Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura

Implementación y Análisis de la Técnica de Mutation Testing en Ambientes de Objetos Dinámicos y Reflexivos

Autores

Brunstein, Gabriel
LU: 447/03
gaboto@gmail.com

Chillo, Nicolás Agustín
LU: 82/03
nchillo@gmail.com

Director

Lic. Hernán Wilkinson

Julio, 2010

Agradecimientos

Queremos agradecer a Hernan Wilkinson por habernos propuesto este tema de tesis, por su dedicación y apoyo y, por sobre todas las cosas, sus aportes realizados tanto para esta tesis como en el cursado de las materias Programación Orientada a Objetos y Diseño Avanzado de Objetos que fueron las bases fundamentales de este trabajo.

Agradecemos también a nuestros amigos y compañeros con los que cursamos y compartimos horas de estudio, trabajos y por sobre todo buenos momentos. Y a nuestros compañeros de trabajo que nos apoyaron en el día a día y nos incentivaron a terminarla.

Agradezco a mis padres Daniel y Silvia por su amor, cariño y comprensión y, sobre todo, por el apoyo que me dieron en cada decision que tomé.

A mis hermanas Natalia, Agustina y Camila y a mi abuela Haydeé porque sin ellas no sería quien soy.

A mis familiares y amigos que me soportaron y distrajeron cuando más los necesitaba.

Nicolas

Agradezco a mis padres Adriana y Osvaldo, por haberme apoyado y motivado desde un principio, sin ellos no hubiera podido realizar la carrera.

A mi novia Patricia, por acompañarme en estos últimos años con amor y paciencia.

A mis hermanos Daiana y Ari, por estar siempre.

A mis amigos y compañeros, con quienes compartí momentos increíbles.

Gabriel

Finalmente queremos mencionar a la Universidad de Buenos Aires por brindarnos la posibilidad de estudiar en una universidad pública y gratuita de gran categoría.

Resumen

Durante los 70, *Mutation Testing* surgió como una técnica para medir la efectividad de un conjunto de tests. La misma consiste en mutar el código fuente del programa que está siendo probado para modificar su comportamiento y verificar si los tests "matan" a estos mutantes. Los mutantes que sobreviven son entonces el punto de partida para mejorar o extender la suite de tests.

Sin embargo, esta es, en principio, una técnica de "fuerza bruta" por lo que su utilización ha quedado relegada debido a sus excesivos tiempos de respuesta. Este problema, sumado a la carencia de herramientas adecuadas, han impedido en la práctica su uso masivo.

En este trabajo se implementó un set de herramientas para realizar Mutation Testing en Smalltalk, un ambiente reflexivo y dinámico con definición meta-circular. Aprovechando estas características, se pudo llevar a cabo una implementación simple y eficiente, contando con la ventaja adicional de no requerirse grandes tiempos de compilación y enlace para la generación de mutantes, como suele suceder en ambientes estáticos.

A lo largo de esta investigación se lograron tres resultados. En primer lugar, reducir considerablemente el tiempo de ejecución de la técnica. Para lograrlo, se crearon diferentes estrategias que utilizan información obtenida de un análisis previo de *Code Coverage*.

En segundo lugar, se implementaron herramientas integradas al resto del ambiente de desarrollo, preparadas para realizar análisis rápidos y ágiles, de porciones o de la totalidad del programa, y facilitando su uso en conjunto con el proceso de desarrollo.

Por último, se definió y utilizó una heurística para definir operadores de mutación que brinden información más precisa para mejorar la calidad de los tests.

Abstract

During the '70s, *Mutation Testing* emerged as a technique to assess the fault-finding effectiveness of a test suite. This technique mutates the source code of the program being tested in order to change its behavior, and verifies whether the tests "kill" those mutants or not. The surviving mutants are then the starting point to improve or extend the test suite.

However, since it is, in principle, a "brute force" technique, its use has been relegated due to its excessive response times. This issue, along with the lack of adequate tools, has greatly hindered its widespread use.

