Criterios de Cobertura sobre `RepOK` para Reducir Test Suites Exhaustivas Acotadas: Estudio de Casos

Valeria Bengolea[1,2], Simón Gutiérrez Brida[1] y Nazareno Aguirre[1,2]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina.
E-mail: {vbengolea, sgutierrez, naguirre}@dc.exa.unrc.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Resumen La generación exhaustiva acotada de casos de test es una técnica de generación de entradas para programas que consiste en construir todas las posibles entradas válidas hasta cierta cota dada. Aunque el tamaño de la *test suite* generada está ligado a la cota provista, inclusive para cotas muy pequeñas las *test suites* obtenidas resultan considerablemente grandes, haciendo al uso de las mismas algo prohibitivo. En este trabajo revisaremos, adicionando nuevos casos de estudios, un enfoque para reducir el tamaño de las *test suites* generadas exhaustivamente presentado previamente. Este enfoque está basado en el uso de criterios de cobertura de código sobre el invariante de representación de la estructura sobre la cual la *test suite* es producida. La implementación de este invariante es utilizada para decidir cuándo dos entradas válidas pueden ser consideradas equivalentes, lo cual sucede si éstas ejercitan el código del invariante de representación de manera similar de acuerdo algún criterio de cobertura de código de caja blanca. Esta relación de equivalencia entre las entradas válidas es aprovechada para descartar casos de test que cubren clases de equivalencias ya cubiertas por algún otro caso de test presente en la *suite*. En este trabajo, se adicionan nuevos casos de estudios que muestran que, a medida que las cotas crecen, reducir las *test suites* exhaustivas aplicando la técnica presentada, produce resultados similares a las *test suites* exhaustivas, en cuanto a habilidad para matar mutantes.

1 Introducción

En la actualidad, *testing* es una de las principales técnicas usadas para detectar errores en el software. Esta tarea consiste en ejecutar un programa con diferentes entradas con la intención de encontrar errores en él [4]. Para incrementar la probabilidad de encontrar errores en el software las entradas utilizadas deben ser heterogéneas, en el sentido que deben "ejercitar" el software de muchas maneras diferentes [13]. Muchas herramientas/técnicas de apoyo al *testing* han sido desarrolladas, en particular en torno a la generación automática de casos de test. Esta actividad resulta sencilla cuando se trata de tipos de datos básicos (por

ejemplo, generación aleatoria), pero es más compleja cuando se trata de rutinas parametrizadas con tipos de datos estructuralmente complejos (AVLs, árboles de búsqueda, grafos, etc), debido a las restricciones que estas estructuras deben satisfacer para ser consideradas válidas o bien formadas. La exploración exhaustiva acotada de todas las posibles entradas en busca de aquellas bien formadas es una técnica muy usada en este contexto [3, 5, 7, 10]. Este enfoque consiste en generar todas las posibles estructuras que satisfacen ciertas restricciones dentro de alguna cota preestablecida. Esto es algo muy costoso e inclusive para cotas pequeñas, muchas veces, produce *test suites* muy grandes, es decir, no sólo el proceso de generación de la *test suite* es costoso sino que además el uso de ésta durante la fase de *testing* se vuelve impracticable. Esta técnica está respaldada por la *hipótesis de la cota pequeña* [6], la cual sostiene que una gran porción de errores puede ser detectada probando el programa con entradas pequeñas; sin embargo, muchas veces cotas relativamente grandes son necesarias para reproducir ciertos errores, por ejemplo, probar rutinas de inserción/eliminación sobre árboles balanceados podría requerir árboles suficientemente grandes para reproducir errores inherentes a las operaciones de rebalanceo del árbol.

