Techniques. *The Coriolis Group*, Marzo, 2000.

[4] Binder B. Testing Object-Oriented Systems; Models, Patterns and Tools. *Addisson-Wesley*, 2000.

[5] Booch, G.; Rumbaugh, J; Jacobson, I. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[6] Díaz, E.; Blanco, R.; Tuya, J. Automated Software Testing with Metaheuristic Techniques. *Ercim News, Number 58*, Julio de 2004.

[7] Grupo de Ingeniería de Software, Universidad de la República. *www.fing.edu.uy/inco/grupos/gris*. Visitada por última vez el 22 de Mayo de 2005.

[8] McGregor, J.; Sykes, D. A Practical Guide to Testing Object-Oriented Software. *Addison-Wesley*, 2001

[9] Myers, G. The art of Software Testing. *John Wiley and Sons*, 1979.

[10] Pfleeger, S. Software Engineering: Theory and Practice. *Prentice Hall*, Febrero 2001.

[11] Pressman, R. : Software Engineering: A Practicioners Approach. McGraw-Hill, Abril 2004.

[12] Sommerville, I. Software Engineering. *Addison-Wesley Pub Co*, Agosto 2000.

[13] Vallespir, D. Generación Automática de Casos de Prueba para Objetos. *http://www.fing.edu.uy/˜dvallesp/indexEspaniol.html*, visitada por última vez el 17 de Noviembre de 2005.

[14] Warmer, J.; Kleppe, A. : The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1999.