In this research, a set of tools was implemented to run Mutation Testing in Smalltalk, a reflexive and dynamic environment with meta-circular definition. Using these environment features to our advantage, a simple and efficient implementation was achieved, with the additional advantage of not requiring long compilation and linking times for the generation of mutants, as is frequently the case with static environments.

Three results were obtained throughout this research. Firstly, the execution time of the technique was reduced. In order to achieve this goal, different strategies, that use information obtained from a previous *Code Coverage* analysis, were created.

Next, integrated tools were implemented in the development environment, which are ready to perform quick analyses of the whole program or portions of it, and facilitate its use in conjunction with the development process.

Lastly, a heuristic was established and used to define mutation operators that provide more accurate information to improve the quality of the tests.

Índice

Índice de figuras

1. Introducción

Uno de los objetivos fundamentales dentro del proceso de desarrollo de software es asegurar la calidad del producto generado. Para lograrlo, existen distintas herramientas y técnicas que permiten testear los desarrollos producidos. Los *Tests de Unidad*, por ejemplo, permiten verificar que unidades individuales de comportamiento funcionen adecuadamente en los casos probados. Al aplicar correctamente esta técnica de testing se puede asegurar cierta confiabilidad sobre el código que esta siendo testeado. Sin embargo, ¿Cómo es posible determinar si los tests de unidad implementados son realmente buenos y lo suficientemente completos como para asegurar esa confiabilidad deseada?

Se sabe que los test no permitirán asegurarnos que el software producido se encuentre libre de errores. Como dijo Dijkstra[Dij72]: *El testing puede ser una técnica muy efectiva para mostrar la presencia de errores, pero es, lamentablemente, inadecuada para mostrar su ausencia* [1]. Sin embargo, sí se puede aprovechar la ventaja que tienen los tests de mostrar la presencia de errores para mejorar su efectividad.

El presente trabajo se centra en la implementación y experimentación de la técnica de *Mutation Testing* en un ambiente de objetos dinámico y reflexivo. Esta técnica, tiene como principal objetivo medir la calidad de un conjunto de tests sobre un modelo de software. Se utiliza, para ello, un ambiente de objetos dinámico y reflexivo que facilita la implementación y el análisis de forma de lograr una herramienta integrada con el ambiente de desarrollo.

Pocas son las herramientas o técnicas que permiten hoy en día tomar mediciones de nuestros tests. Una de ellas bastante conocida y prácticada es *Code Coverage*. Esta técnica consiste en medir el *coverage* (cubrimiento) de los tests analizando qué porcentaje de colaboraciones del programa que está siendo testeado se evalúan como consecuencia de correr los tests. Esta técnica es utilizada muchas veces con el objetivo de obtener métricas. Sin embargo, tener un cubrimiento del 100 % no asegura que realmente se esté testeando suficientemente un programa. Con medir coverage es posible determinar qué partes del código son abarcadas por un conjunto de tests, pero no determinan la completitud de los mismos. Por ejemplo, supongamos que tenemos la clase DebitCard con el siguiente método de igualdad:

```
DebitCard>>= anotherDebitCard
    ^(type = anotherDebitCard type) and: [ number = anotherDebitCard number ]
```

Supongamos también que tenemos el siguiente test de unidad:

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual
   self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

Al medir Code Coverage sobre el test obtendríamos una cobertura del 100 %; sin embargo, si analizamos el test descubriremos que no es lo suficientemente completo como para cubrir todas las alternativas posibles. Si, por ejemplo, el método es modificado quitando la segunda parte de la igualdad:

```
DebitCard>>= anotherDebitCard
    ^(type = anotherDebitCard type)
```

el test seguiría pasando y cubriendo el 100 % del código y sin embargo es evidente que el comportamiento del programa sería distinto. Por lo que el test en cuestion no es lo suficientemente completo.

