

# JGenTest: Generador de Test automático para Java a través de ejecución concólica

Javier Ignacio Godoy<sup>1</sup>, Hernán Czemerinski<sup>2</sup>  
<sup>1</sup>j.godoy277@gmail.com, <sup>2</sup>hczemeri@dc.uba.ar

<sup>1</sup>Estudiante, <sup>2</sup>Docente cátedra

<sup>1</sup> <sup>2</sup>Práctica Profesional Supervisada II, Licenciatura en Sistemas

Universidad Nacional de General Sarmiento, Buenos Aires, Argentina

**Resumen** Los casos de test unitarios son útiles para probar si la implementación de una función tiene el comportamiento esperado. Si bien no se puede garantizar al 100% la correcta implementación, los test nos dan mayor seguridad de su comportamiento, además de permitir que futuras modificaciones en la implementación mantengan el comportamiento esperado al momento de hacer los test. Sin embargo, la tarea de hacer los test manualmente es compleja y costosa, lo que lleva a muchos programadores a obviar esta tarea, lo cual no es nada recomendable. La herramienta presentada en este artículo permite generar automáticamente casos de test unitarios en el lenguaje Java garantizando cubrir todas las sentencias y todas las ramas (alcanzables) del programa. Esto se lleva a cabo utilizando la técnica conocida como ejecución concólica.

**Keywords:** testing, generación automática de casos de test, ejecución concólica.

## 1. Introducción

El desarrollo de software por lo general es complejo y su mantenimiento costoso. Es común que, al agregar o cambiar una funcionalidad, estemos generando un bug sin percatarnos de ello. Esto puede provocar incluso la imposibilidad de mantener un producto de software, especialmente si los encargados de hacerlo no son quienes escribieron el código original. Modificar implementaciones de terceros (o las propias luego de un tiempo de haberlas desarrollado) generalmente provoca temor de generar bugs o dejar inconsistente el sistema.

JUnit, un framework Java para la ejecución de test unitarios, suele usarse para obtener ciertas garantías de que un programa Java se comporta de la forma esperada, minimizando las consecuencias anteriormente mencionadas y mejorando la calidad del código. Lo recomendable es crear al menos un test por cada método implementado. Mientras más test unitarios para cada método se escriban, mayor seguridad de su comportamiento se tendrá. Implementar casos de test se considera esencial en las compañías de software para dar un producto de calidad a los clientes. Sin embargo, muchos programadores continúan escribiendo

código sin sus respectivos casos de test. Esto podría deberse por desconocimiento o debido al tiempo extra que requiere escribir los casos de test, aunque las ventajas que otorga lo hacen necesario. Estudios sugieren que aproximadamente el 50 % del esfuerzo de desarrollar software está representado por el testing [7], como una muestra de la importancia y el esfuerzo que requiere esta disciplina.

Existen varios enfoques para tratar la generación de casos de test automáticos. Uno de estos es la generación de inputs aleatorios a partir de los cuales se crean los casos de test. Entre sus ventajas se destacan la rapidez de generación de los test con una buena cobertura de código. Sin embargo, presenta al menos dos problemas importantes. Por un lado, genera conjuntos de test cuyos inputs son redundantes [2]. Por otro lado, la probabilidad de seleccionar aleatoriamente un input particular que detecte un bug en el código puede ser muy pequeña [2]. Por ejemplo, la rama verdadera del condicional “if ( $x == 10$ ) then ...” sólo tiene una chance de ser ejecutada sobre  $2^{32}$  si  $x$  es un input del programa de tipo entero de 32 bit que es inicializado aleatoriamente [1].

Otro de los enfoques utilizados es la ejecución simbólica: en lugar de ejecutar un programa usando valores concretos como inputs del mismo -números, por ejemplo-, se ejecuta con representaciones simbólicas para un conjunto de clases de inputs [3]. Una ejecución simbólica está representado por valores simbólicos para los parámetros de entrada y una fórmula -restricción- simbólica para el output. Cada expresión condicional en el programa representa una restricción que determina un camino en el árbol de ejecución. Nótese que cada camino, en caso de tener solución factible, determina diferentes valores como inputs. Para analizar los diferentes caminos en el árbol de ejecución, se suele utilizar backtracking. Sin embargo, para ejecuciones grandes y complejas, se vuelve computacionalmente difícil de llevar a cabo.