En este artículo, presentamos nuevos casos de estudio que muestran cúan efectiva es la técnica expuesta en [2] para reducir el tamaño de las *test suites* generadas exhaustivamente. Esencialmente, esta técnica se basa en la definición de un nuevo criterio de cobertura de *Caja Negra* con respecto al código bajo prueba (se prescinde de éste) definido en términos de algún criterio de *Caja Blanca* sobre el código de `repOK`. Más precisamente, este enfoque se basa en el uso de criterios de cobertura de código sobre el invariante de representación de la estructura, es decir sobre las restricciones que la estructura debe satisfacer para ser considerada bien formada. Estas restricciones son usadas para definir una relación de equivalencia entre las entradas válidas. Es decir, dada una implementación del invariante de representación, la cual comúnmente es llamada `repOK` [8], se analiza cómo el código de la misma es ejercitado por diferentes entradas y en base a eso se define una partición sobre todas las entradas válidas. Dos casos de test están en la misma clase de equivalencia, si ejercitan el código del `repOK` de la misma manera según algún criterio de cobertura de caja blanca dado. De esta manera se eliminan casos de test que cubren clases de equivalencias ya cubiertas por otros casos de test ya presentes en la *suite*.

La intención de este artículo es analizar esta técnica con nuevos casos de estudio. En la sección 4 presentamos algunos casos de estudio que muestran como la efectividad de esta técnica crece a medida que las cotas se incrementan.

2 Preliminares

Generación Exhaustiva de Casos de Test para Estructuras Complejas. Las estructuras complejas son usadas en una variada gama de aplicaciones, incluyendo aplicaciones que manipulan código fuente tales como compiladores y refactoring engines, archivos XML los cuales deben respetar reglas de buena formación, etc. La generación exhaustiva acotada de casos de test es una técnica muy usada

en muchos contextos pero particularmente útil cuando se trata de este tipo de estructuras. Ésta consiste en, dado un programa bajo prueba, generar todos las posibles entradas dentro de un rango especificado. Una característica de este enfoque es el tamaño de las *test suites* producidas, el cual es generalmente grande, inclusive para cotas pequeñas. Como ejemplo, supongamos que deseamos generar entradas para probar la rutina `merge` sobre binomial heaps; la *test suite* obtenida para cota 6 resulta en 57,790,404 entradas válidas.

Criterios de Cobertura de Código. Un criterio de cobertura de código es una forma de medir cuán adecuada es una *test suite*, es decir cuán bien sus entradas "ejercitan" el programa bajo prueba. Los criterios de cobertura se clasifican principalmente en criterios de *Caja Blanca* y criterios de *Caja Negra* [4]. Estos últimos, ven el código del programa bajo prueba como una caja negra, es decir, solo la especificación del mismo es usada para medir cuál adecuada es la *test suite*. Un ejemplo de éstos es el criterio de partición de clases de equivalencia, el cual consiste en dividir las entradas en clases de equivalencias según la especificación del programa bajo prueba.

Por otro lado, los criterios de *Caja Blanca*, se caracterizan por utilizar la estructura del código del programa bajo prueba para medir cuán adecuada es una *test suite*. Ejemplos de estos criterios son Cobertura de Decisión y Cobertura de Caminos. Una *test suite* satisface el criterio de cobertura de decisión si cada punto de decisión del código se evalúa por `true` y por `false` por algún test en la *suite*. En el caso del criterio de cobertura de Caminos, dada la representación del código fuente como un grafo, una test suite satisface este criterio si todos los posibles caminos del grafo desde el nodo de inicio al nodo final son ejercitados por algún test presente en la *suite*.

Otro criterio, también considerado de caja blanca, es *testing de mutación*. Este criterio mide la efectividad de una *test suite* según cuán buena es ésta para detectar errores en el programa. Para ello lo que se hace es insertar cambios, llamados mutaciones, al código del mismo obteniendo programas distintos al original. A cada uno de estos programas se lo denomina *mutante*. Si un mutante y el programa original se comportan de manera diferente para algún caso de test en la *suite*, el error es detectado por la *suite* (el mutante está muerto), en otro caso el mutante queda vivo [13]. Tomando un conjunto de mutantes y una *test suite* se mide el porcentaje de mutantes que ésta pudo matar.

3 Reducción de Test Suites Exhaustivas

En esta sección, explicamos la técnica mencionada, para reducir *test suite* exhaustivas, asumiendo que se posee una implementación del invariante de representación, sobre la cual la *test suite* es producida.