Existe, a su vez, otra técnica llamada *Mutation testing* que consiste en introducir pequeñas modificaciones al código fuente del programa original intentando simular posibles fallas. El objetivo de esta técnica es que los tests detecten estas "fallas controladas" introducidas y de esa forma poder medir los tests implementados. Con esta técnica, es posible profundizar el análisis de los tests y obtener una retroalimentación a la hora de mejorar la calidad de los mismos. Cuantas más "fallas controladas" se detecten, mayor será la cantidad de errores detectables por los tests, lo que implicará una mayor calidad del software producido.

[1] Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence

Si bien ya existen algunas herramientas de mutation testing implementadas sobre ciertos lenguajes, no existe en la actualidad ninguna implementación integrada en un ambiente de objetos dinámicos. Es por esto que se decidió implementar *MuTalk*, un conjunto de herramientas de mutation testing en *Smalltalk* que, aprovechando las capacidades reflexivas del ambiente, permite que esta técnica brinde retroalimentación en tiempos aceptables integrándolas al ambiente de programación estándar.

A su vez, con el objetivo de mejorar los tiempos de respuesta, se combinó la técnica de Mutation Testing con la de Code Coverage y se realizó un análisis de las fallas controladas a introducir. De esta forma, se logró una herramienta completa que permite generar mediciones de los tests.

Para detallar el trabajo realizado, comenzaremos por explicar en el capítulo 2 los conceptos básicos referidos a la técnica de Mutation Testing y las características de los ambientes de objetos dinámicos y reflexivos. A partir de allí realizaremos, en el capítulo 3, una breve recorrida por los principales trabajos existentes para finalizar en los objetivos del trabajo de tesis en el capítulo 4. Una vez presentados los objetivos, detallaremos en el capítulo 5 la solución propuesta y finalizaremos en los capítulos 6 y 7 con las conclusiones y los trabajos futuros.

2. Conceptos básicos

2.1. Mutation Testing

2.1.1. Historia

La idea de *Mutation Testing* se concreta a fines de los 70 cuando DeMillo, Lipton y Sayward presentan el paper "*Hints On Test Data Selection: Help for the Practicing Programmer*"[RAD78] donde describen dos principios fundamentales: *la hipotesis del programador competente* y el *efecto de acoplamiento* (coupling effect). El primero de los principios afirma que los programadores son en general competentes y que producen programas cercanos al programa correcto. El segundo principio, el efecto de acoplamiento, afirma que los tests que distinguen programas con errores menores son tan sensibles que permiten distinguir programas con errores más complejos.

Fue a partir de estos dos principios que los autores decidieron construir un sistema que permita medir la efectividad de los tests mediante la introducción de pequeñas modificaciones del programa original (lo que denominarémos *mutaciones*) y luego comprobando si esa modificación podía ser detectada utilizando los tests del sistema.

Así surgió la técnica de *Mutation Testing*. A partir de entonces se han escrito varios trabajos y desarrollado varias aplicaciones para distintos lenguajes de programación[Webb] pero ninguno se ha ocupado de hacerlo para un ambiente de objetos dinámico y reflexivo, como por ejemplo *Smalltalk*.

A pesar de su relativamente larga historia, esta técnica no logró establecerse con suma fuerza en la industria. Esto se debió principalmente a tres causas [OU00]:

1. Falta de incentivo económico destinados a mejorar la calidad del proceso de testing.
2. Incapacidad de integración del proceso de testing en el desarrollo de software.
3. Carencia de tecnología que permita llevar a cabo el análisis de forma automática y económica.

Con respecto a la primera causa, se puede decir que la tendencia está cambiando. A lo largo de los años, el desarrollo de software ha madurado y se ha constituido fuertemente sobre diversas áreas, muchas de las cuales requieren una alta confiabilidad de los programas desarrollados. A partir de entonces, se ha comenzado a darle mayor importancia a la calidad de los desarrollos destinando gran parte de los recursos en mejorar la calidad de los programas producidos.