Un método similar a la ejecución simbólica para tratar la generación de test automáticos es el conocido como ejecución concólica. Como su palabra lo indica, es una técnica que une la ejecución concreta con la simbólica (CONCrete-symbOLIC execution); para cada valor concreto ingresado como input del programa (parámetros de una función), se registran los valores simbólicos de estos parámetros y se mantiene el valor simbólico a través de las diferentes asignaciones que se realicen en el cuerpo del programa. Al igual que en la ejecución simbólica, cada camino del árbol de ejecución determina una fórmula de las variables simbólicas representadas como restricciones para ese camino particular. En vez de utilizar backtracking, las condiciones simbólicas de un determinado camino son utilizadas en un SMT Solver para obtener nuevos valores concretos que permitan recorrer un camino alternativo del programa. Utilizando estos valores concretos como nuevos inputs del programa, y siguiendo una operación similar con el resto de los caminos del árbol de ejecución, se puede obtener un conjunto de clases de inputs que funcionen como casos de test. El presente artículo describe una herramienta que utiliza esta técnica.

## 2. Ejemplo

A continuación se describe un ejemplo de cómo la herramienta construye los test para el método Java de la Figura 1 (a).

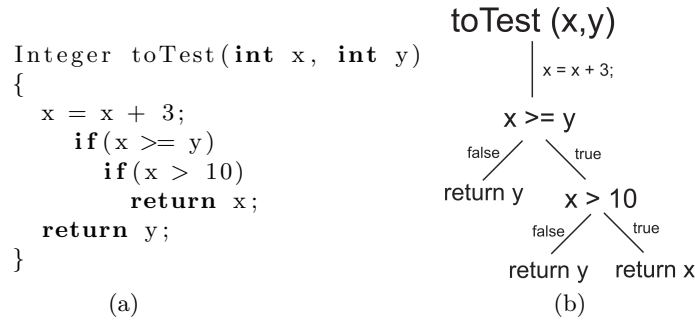
Antes de comenzar la ejecución, el código es instrumentado de forma tal de ir manteniendo un mapeo entre las variables y sus valores simbólicos. Al iniciar la ejecución sobre el código ya instrumentado, la herramienta asigna el valor cero a ambas variables. De esta forma, se tiene que  $x = 0, y = 0$  como valores concretos y  $\{x \mapsto x_0, y \mapsto y_0\}$  son las expresiones simbólicas asociadas. En la primer sentencia del método, se actualiza el valor de la variable  $x$ , por lo tanto, se actualizan las expresiones simbólicas  $\{x \mapsto x_0 + 3, y \mapsto y_0\}$ . Luego, la evaluación condicional del primer if da verdadero, y la del segundo da falso, y la ejecución de la función termina. La herramienta recolecta las restricciones de los predicados condicionales ejecutados:  $x_0 + 3 \geq y_0$  (del primer If) y  $x_0 + 3 \leq 10$  (del segundo If). Estas restricciones ( $x_0 + 3 \geq y_0, x_0 + 3 \leq 10$ ) las llamaremos restricciones simbólicas. Luego se niega el último predicado ( $x_0 + 3 > 10$ ) y se resuelven las restricciones simbólicas para obtener un camino alternativo al ya recorrido. Estas restricciones se resuelven por medio de un SMT Solver: se envían las restricciones y devuelve valores concretos para las variables simbólicas de forma tal que se satisfagan las ecuaciones. Para las restricciones  $x_0 + 3 \geq y_0, x_0 + 3 > 10$  se proponen los valores  $x_0 = 8, y_0 = 11$ . Se ejecuta otra vez el método `toTest` con estos valores, dando verdadero tanto el primer if como el segundo; por lo tanto ambas ramas ya han sido cubiertas y se descarta la condición simbólica correspondiente a esa bifurcación, quedando todavía pendiente obtener valores que hagan falsa  $x_0 + 3 \geq y_0$ . Por lo tanto se niega esta condición obteniendo  $x_0 + 3 < y_0$  y se le pide al Solver valores que la satisfaga. Éste devuelve  $x = -1, y = 3$ . Al ejecutar con estos valores el método, no se pasa el primer If, con lo que se termina de recorrer todos los posibles caminos del árbol de ejecución.

Finalmente, con los pares de valores  $\{(x = 0, y = 0), (x = 8, y = 11), (x = -1, y = 3)\}$  se crean tres test, uno por cada par de valores devuelto. En la Figura 1 (b) se muestra el árbol de ejecución del método `toTest`. Cabe destacar que cada hoja del árbol representará un conjunto diferente de valores que funcionarán como input para testear dicho método, asegurando el 100% de cobertura de código alcanzable<sup>1</sup>, tanto de sentencias como de ramas.