Esta reducción esta basada en el uso del código del `repOK` para definir equivalencias entre las entradas válidas y eliminar, de acuerdo a un índice de reducción sobre la *test suite* exhaustiva, casos de test que son equivalentes a algún otro presente en la *suite*. En primer lugar describimos, mediante un ejemplo, cómo

definir sobre el `repOK` este criterio de cobertura. Supongamos que tenemos la siguiente implementación de *Listas Simplemente Encadenadas*:

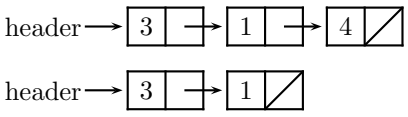
```
class SinglyLinkedList {
    private Node header;
    private int size;
    ...
}
```

```
class Node {
    private Integer elem;
    private Node next;
    ...
}
```

El invariante de representación de esta estructura debe chequear que la lista es acíclica y que su tamaño es consistente, es decir el valor del campo *"size"* se corresponde con la cantidad de nodos en la lista. La siguiente es una implementación del mismo:

```
public boolean repOK() {
    if (header == null)
        return size ==0
    Set<Node> visited = new java.util.HashSet<Node>();
    visited.add(header);
    Node current = header;
    while (true) {
        Node next = current.getNext();
        if (next == null)
            break;
        if (!visited.add(next))
            return false;
        current = next;
    }
    if (visited.size()!= size)
        return false;
    return true;
}
```

Supongamos que hemos generado, con algún mecanismo, una *test suite* exhaustiva de tamaño k y que contamos con las siguientes listas:



Ambos pueden ser considerados equivalentes según el `repOK` presentado y bajo el criterio de cobertura de decisión, ya que los dos evalúan todos los puntos de decisión de `repOK` en los mismos valores. La elección de un criterio diferente daría lugar a otra partición de los valores de entrada, por ejemplo, teniendo en cuenta cobertura de *decisión con conteo* (una variante de cobertura de decisión [2], en donde dos entradas e_1 y e_2, son consideradas equivalentes de acuerdo a un programa P si y solo si, para cada decisión de P, la cantidad de veces que e_1 evalúa en `true` (y en `false`) es igual a la cantidad de veces que e_2 evalúa

en `true` (y en `false`)), estas listas no son equivalentes ya que la cantidad de veces que cada punto de decisión es ejercitado por `true` y por `false` difiere de una lista a otra. En particular, la cantidad de veces que `next == null` evalúa en false es diferente para cada una de estas listas.

Supongamos que poseemos una rutina manipulando estas listas que deseamos probar, y que k es una cantidad de test grande para la cual no poseemos suficientes recursos para ejecutarlos a todos, solo podemos utilizar un porcentaje de ellos. Llamemos n al tamaño esperado de la *test suite* reducida (por ejemplo, n podría ser $k/10$ si esperamos una reducción de un orden de magnitud). Una reducción de esta *test suite* puede ser conseguida siguiendo los pasos detallados a continuación:

- Elegir un criterio de cobertura de caja blanca para ser aplicado sobre `repOK`.
- Determinar el número de clases de equivalencias cubiertas por las entradas de la *test suite* exhaustiva, llamemoslo c.
- Determinar un límite para la cantidad deseada de casos de test que se quiere obtener por cada clase de equivalencia, llamemoslo max_{eq}, por ejemplo, podríamos tomar $max_{eq} = n/c$.
- Procesar la *test suite* exhaustiva para dejar solo max_{eq} tests por cada clase de equivalencia.

Notar que este proceso no solo depende del criterio de cobertura elegido sino también de la implementación del `repOK`, ya que los criterios de cobertura de caja blanca están fuertemente ligados a la estructura del programa.