En el caso de la segunda causa el panorama ha cambiado mucho. Actualmente es común, en el desarrollo de software, contar con una gran batería de tests que tienen el propósito de no perder la confiabilidad del programa a la hora de introducir cambios. Para ello se requirió adaptar el proceso de desarrollo de sofware integrando el proceso de testing al proceso diario. Esta integración generó técnicas de desarrollo guiadas por los test bastante usadas en la actualidad (como es el caso de *Test Driven Development*)[Bec03].

Finalmente, la falta de automatización fue una de las causas que mayor rechazo generó. Como se muestra en la figura 1, inicialmente el proceso de Mutation Testing involucraba muchas tareas a ser realizadas en forma manual lo cual generaba un gran overhead en el trabajo de testing[Off95] relegando importancia al testing del sistema.

Como veremos a continuación, este trabajo de tesis tiene como principal propósito atacar esta última causa a través de la implementación de herramientas de Mutation Testing integradas en el ambiente de desarrollo que permitan facilitar la utilización de esta técnica para adoptarlo en el proceso de desarrollo diario.

2.1.2. Descripción de la técnica de Mutation Testing

Como se mencionó anteriormente, la técnica de Mutation Testing consiste en crear nuevas versiones del programa original. Cada nueva versión, denominada *mutante*, contiene una única modificación

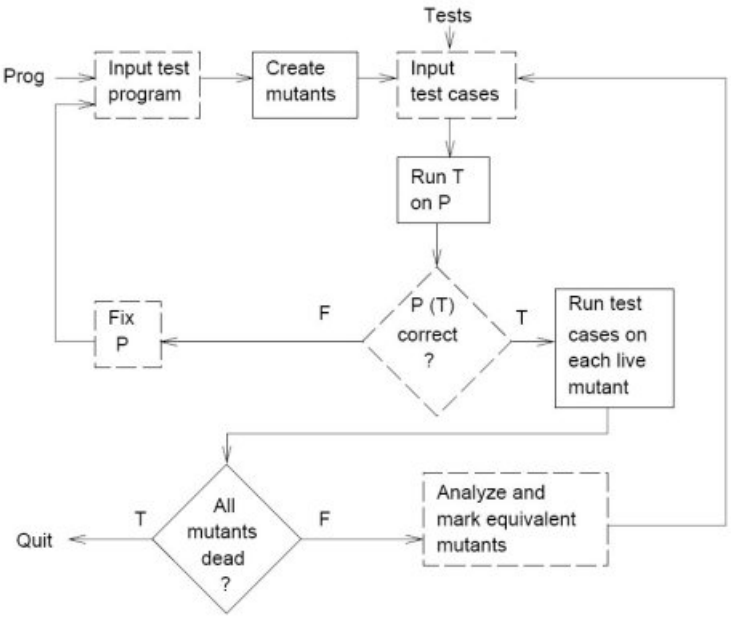

Figura 1: Proceso de Mutation Testing [OU00]. Los recuadros con guiones son tareas a ser realizadas en forma manual

respecto al programa original. La idea principal de la técnica es ejecutar los tests del sistema sobre las mutaciones con el fin de distinguir estos mutantes del programa original.

Para comprenderlo analicemos el siguiente diagrama:

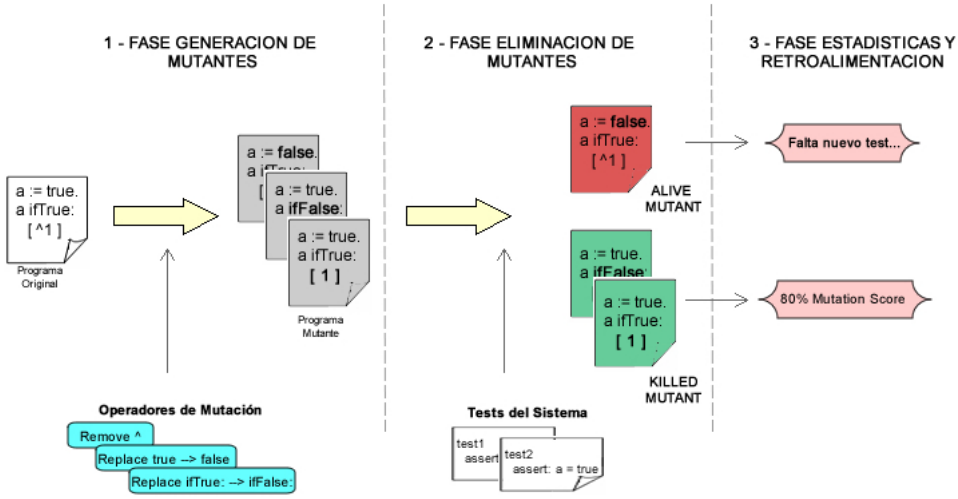

Figura 2: Fases del proceso de Mutation Testing

La primera fase del proceso de mutation testing (*Fase de generación de Mutantes*) se encarga de generar los mutantes. Los mutantes son creados aplicando *operadores de mutación* sobre el programa original. Un operador de mutación es un tipo de modificación controlada que es posible aplicar a un programa con el objetivo de generar una versión modificada del mismo. Cada modificación respecto al programa original genera un mutante. Por ejemplo el operador de mutación de *Replace and: for or:* aplicado al método:

```
AnObject>>= anotherObject
    ^aCondition and: [ anotherCondition ]
```

generaría el siguiente mutante:

```
AnObject>>= anotherObject
    ^aCondition or: [ anotherCondition ]
```

Para que esta técnica sea efectiva es necesario realizar una buena selección de operadores que permita cubrir un gran rango de errores utilizando la menor cantidad de mutantes posibles. Esto se debe a que cuanto más mutantes se creen, mayores son los tiempos de procesamiento; por lo que es fundamental no generar mutantes sin sentido.

Los operadores de mutación podrían llegar a ser desde pequeñas fallas hasta grandes cambios. Por ejemplo, algunos operadores podrían ser:

- Cambiar operadores aritméticos. Ej: reemplazar $A + B$ por: $A–B$, A/B, $A * B$.
- Modificar expresiones, agregando, cambiando ó quitando operadores relacionales. Ej: reemplazar $A < B$ por: $A > B$, $A = B$, $A! = B$.
- Modificar, eliminar, agregar operadores lógicos. Ej: reemplazar A *and* B por: A *or* B. Reemplazar A por *!A* (negación de a).
- Cambiar el envío de un mensaje *select:* a una colección por *reject:*.

Al aplicar operadores de mutación se determina el conjunto de mutantes que deberán ser descartados comparando el comportamiento del mutante con el del programa original. La idea de los operadores es simular pequeñas modificaciones que faciliten la detección de casos no previstos o testeados.

Una vez generados los mutantes, la segunda fase del proceso (*Fase de Eliminación de Mutantes*) es determinar si los tests con los que la aplicación cuenta distinguen los programas mutantes del programa original. Un mutante es distinguido si se detecta un comportamiento diferente al esperado. En dicho caso, se puede considerar que la mutación que se aplicó afectó la ejecución normal del programa por lo que será necesario descartarlo. Acá entran en juego los tests del sistema. Una vez generados los mutantes, se deberán correr los tests del sistema para cada uno de ellos. Al correr los tests puede pasar lo siguiente:

- Si al correr los tests del sistema para un mutante determinado al menos un tests falla se dice que el mutante es *matado* (Killed Mutant). Que falle un test, significa que no se cumplió alguna aserción, que se produjo alguna excepción no controlada o que el programa quedó ejecutando un ciclo infinito. Cualquiera de estas alternativas es distinta respecto al comportamiento esperado del sistema (que el test sea correcto) por lo que se podría decir que esa falla introducida en ese mutante es detectada con los tests que cuenta el programa.
- Si ningún tests del sistema falla al ejecutarlos sobre el mutante se dice que el mutante *sobrevivió* (Alive Mutant). Si un mutante sobrevive podría significar que estamos ante un mutante que alteró el comportamiento del programa original y sin embargo no fue detectado. De ser así, esa modificación introducida que produjo el mutante pasaría desapercibida con los tests actuales.