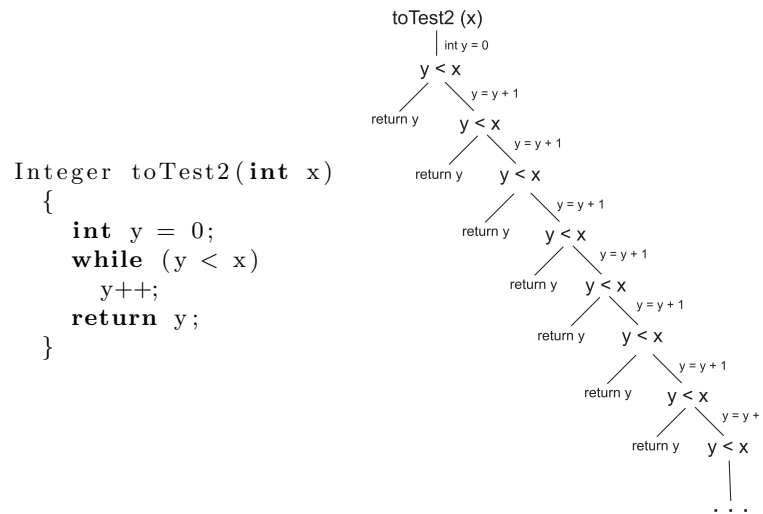
## 3. El problema de los ciclos

Es importante destacar que el enfoque propuesto presenta un problema inherente a la ejecución simbólica al tratar con código que posea ciclos. El problema es que el proceso iterativo de generación de nuevos inputs puede continuar indefinidamente. El ejemplo de la Figura 2 ilustra este fenómeno.

<sup>1</sup> Puede suceder que el método a testear contenga “código muerto”, es decir, código que jamás se ejecutará independientemente de los valores de entrada de dicho método



**Figura 1.** Ejemplo de código Java (a) junto a su árbol de ejecución (b) correspondiente



**Figura 2.** Ejemplo de código con ciclos y un árbol de ejecución infinito

La herramienta comienza la ejecución con  $x = 0$ , siendo  $x_0$  su valor simbólico. Dado que  $y = 0$ , la condición de la guarda del while no se cumple y en consecuencia se recolecta la restricción simbólica ( $0 \geq x_0$ ). A continuación se niega la condición obteniéndose ( $0 < x_0$ ) y se la entrega al SMT solver. Éste devuelve como solución  $x = 1$ . Nuevamente se ejecuta la función con este nuevo valor, cumpliéndose la primera vez el while, pero no la segunda. Por lo tanto se recolectan las condiciones ( $0 < x_0, 1 \geq x_0$ ). Luego se entregan estas restricciones nuevamente al solver, negando la última ( $0 < x_0, 1 < x_0$ ). Como resultado devuelve  $x = 2$ . Se vuelve a ejecutar la función con este valor, entrando dos veces al while, por lo que el valor de  $y = 2$ . Las restricciones simbólicas en esta ejecución son  $0 < x_0, 1 < x_0, 2 \geq x_0$ . Se niega la última condición y el solver devuelve  $x = 3$ . Esto permite que al ejecutar nuevamente la función con este valor se ingrese tres veces al while, obteniendo  $y = 3$  y por lo tanto  $0 < x_0, 1 < x_0, 2 < x_0, 3 \geq x_0$  como restricciones simbólicas. Como es de esperar, al negar la última condición

( $0 < x_0, 1 < x_0, 2 < x_0, 3 < x_0$ ) el solver devuelve  $x = 4$ . Se puede observar que este proceso de generación de inputs puede continuar indefinidamente. Por este motivo, es necesario buscar una alternativa al tratar con ciclos.

El enfoque adoptado en este trabajo es ejecutar los ciclos a lo sumo una cantidad fija de veces  $k$  (parametrizable en la herramienta). Para esto es necesario un pre procesamiento cuyo objetivo es reemplazar los ciclos presentes en los métodos por su equivalente en If's anidados una cantidad fija de veces. Veámoslo mejor con un ejemplo:

Para  $k = 1$

<pre><b>while</b> (x &lt; y + 1){   x++; }</pre>	<pre><b>if</b> (x &lt; y + 1){   x++; }</pre>
--	---

Para  $k = 2$

<pre><b>while</b> (x &lt; y + 1){   x++; }</pre>	<pre><b>if</b> (x &lt; y + 1){   x++;   <b>if</b> (x &lt; y + 1){     x++;   } }</pre>
--	--

Para  $k = n$

<pre><b>while</b> (x &lt; y + 1){   x++; }</pre>	<pre><b>if</b> (x &lt; y + 1){   x++;   //   // (n-2 IF 's)   //   <b>if</b> (x &lt; y + 1){     x++;   } }</pre>
--	---

De esta forma, el pre procesamiento permite modificar el programa original para evitar que el proceso de generación de inputs continúe indefinidamente. Este programa modificado sin ciclos será sobre el cual se lleva a cabo el proceso de instrumentación que se explicará más adelante.