4 Casos de Estudios y Evaluación de resultados

En esta sección presentamos algunos casos de estudios que evalúan la técnica mencionada. Para ello hemos seleccionado algunas rutinas sobre tres estructuras de datos complejas alojadas en memoria dinámica: rutinas de inserción, eliminación y búsqueda sobre árboles binarios de búsqueda balanceados tomados de Roops benchmark [12], la rutina `longthestPath` sobre grafos dirigidos y `LisAsSet` sobre ListToSet tomada de [1]. Esta última, consiste de una lista `t` y un conjunto `s`, ambos implementados sobre listas encadenadas, con el fin de determinar si `s` es el resultado de convertir `l` en un conjunto, sin tener en cuenta las repeticiones y el orden de los elementos. Además, tal como en [2], hemos seleccionado tres criterios de cobertura de *caja blanca* para ser aplicados sobre `repOK`, ellos son: *Cobertura de Decisión*, *Cobertura de Caminos* y *Cobertura de Decisión con conteo* (variante de cobertura de decisión).

Para realizar los experimentos que se detallan a continuación, procedimos de la misma manera que en [2]: tomamos el `repOK` de cada una de las estructuras indicadas y automáticamente instrumentamos el `repOK`, para los criterios mencionados, para obtener de cada llamada del mismo sobre una entrada válida la clase de equivalencia a la cual esta entrada pertenece. Luego, corrimos este `repOK` sobre los casos de test de la *test suite* exhaustiva acotada para determinar a qué clase de equivalencia pertenecen. Por último construimos *test suites* reducidas

que seleccionan algunas entradas para cada clase de equivalencia cubrible según un criterio. Es importante tener en cuenta que debido a que estamos trabajando sobre la *test suite* exhaustiva podemos determinar exactamente cuáles son las clases de equivalencia cubribles por cada criterio.

De esta manera, reducimos las *suites* al 10% y al 1% de su tamaño original. La forma en la que la reducción es realizada asegura que los casos de test seleccionados son los primeros casos de test de cada clase de equivalencia. La selección ha sido realizada, cuando fue posible, tomando equitativamente objetos de cada clase, es decir, tomamos como máximo N/M casos de test por cada clase de equivalencia, donde N es el tamaño máximo esperado de la test suite reducida (por ejemplo, 10% del tamaño de la test suite exhaustiva) y M es el número de clases de equivalencias cubiertas por la *suite* exhaustiva. En los casos en que la *test suite* fue demasiado pequeña para tomar el 10% (1%) de los casos de test se tomó como mínimo, una entrada por cada clase de equivalencia cubierta.

Como se puede observar, este trabajo no involucra el proceso de generación de la *test suite* exhaustiva, es decir cualquier técnica podría ser usada durante la generación. Sin embargo, es importante mencionar que todos las *suite* exhaustivas sobre las cuales los experimentos se realizaron, fueron generadas usando **korat** [3].

Para medir la efectividad de esta técnica, tomamos algunas rutinas que manipulan las estructuras de datos seleccionadas y generamos mutantes para cada una de estas, para luego comparar la habilidad en cuanto a matar mutantes de cada *suite* reducida con la de la *test suite* exhaustiva. Otro resultado reportado es la habilidad de matar mutantes de las *test suites* minimales, es decir tomando un caso de test por cada clase de equivalencia (UPC). Para generar los mutantes hemos utilizado la herramienta Mujava [11, 9].

ListToSet (`ListAsSet`)

Nuestro primer caso de estudio se corresponde a la operación *"list as set"* descripta anteriormente. La tabla 1 muestra, para algunas cotas dadas, el tamaño de las test suites exhaustivas acotadas (EA), y para cada uno de los criterios de caja blanca mencionados, mostramos los correspondientes tamaños de las test suites con reducción al 10% y al 1% del tamaño original de la *suite*. Además indicamos la cantidad de clases de equivalencia cubiertas para cada criterio. Cabe mencionar que las cotas especificadas corresponden al máximo número de nodos en la lista y en el conjunto, y el número de valores enteros permitidos.