Es importante destacar que no siempre que un mutante sobrevive significa que faltan casos a ser testeados. Podría ocurrir, por ejemplo, que los operadores de mutación generen *mutantes equivalentes* al programa original, es decir, mutantes que tienen el mismo comportamiento que el programa original. Estos casos son falsos positivos y se verán algunos ejemplos más adelante durante el desarrollo de este trabajo.

Finalmente, la última fase del proceso (*Fase de Estadísticas y Retroalimentación*) es utilizar las métricas generadas por la técnica y analizar los mutantes que sobreviven con el objetivo de medir nuestros tests y mejorarlos.

Para ello, una vez que se generaron todos los mutantes deseados se podrá evaluar los test del sistema con cada uno de ellos y con el resultado de este proceso se podrá obtener métricas que permitan medir

la calidad de los tests implementados. Por ejemplo, una métrica podría ser la relación entre la cantidad de mutantes eliminados respecto a la cantidad de mutantes generados. Esta relación es conocida como *Mutation Score*. Si se obtiene un Score de 100 %, significa que los tests con los que contamos están capturando todos los errores introducidos. En cambio, si el Score es 0 %, significaría que los tests no están detectando ninguno de los errores introducidos.

A su vez, analizando cada mutante que sobrevivió se podría ver un posible error en el programa que no sería detectado con los tests que contamos. Para solucionar este problema a veces es necesario desarrollar nuevos casos de tests, mejorar los existentes o simplemente corregir el programa. Esto genera una retroalimentación que permite mejorar el sistema generado o la suite de tests del sistema.

2.1.3. Ejemplo de aplicación de la técnica de Mutation Testing

Para comprender la técnica en su totalidad, analicemos el siguiente ejemplo.

Supongamos nuevamente que estamos creando un modelo de tarjetas de débito donde cada tarjeta es una instancia de la clase **DebitCard** y tiene como colaboradores internos al tipo de la tarjeta y al número [2].

```
Object subclass: #DebitCard
        instanceVariableNames: 'type number'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Testing'
```

Supongamos también que definimos la igualdad de las tarjetas de débito comparando las tarjetas por su tipo y número:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
         and: [ number = anotherDebitCard number ]
```

y supongamos, a su vez, que tenemos los siguientes casos de test donde se verifica la igualdad de 2 tarjetas de débito con igual tipo y número y con igual tipo y diferente número respectivamente:

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual

        self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

```
DebitCardTest>>testDebitCardWithDifferentNumberShouldBeDifferent

        self deny: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 789).
```

Si aplicamos la técnica de mutation testing usando como único operador de mutación el operador que reemplaza el mensaje `#and:` por `#or:` obtendríamos el siguiente mutante:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
         or: [ number = anotherDebitCard number ]
```

que al ser evaluado sobre los tests existentes moriría, pues el primer test (*testDebitCardWithSameNumberShouldBeEqual*) seguiría siendo válido pero no así el segundo (*testDebitCardWithDifferentNumberShouldBeDifferent*). Por lo tanto, aplicada la técnica de Mutation Testing sobre la igualdad de tarjetas de débitos antes definida generaría un único mutante que fue eliminado por los tests correspondintes. Nuestro *Mutation Score* sería entonces del 100 % (1 mutante generado, 1 mutante eliminado).

[2] Este modelo es una simplificación del modelo real. Solo se implementa lo necesario para que el ejemplo tenga sentido.

2.2. Ambientes Dinámicos y Reflexivos

Como ya se dijo en páginas anteriores, uno de los propósitos de esta tesis es utilizar la técnica de Mutation Testing en un ambiente dinámico y reflexivo. Un ambiente dinámico y reflexivo permitiría integrar en forma eficiente la técnica de Mutation Testing en el proceso de desarrollo de Software. Antes de explicar por qué, es necesario explicar qué significa que un ambiente sea dinámico y reflexivo; y qué beneficios tiene un ambiente de este tipo con respecto a uno sin estas características.