## 4. Descripción del método

El objetivo del prototipo de la herramienta desarrollada es generar casos de test para las funciones/métodos de una clase Java que tenga parámetros de tipo entero. En otras palabras, la herramienta recibe como input un programa y devuelve una clase java con casos de test para los métodos deseados, lista para ejecutar con JUnit.

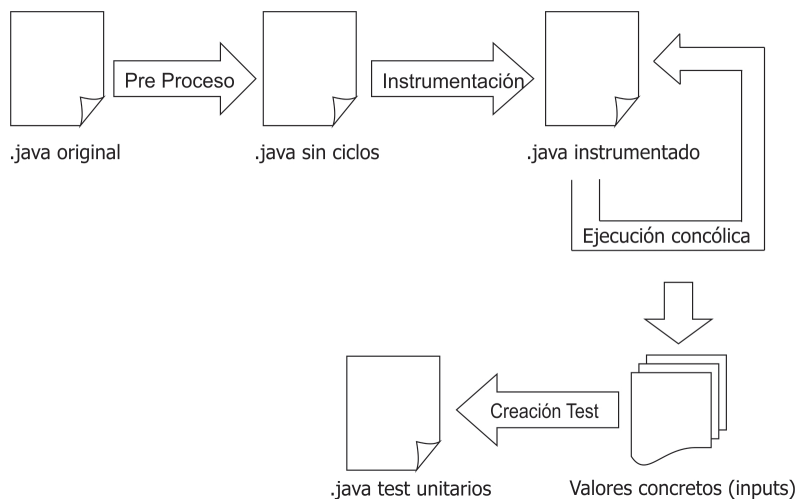
Para lograrlo se empleó la técnica conocida como ejecución concólica, donde se conjuga la ejecución de un programa, cuyos inputs son valores concretos, con la representación simbólica de estos, registrando las condiciones del programa (restricciones). Estas restricciones están determinadas por los valores simbólicos, los cuales se van actualizando según sea necesario en un Mapa lógico  $\delta$  representado inicialmente por  $\{p \mapsto p_0; q \mapsto q_0\}$  donde  $p, q$  son parámetros de entrada de un método. Si los parámetros sufren alguna modificación, también se actualiza su valor simbólico en dicho Mapa. Por ejemplo, si dentro del cuerpo del método se ejecuta  $p = p + q + 1;$ , el Mapa  $\delta$  se actualiza a  $\{p \mapsto p_0 + q_0 + 1; q \mapsto q_0\}$ .

Luego de una ejecución para valores concretos, se tiene un conjunto de ecuaciones (o restricciones simbólicas), de las cuales se niega la última de ellas. Luego, este nuevo conjunto de ecuaciones representará un camino diferente en el árbol de ejecución. Para obtener valores concretos que permitan al programa ir por dicho camino, el conjunto de ecuaciones es analizado con un SMT solver, Z3 [5], cuyo output serán los valores concretos deseados (una clase de valores que representen dichas restricciones). Con estos valores se vuelve a ejecutar el programa, obteniendo un nuevo conjunto de restricciones simbólicas que representa un camino diferente del árbol de ejecución. Este proceso finaliza luego de obtener valores concretos para cada camino diferente del árbol de ejecución. Una vez finalizado, se genera un caso de test por cada conjunto de valores.

Para obtener el conjunto de restricciones con la representación simbólica de los parámetros de entrada de la función, es necesario instrumentar el programa. Spoon [4], una herramienta para analizar y transformar código fuente Java, es la herramienta utilizada para la instrumentación. Este es un proceso complejo que se detallará en la subsección siguiente. En la Figura 3 se puede observar un esquema general de los procesos que se realizan.

### 4.1. Instrumentación

Como se mencionó antes, el objetivo de la instrumentación es obtener todas las restricciones simbólicas de los diferentes caminos del árbol de ejecución para conseguir valores concretos que ejecuten todos los caminos posibles. Esto se logra modificando el programa sin cambiar su semántica original pero agregando sentencias. Estas permiten recolectar restricciones simbólicas en base a los parámetros simbólicos de la función teniendo en cuenta las asignaciones y las sentencias condicionales presentes en el código. Una vez finalizada la instrumentación, el programa a ejecutar es el instrumentado.