Por otro lado, la rutina `ListAsSet` fue mutada obteniendo 49 mutantes. La tabla 2 muestra la habilidad de matar mutantes de cada *test suite* indicando la cantidad de mutantes que permanecen vivos. Como se puede apreciar, los resultados obtenidos muestran que, a medida que las cotas crecen, las reducciones al 10% son tan efectivas como las *test suites* exhaustivas y en el caso de las reducciones al 1%, los resultados comienzan a aproximarse a lo óptimo.

Grafos Dirigidos

En este caso, tomamos grafos dirigidos y la operación *"longthestPath"*, la cual calcula y retorna el camino más largo en el grafo. La tabla 3 muestra, para

Cotas	EA	cob. Decisión			Cob. Decisión con Conteo			Cob. Caminos		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
0-2,0-2,3,3,3	91	6	6	6	9	9	9	9	9	8
0-4,0-4,3,3,3	91	6	6	6	9	9	9	9	9	8
0-4,0-4,4,4,3	320	23	6	6	30	16	16	30	16	10
0-5,0-5,5,5,4	5,456	286	41	6	382	48	25	382	48	10
0-5,0-5,5,5,5	24,211	1240	151	6	1555	205	25	1555	205	10

Table 1. Tamaño de las *suites* exhaustivas acotadas y las reducción basada en `repOK` para probar la rutina ListAsSet

Cota	EA	Cob. Decisión			Cob. Decisión c/conteo			Cob. Caminos		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
0-2,0-2,3,3,3	2	15	15	15	15	15	15	15	15	15
0-4,0-4,3,3,3	2	15	15	15	15	15	15	15	15	15
0-4,0-4,4,4,3	2	15	15	15	15	15	15	15	15	15
0-5,0-5,5,5,4	2	5	15	15	2	15	15	2	15	15
0-5,0-5,5,5,5	2	2	5	15	2	5	15	2	5	15

Table 2. Efectividad, con respecto a *testing de mutación*, de las *test suites* exhaustivas y de las *test suites* reducidas sobre la rutina ListAsSet (se reportan mutantes vivos).

algunas cotas (número exacto de nodos en el grafo), el tamaño de las *test suites* exhaustivas, tamaño las *test suites* reducidas y el número de clases cubiertas en cada caso.

La rutina `longthestPath` fue mutada obteniendo 78 mutantes. En la tabla 4 se reportan, para las diferentes test suites y cotas, la cantidad de mutantes vivos en cada caso.

Árboles binarios de Búsqueda Balanceados (`insert, contains and delete`)

Nuestro último caso de estudio corresponde a *Árboles de búsqueda Binarios Balanceados*, sobre la cual se desea probar las rutinas de inserción, eliminación y búsqueda. La tabla 5 muestra, para varias cotas, los tamaños de las suites obtenidas y la cantidad de clases de equivalencia cubiertas en cada caso. En este caso, las cotas indican la máxima cantidad de nodos en el árbol, el rango para el campo "size" de la estructura y el número de claves permitidas en el árbol.

En este caso, los caminos y tamaños de las test suites fueron, para algunas cotas, demasiado grandes para permitirnos hacer el análisis. Por ello, hemos considerado para este caso de estudio una version acotada de cobertura de caminos [13], en la cual no se tienen en cuenta repeticiones de nodos.

Nuevamente, se utilizó *testing de mutación* para medir la efectividad de las test suites reducidas. La cantidad de mutantes obtenidos para cada rutina fueron 117 para `insert`, 116 para `delete` y 3 para la rutina `contains`. La tabla 6 muestra los resultados obtenidos. Como se puede observar, en muchos casos, logramos

Cota	EA	Cob. Decisión			Cob. Decisión c/conteo			Cob. Caminos		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
2	382	21	2	2	34	9	9	25	25	5
3	47,672,840	2,383,647	238,369	2	3,479,393	386,427	28	4,384,880	460,407	1000

Table 3. Tamaño de las *suites* exhaustivas acotadas y las reducción basada en `repOK` para probar la rutina `longthestPath`.