2.2.1. Dinamismo

Los ambientes dinámicos brindan más libertades para cambiar el comportamiento de los programas en tiempo de ejecución. Para permitir este cambio de comportamiento dinámico, en estos ambientes, cuestiones como el chequeo de tipos y el *binding* de métodos, se resuelven lo más tarde posible. En los ambientes estáticos en cambio, muchas de estas etapas se llevan a cabo al realizarse la compilación de todo el programa.

En la mayoría de los ambientes de programación de hoy en día, y principalmente en los estáticos, para cambiar el comportamiento de un programa, es necesario editar el código fuente (comúnmente en archivos), recompilarlo y reiniciar el programa. Este esquema hace naturalmente difícil modificar un programa estando éste en ejecución.

Existen, a su vez, casos como el de Smalltalk [GR83] o Self [US87] en donde no existe un tiempo diferente al de ejecución. Los objetos están vivos en una imagen persistente del programa en la cual se puede programar modificando la definición de los objetos sin necesidad de detener ninguna ejecución [NBD+05]. Esta característica, resulta en lenguajes naturalmente más dinámicos.

Por otro lado, otra característica que aporta al dinamismo de un lenguaje o ambiente, es la reflexividad, que explicaremos a continuación.

2.2.2. Reflexividad

La reflexión [Bou] es la habilidad que tiene un sistema para poder actuar sobre si mismo.

Un sistema reflexivo incorpora estructuras *casualmente conectadas* que representan aspectos de sí mismo. Es importante destacar que el requerimiento de que la conexión sea casual es fundamental: un sistema se dice casualmente conectado a su dominio si la estructura interna y el dominio que representan están relacionado de tal manera que si uno cambia, esto implica su correspondiente efecto en el otro [Mae87].

Se puede particularizar esta definición diciendo que un ambiente de programación reflexivo tiene una representación de su propia estructura y comportamiento disponible desde si mismo y, además, la representación cambia si la estructura y lenguajes cambian y viceversa.

La reflexividad permite realizar computaciones sobre computaciones, también llamadas *meta - computaciones*. Ambos tipos de computaciones se realizan en diferentes niveles conceptuales: el nivel base y el meta-nivel. Además, ambos niveles están casualmente conectados [nFBD+08].

Con metaprogramación (programación en el meta-nivel), entonces, un sistema puede, en menor o mayor medida, observar, analizar y modificar su propia ejecución. Por ejemplo, en un ambiente de objetos podría, programáticamente y estando el programa en ejecución, modificarse el comportamiento de un objeto, analizar los mensajes que recibe, analizar o modificar su estado, etc.

Esta característica de poder observarse y modificarse le da a los ambientes de este tipo un plus para desarrollos de herramientas que involucren integración con los ambientes de desarrollo. Este es el motivo por el cual durante este trabajo de tesis se utilizó un ambiente reflexivo. Más adelante veremos ejemplos concretos de la utilización de metaprogramación.

Existen en la actualidad varios lenguajes dinámicos y reflexivos. Algunos de ellos son LISP, Perl, Ruby y Smalltalk. Para este trabajo de investigación utilizamos Smalltalk ya que consideramos que se trata de un lenguaje de objetos maduro cuyas características dinámicas y reflexivas dadas por su

definición meta-circular facilitan la implementación e integración de herramientas de desarrollo.

2.2.3. Smalltalk

Smalltalk [GR83] es un ambiente de objetos que tiene la cualidad de ser dinámico y reflexivo. Como mencionamos anteriormente, los objetos están vivos en la imagen y no existe un tiempo diferente al de ejecución. Además, es dinámico en la mayoría de los aspectos: es dinámicamente tipado, la ejecución de métodos se resuelve por late-binding y cuenta con facilidades reflexivas gracias a su implementación meta-circular que permite tener gran control sobre el ambiente de desarrollo.