**Figura 3.** Esquema de los procesos realizados para generar los casos de test

El proceso de instrumentación se realiza tomando como input el programa modificado con el pre proceso explicado en la Sección 3 debido al problema con los ciclos. De esta forma, el programa a instrumentar en lugar de contener ciclos, contiene una cantidad fija de If's en su lugar. A continuación se presenta un resumen de los procesos que lleva a cabo la instrumentación:

- Asignaciones:** se detectan y registran todas las asignaciones en el método, incluyendo declaraciones locales con algún valor asignado. Luego de cada asignación, se agregan las sentencias necesarias para actualizar en el mapa descrito anteriormente los valores simbólicos de las variables. Por ejemplo, para  $x = 2, y = 10$  el Mapa  $\delta$  se comporta de la forma que ilustra el Cuadro 1. Nótese que la segunda columna representa el momento inmediatamente posterior a la ejecución del código de la primer columna asociada.

<code>public Integer m(int x, int y) {</code>	$\delta = \{x \mapsto x_0; y \mapsto y_0\}$
<code>  x = x +3;</code>	$\delta = \{x \mapsto x_0 + 3; y \mapsto y_0\}$
<code>  if (x ≥ y ) {</code>	
<code>    x = x +78;</code>	
<code>    return x;</code>	
<code>  }</code>	
<code>  y = x - 2;</code>	$\delta = \{x \mapsto x_0 + 3; y \mapsto x_0 + 3 - 2\}$
<code>  return y;</code>	
<code>}</code>	

**Cuadro 1.** Asignaciones con su representación simbólica dado por el Mapa  $\delta$ . Método ejecutado con  $(x = 2, y = 10)$

- **Sentencias IF's:** el árbol de ejecución está representado por los diferentes caminos posibles que puede tener un programa. Estos están determinados por las estructuras condicionales presentes en el código. Un método sin condicionales, es un método con un único camino posible a ejecutar. Por esta razón, las restricciones simbólicas mencionadas están determinadas por las sentencias condicionales cuyos valores son los valores simbólicos de los parámetros de entrada, cuando corresponda. En la Figura 1 se puede observar un ejemplo de un método con su correspondiente árbol de ejecución. En el Cuadro 2 se presenta un ejemplo de cómo se recolectan las restricciones simbólicas que luego serán enviadas a un solver para obtener valores concretos. Cabe destacar que las restricciones simbólicas se recolectan para una ejecución concreta del método a testear, es decir, con valores concretos para cada parámetro de la función.

<code>public Integer m1(int x, int y) {</code>	$\delta = \{x \mapsto x_0; y \mapsto y_0\}$
<code>  x = x * 3;</code>	$\delta = \{x \mapsto x_0 * 3; y \mapsto y_0\}$
<code>  if (x == y) {</code>	$\alpha = (x_0 * 3 == y_0)$
<code>    x = x+y;</code>	$\delta = \{x \mapsto x_0 * 3 + y; y \mapsto y_0\}$
<code>      if (x ≥ 100 &amp;&amp; y &gt; 0)</code>	$\alpha = (x_0 * 3 == y_0, !(x_0 * 3 + y_0 \geq 100 \ \&\& \ y_0 > 0))$
<code>      return x;</code>	
<code>    }</code>	
<code>  if (x% 2 == 0)</code>	$\alpha = (x_0 * 3 == y_0, !(x_0 * 3 + y_0 \geq 100 \ \&\& \ y_0 > 0, (x_0 * 3 + y_0) \% 2))$
<code>    return x;</code>	
<code>    return y;</code>	
<code>  }</code>	

**Cuadro 2.** Ejemplo de recolección de restricciones simbólicas  $\alpha$ . El Mapa lógico  $\delta$  actualiza los valores simbólicos de los parámetros de entrada de la función. Método ejecutado con  $(x = 0, y = 0)$

Los restricciones  $\alpha$  serán las enviadas al SMT Solver para obtener valores concretos para  $x$  e  $y$ .

#### 4.2. Interacción con el SMT Solver

El SMT solver utilizado en este trabajo es el Z3, desarrollado por Microsoft Research. El uso de este componente es fundamental en el desarrollo de la ejecución concólica para obtener los valores a utilizar en los casos de test generados. Este solver permite chequear la satisfacibilidad de fórmulas lógicas sobre una o más teorías, otorgando valores que satisfagan dichas fórmulas cuando sea posible.