Cotas	EA	Cob. Decisión			Cob. Decisión			Cob. Caminos		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2	10	12	19	19	10	16	16	12	12	12
3	2	2	2	14	2	2	11	2	2	7

Table 4. Efectividad, con respecto a *testing de mutación*, de las *test suites* exhaustivas y de las *test suites* reducidas sobre la rutina `longthestPath` (se reportan mutantes vivos).

resultados similares a los de la *test suite* exhaustiva acotada, particularmente cuando la cota (cantidad de nodos y claves permitidas) crece.

5 Conclusiones y Trabajos Futuros

Las estructuras complejas no solo forman parte de librerías de estructuras de datos, sino que también se encuentran en muchas aplicaciones; por ejemplo, aplicaciones para el análisis de sitios web, en donde estos se pueden interpretar como grafos con ciertas características. Varias técnicas han sido desarrolladas en torno a la generación automática de casos de test para estructuras complejas, en particular la generación exhaustiva acotada. Esta técnica resulta en *test suites* generalmente grandes cuyo uso no siempre es posible durante el *testing*. En este trabajo se analiza la técnica presentada en [2] para reducir estas *suites*, la cual está basada en el uso del invariante de representación de las entradas. Dicho enfoque define un nuevo criterio de cobertura de caja negra, que está basado en la definición de una relación de equivalencia sobre las entradas, usando un criterio de cobertura de caja blanca sobre la implementación del invariante de representación: si dos entradas ejercitan el `repOK` de la misma manera de acuerdo a un criterio de caja blanca, estas pueden ser consideradas equivalentes, luego esta partición es utilizada para eliminar de la *suite* casos de test que cubren clases de equivalencias ya cubiertas por otra entrada presente en la *suite*.

En [2], esta técnica es analizada utilizando diversos casos de estudios, entre ellos *RedBlackTrees*, *Binomial Heaps*, y *Listas Doblementes Encadenadas*. Estos casos de estudios muestran la efectividad de este enfoque usando *testing de mutación* para tres criterios de cobertura con diferente complejidad, en particular, una variante de cobertura de decisión, *cobertura de decisión con conteo*, la cual tiene en cuenta el número de veces que cada decisión en el programa es evaluada

Cota	EA	Cob. Decisión			Cob. Decisión C/conteo			Cob. Caminos Simples		
		10%	1%	CC	10%	1%	CC	10%	1%	CC
3,0,3,3	33	5	5	5	5	5	5	4	4	4
4,0,4,4	100	10	5	5	7	7	7	6	4	4
5,0,5,5	310	29	5	5	30	10	10	16	4	4
3,0,3,6	342	30	5	5	30	5	5	18	4	4
4,0,4,8	3208	264	30	5	278	28	7	96	18	4
5,0,5,10	25,730	1524	214	5	2116	235	10	663	84	4

Table 5. Tamaño de las *suites* exhaustivas acotadas y las reducción basada en `repOK` para probar la rutinas `remove`, `insert` and `contains`.

Cota	Oper.(Mutantes)	BE	Decision Cov.			Count. Decision Cov.			Simple Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
3,0,3,3	insert(117)	21	87	87	87	87	87	87	87	87	87
	contains(3)	0	0	0	0	0	0	0	0	0	0
	delete(116)	44	59	59	59	59	59	59	86	86	86
4,0,4,4	insert(117)	21	71	87	87	87	87	87	87	87	87
	contains(3)	0	0		0	0	0		0	0	0
	delete(116)	20	50	59	59	46	46	46	59	86	86
5,0,5,5	insert(117)	17	36	87	87	64	87	87	63	87	87
	contains(3)	0	0	0	0	0	0	0	0	0	0
	delete(116)	16	33	59	59	18	38	38	45	86	86
3,0,3,6	insert(117)	21	58	87	87	58	87	87	63	87	87
	contains(3)	0	0	0	0	0	0	0	0	0	0
	delete(116)	40	40	59	59	40	59	59	58	86	86
4,0,4,8	insert(117)	17	21	64	87	24	64	87	58	71	87
	contains(3)	0	0	0	0	0	0	0	0	0	0
	delete(116)	20	29	38	59	24	33	46	29	50	86
5,0,5,10	insert(117)	17	19	26	87	19	36	87	52	63	87
	contains(3)	0	0	0	0	0	0	0	0	0	0
	delete(116)	16	18	31	59	18	27	38	18	39	86

Table 6. Efectividad, con respecto a *testing de mutación*, de las *test suites* exhaustivas y de las *test suites* reducidas sobre las rutinas `remove`, `insert` y `contains` para avl tree (se reportan mutantes vivos).

en `true` y en `false`. En este trabajo hemos presentado tres casos de estudios, con diferente complejidad para analizar la efectividad de esta técnica. Para ello hemos utilizado los mismos criterios de cobertura mencionados. El uso del criterio de *cobertura de decisión con conteo* es importante en este contexto, debido a que existe una relación, entre el tamaño de la estructura, y el número de veces que las decisión en el `repOK` son evaluadas con un valor particular. Por otro lado, en uno de los casos, *árboles binarios de búsqueda balanceados (AVL tree)*, fue necesario utilizar una variante de cobertura de caminos, llamada *cobertura de caminos simples*, la cual no tiene en cuenta caminos en el grafo de flujo de

control con repetición de nodos. Esta variante fue necesaria ya que la cantidad de caminos, y sus tamaños, eran demasiado grandes, complicando el análisis. Los resultados muestran que, particularmente para reducciones al 10% , cuando las cotas crecen se alcanzan resultados similares a las *test suites* exhaustivas acotadas. Además, en muchos casos en donde la cantidad de mutantes que sobreviven es mayor en las *test suites* reducidas, se puede observar que la relación entre la cantidad de mutantes muertos, y el tamaño de las *test suites* obtenidas, es muy buena. Un ejemplo de esto es `insert` para *AVL Trees* con cota 5,0,5,10 en donde, si bien dos mutantes quedan vivos, en comparación con la test suite exhaustiva (17 vs 19 mutantes vivos), éste puede considerarse un buen resultado, ya que esto se logra con solo el 10% de los tests (25,730 tests vs 1524 tests).

Actualmente, nos encontramos examinando esta técnica para casos de estudios que manipulan código fuente, tales como "refactoring engines" y compiladores, para los cuales esperamos obtener resultados similares a los presentados.

Referencias

1. N. Aguirre, V. Bengolea, M. Frias and J. Galeotti, *Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs*, in Proc. of Intl. Conference on Tests and Proofs TAP 2011, LNCS 6706, Springer, 2011.
2. V. Bengolea, N. Aguirre, D. Marinov and M. Frias, *Using Coverage Criteria on RepOK to Reduce Bounded-Exhaustive Test Suites*, in Proc. of Intl. Conference on Tests and Proofs TAP 2012, LNCS 7305, Springer, 2012.
3. C. Boyapati, S. Khurshid and D. Marinov, *Korat: Automated Testing based on Java Predicates*, in Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2002, ACM Press, 2002.
4. G. J. Myers, *The Art of Sofware Testing*, 2nd. Ed., John Wiley & Sons, Inc, 2004.
5. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov, *Test Generation through Programming in UDITA*, in Proc. of Intl. Conference on Software Engineering ICSE 2010, ACM Press, 2010.
6. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
7. S. Khurshid and D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Automated Software Engineering 11(4), Springer, 2004.
8. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification and Object-Oriented Design*, Addison-Wesley, 2000.
9. Y.-S. Ma, J. Offutt and Y.-R. Kwon, *MuJava : An Automated Class Mutation System*, Journal of Software Testing, Verification and Reliability, 15(2), Wiley, 2005.
10. A. Milicevic, S. Misailovic, D. Marinov and S. Khurshid, *Korat: A Tool for Generating Structurally Complex Test Inputs*, in Proc. of Intl. Conference on Software Engineering ICSE 2007, IEEE Press, 2007.
11. MuJava, `http://www.cs.gmu.edu/~offutt/mujava/`.
12. *Roops*, `http://code.google.com/p/roops/`.
13. H. Zhu, P. Hall and J. May, *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys 29(4), ACM Press, 1997.