El uso de Z3 en el desarrollo de la herramienta se ha mencionado a lo largo del presente artículo: las restricciones simbólicas recolectadas mediante la ejecución del código instrumentado -un método particular con parámetros enteros-



se envían a Z3 luego de negar la última de estas restricciones <sup>2</sup>. Este solver las procesa y, de ser posible, devuelve valores concretos que satisfagan dichas restricciones (valores concretos para las variables simbólicas). Estos valores son utilizados como input del método instrumentado para recorrer un camino alternativo al anterior, recogiendo nuevas restricciones simbólicas que serán procesadas por Z3 (siempre negando la última restricción), hasta que se hayan procesado todas estas restricciones.

Usar un SMT solver tiene como principal ventaja que permite obtener valores concretos que aseguren recorrer todas las ramas alcanzables del árbol de ejecución, obteniendo 100 % de cobertura en los test -siempre y cuando no exista código muerto en el programa-. Sin embargo, presenta algunas desventajas en cuanto a limitaciones que se detallan en el apartado siguiente.

## 5. Limitaciones

A continuación se describen brevemente limitaciones que abordan dos cuestiones: limitaciones inherentes a la ejecución concólica y limitaciones del prototipo de la herramienta desarrollado que pueden superarse.

Uno de los principales límites de la ejecución concólica es la dependencia de un SMT Solver que permita resolver las restricciones simbólicas. Existe una restricción sobre las teorías que pueden incluirse que depende del SMT solver subyacente. Si bien se ha avanzado en las teorías que manejan (enteros, reales, strings, arrays, etc.), aún existen tipos no soportados. Otra limitación es el tiempo que puede demandar el Solver en encontrar una solución para ciertas ecuaciones.

En cuanto a las limitaciones del prototipo, por un lado, actualmente sólo puede generar casos de test para métodos cuyos parámetros y restricciones en el código son de tipo entero. La llamada a funciones dentro del código tampoco está soportada actualmente. Por otro lado, las operaciones soportadas son: suma (+), resta (-), división (/), producto (\*) y módulo (%).

## 6. Conclusiones y trabajos futuros

Al tratar con testing es importante tener en mente la célebre frase “El Testing sólo puede mostrar la presencia de defectos, no su ausencia” (E. W. Dijkstra). Como se ha mencionado, no es posible garantizar al 100 % que los test que se generen (independientemente de la forma adoptada) validan la correcta implementación del código. Sin embargo, estos nos permiten tener mayor seguridad en la implementación al detectar errores, además de otorgar ventajas para la mantenibilidad del código.

A lo largo del artículo se ha presentado el desarrollo de un prototipo para la generación de test automáticos para programas Java. Este tiene la ventaja de evitar a los programadores generar manualmente cada test, asegurando el 100 %

<sup>2</sup> Esto se conoce como depth first exploration y permite explorar todas las ramas del árbol de ejecución

de cobertura del código alcanzable. Para esto se ha utilizado la técnica ejecución concólica, la cual conjuga la ejecución simbólica con la ejecución concreta. Se ha puesto en evidencia, por medio de un ejemplo, el problema que presentan los ciclos en ciertas ocasiones al aplicar la ejecución simbólica: se puede incidir en un proceso infinito de generación de inputs. Se ha presentado una solución a este problema, reemplazando los ciclos por una cantidad fija (parametrizable en la herramienta) de if anidados.

Los casos de test unitarios tienen la particularidad de necesitar conocer el valor esperado para asegurar el comportamiento que se está testeando. La generación automática da como resultado test que siempre se cumplen. Su principal utilidad radica en test de regresión, permitiendo corroborar si se mantiene el comportamiento, por ejemplo, al modificar un programa una vez que se tienen los test generados automáticamente.

Se han mencionado ciertas limitaciones propias de la herramienta desarrollada. El trabajo futuro es superar algunas de estas. Actualmente sólo puede generar test automáticamente para métodos cuyos parámetros y restricciones en el código sean de tipo entero, además de soportar las operaciones aritméticas básicas (+, -, \*, /, %). Se pretende extender la herramienta a los tipos de datos y operaciones aritméticas soportadas por el SMT Solver Z3, además de soportar la generación de test sobre programas que realizan llamadas a funciones dentro del código.

## Referencias

1. P. GODEFROID, N. KLARLUND y K. SEN. *DART: Directed automated random testing*. 2005.
2. K. SEN, D. MARINOV y G. AGHA. *CUTE: A concolic unit testing engine for C*. 2005.
3. KING, JAMES C. *Symbolic Execution and Program Testing*. 1976.
4. RENAUD PAWLAK, CARLOS NOGUERA y NICOLAS PETITPREZ. *Spoon: Program Analysis and Transformation in Java*, [Research Report] RR-5901, Inria. 2006.
5. LEONARDO DE MOURA y NIKOLAJ BJØRNER. *Z3: An Efficient SMT Solver*, Microsoft Research. 2008.
6. CRISTIAN L. VIDAL, RODOLFO F. SCHMAL, SABINO RIVERO y RODOLFO H. VILLARROEL. *Una Revisión sobre la Ejecución Simbólica de Programas Computacionales*. 2014
7. GREGORY TASSEY. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. 2002.
8. JUNIT. <http://junit.org/>.
9. DANIEL PAQUÉ. *From Symbolic Execution to Concolic Testing*. 2014.
10. K. SEN y G. AGHA. *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools*. 2005.

## Apéndice A. Preproceso

Con el objetivo de otorgar a la herramienta desarrollada mayor flexibilidad en el código que acepta, se han aplicado una serie de preprocesos sobre el código fuente original. Esto se realiza ya que el framework Spoon no detecta correctamente ciertas expresiones. Por ejemplo, para la instrucción `a+=2` detecta que el valor a asignar es 2, cuando debería ser `a + 2`.

Este preproceso se encarga básicamente de dos tareas. En primer lugar, se hace una conversión de las sentencias que contienen operadores especiales de postincremento(`a++`), preincremento(`++a`), postdecremento(`a--`) y predecremento(`--a`) a su versión estándar. Esto se realiza tanto para las sentencias simples que sólo contienen estos operadores como para las sentencias de tipo declaraciones locales o asignaciones. Algo similar se realiza para los operadores de asignación `+=`, `-=`, `*=`, `/=` y `%=`. Son reemplazados por su “versión extendida”. A continuación algunos ejemplos de estas modificaciones sintácticas en el código:

- `a++` es reemplazado por `a = a + 1;`
- `int x = a++` es reemplazado por `int x = a; a = a + 1;`
- `x = --a` es reemplazado por `a = a - 1; x = a;`
- `x+=a` es reemplazado por `x = x + a;`
- `x*=2+y` es reemplazado por `x = x * (2+y);`

En segundo lugar, se hace una conversión de los ciclos tipo `for` y `do while` presentes en el código fuente a su equivalente en `while`. En el Cuadro 3 se puede observar un ejemplo.

Antes	Después
<pre>for (int i = 0; condition; ++i) {   P; }</pre>	<pre>int i = 0; while (condition) {   P;   ++i; }</pre>
<pre>do {   S; } while (condition);</pre>	<pre>S; while (condition) {   S; }</pre>

**Cuadro 3.** Transformación de `for` y `do while` a su equivalente en `while`

De esta forma, al momento de procesar los ciclos (por lo comentado en la sección 3), se puede pensar únicamente en el procesamiento de los ciclos tipo `while`, facilitando dicho procesamiento.

## Apéndice B. Herramienta gráfica

A continuación se presentan algunas características y usos de la herramienta gráfica.

Su interfaz fue desarrollada con JavaFX. Sin embargo, no se descarta la posibilidad de reemplazarla por un plugin para Eclipse, a fin de facilitar su uso. Actualmente consta de tres opciones generales:

### B.1 Buscar

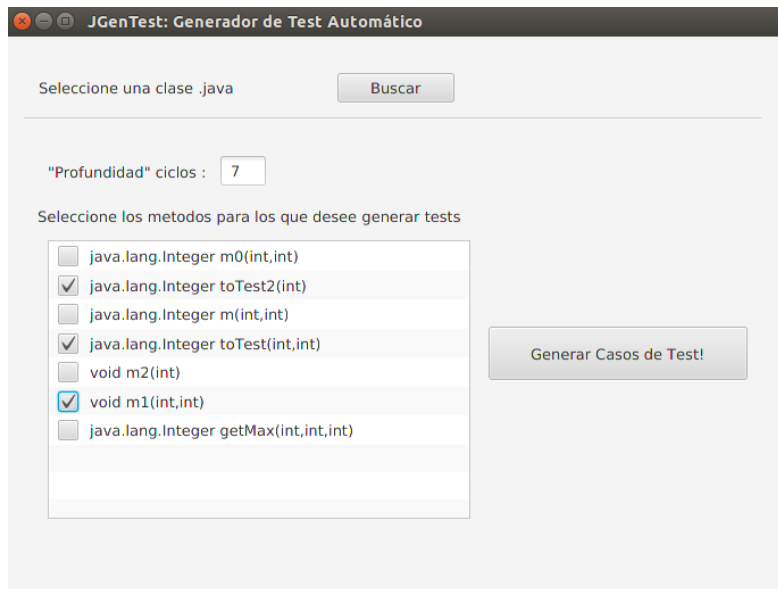
Su objetivo es seleccionar un archivo .java en el filesystem sobre la cual se desea generar los test. Por lo general, debería ser una clase perteneciente a un proyecto Java. En la figura 4 se puede observar la interfaz gráfica de la herramienta al iniciarla.



**Figura 4.** JGenTest: Imagen de la herramienta al iniciarla

### B.2 Selección de métodos

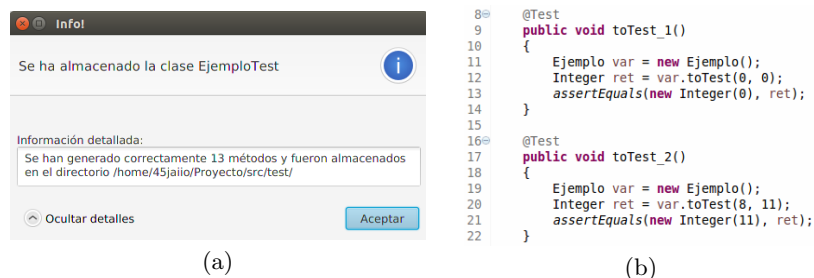
Luego de seleccionar la clase .java, la herramienta carga los métodos que existen en dicha clase, indicando el tipo de retorno y los parámetros de cada una. Además se habilita el campo **Profundidad ciclos**. Este representa el parámetro  $k$  explicado en el artículo en relación al problema con los ciclos durante la ejecución simbólica. En lugar de ejecutarse infinitamente, se ejecutan (a lo sumo)  $k$  veces, dependiendo de los valores concretos de los parámetros de entrada. En la figura 5 se muestra un ejemplo donde se seleccionan 3 métodos a testear.



**Figura 5.** Métodos seleccionados a testear. El valor 7 es la cantidad máxima de veces que se ejecutan los ciclos

### B.3 Generación de test automático

Una vez seleccionados los métodos que se desean testear, al hacer clic en **Generar casos de Test!**, se llevan a cabo los procesos indicados en la figura 3. Luego de generar los casos de test, se indica por medio de un mensaje la cantidad de test generados y el lugar donde se almacenó. Por defecto, se almacenan en la misma carpeta donde se encuentra el archivo .java seleccionado en el primer paso.



**Figura 6.** (a)Mensaje de generación correcta de los test, indicando la ubicación del archivo y la cantidad de test generados. (b)Ejemplos de test generados

## Apéndice C. Análisis de cobertura

Para verificar la cobertura de los test generados, se ha utilizado el plugin para Eclipse EcEmma. Este permite observar de forma sencilla el código que se ha cubierto con los test. Para los test generados anteriormente se puede observar que se ha cubierto el 100% del código:

(a)

```

1 package test;
2 import static org.junit.Assert.assert
3 import org.junit.Test;
4
5 public class EjemploTest
6 {
7
9 public void toTest_1(){}
15
17 public void toTest_2(){}
23
25 public void toTest_3(){}
31
33 public void m1_1(){}
38
40 public void m1_2(){}
45
47 public void toTest2_1(){}
53
55 public void toTest2_2(){}
61
63 public void toTest2_3(){}
69
71 public void toTest2_4(){}
77
79 public void toTest2_5(){}
85
87 public void toTest2_6(){}
93
95 public void toTest2_7(){}
101
103 public void toTest2_8(){}
109 }

```

(b)

```

package test;
public class Ejemplo
{
    public Integer toTest(int x, int y)
    {
        x = x + 3;
        if (x >= y)
            if (x > 10)
                return x;
            return y;
    }

    public Integer toTest2(int x)
    {
        int y = 0;
        while (y < x)
            y++;
        return y;
    }

    public void m1(int x, int y)
    {
        if (x > 2)
        {
            y = x + 1;
        } else
        {
            y = y + 1;
        }

        for (int i = 0; i < 10; i++)
        {
            if (x <= 2)
            {
                y = y + 1;
            } else
            {
                x = x + 2;
            }
        }
    }
}

```

**Figura 7.** (a)Test Generados. (b)En verde, código cubierto con los test