Con más detalle entonces, podemos decir que Smalltalk logra el dinamismo a través de las siguientes propiedades:

- Posee un **lenguaje dinámicamente tipado**, esto significa que el chequeo de tipo de una variable se hace en tiempo de ejecución y no de compilación. Es decir, las variables no tienen tipo, lo que implica que, cuando se produce un error de tipos un objeto puede recibir un mensaje que no sabe responder y eso se determina estando el programa en ejecución. Esto agiliza el desarrollo en comparación con ambientes con tipado estático, en donde las variables requieren la definición de su tipo y, generalmente, en la etapa de compilación del programa se utiliza esto para hacer un chequeo de errores de tipado, previniendo cualquier ejecución hasta que esto no se encuentre corregido.
- El algoritmo de method lookup asegura el **late-binding**, esto significa que cuando un objeto recibe un mensaje, el metodo que se va a evaluar se determina en tiempo de ejecución (y no de compilación). Esta característica permite el uso de polimorfismo sin restricciones.
- Posee **compilación granular**, es decir, la compilación se produce al guardarse un método modificado sin necesidad de detener la ejecución y únicamente compilando el código fuente del método en cuestión. Como veremos en la implementación de las herramientas de esta tesis, esta compilación puede realizarse programáticamente.
- Es Reflexivo, ya que posee un metamodelo completo y abierto.

Con respecto a la reflexividad, cabe destacar que, en la actualidad, la mayoría de los lenguajes de programación orientados objetos proveen acceso a su representación. En algunos casos, por ejemplo en los lenguajes estáticos como *Java* o C#, estas descripciones pueden ser consultadas. Sin embargo, en algunos lenguajes dinámicos y reflexivos (como *Smalltalk*) estas representaciones pueden no solo ser consultadas sino también modificadas.

En Smalltalk, esta posibilidad esta dada gracias a su definición *meta-circular*. Esto significa que al igual que el nivel-base, el nivel-meta está formado en término de objetos (*meta-objetos*) que son reificaciones de los elementos del programa (ejecución y estructura), los cuales saben responder mensajes como cualquier otro objeto del ambiente.

En términos prácticos, esta reflexividad resulta de gran utilidad en la implementación de herramientas como las del presente trabajo. Por ejemplo, con técnicas de metaprogramación se puede medir el cubrimiento de los tests, modificar clases y métodos en tiempo de ejecución (para instalar mutantes por ejemplo), compilar código en tiempo de ejecución, etc.

Además, en Smalltalk no solo existen los objetos del metamodelo, sino que el mismo ambiente de desarrollo está programado en sí mismo, con el mismo lenguaje y con objetos en la imagen. Esto brinda una manera abierta y flexible de modificar y/o extender este mismo ambiente pero a su vez, como veremos más adelante, al contar con esta libertad se pueden comenter errores que deriven en la desestabilización del sistema.

3. Trabajos Existentes Relacionados

La idea original de mutation testing surgió en el año 1971 con un trabajo de Richard Lipton titulado "Fault Diagnosis of Computer Programs". Sin embargo, comúnmente se cita como trabajo fundacional al paper "Hints On Test Data Selection: Help for the Practicing Programmer" DeMillo, Lipton y Sayward en donde previamente a mencionar la técnica, se definen los dos principios fundamentales: "la hipótesis del programador competente" y "el efecto de acoplamiento" que se mencionan en páginas anteriores de este informe.

Por varios años, la técnica de Mutation Testing estuvo en desuso y no hubo mucho trabajos sobre la misma (por razones ya expuestas en la Sección 2). Sin embargo, durante los años 90 y gracias a los avances tecnológicos la cantidad de trabajos y herramientas aumentaron lo que permitieron comenzar a utilizar e investigar con mayor auge esta técnica hasta entonces casi relegada.

Al respecto, Yue Jia y Mark Herman ([JH09]) escribieron recientemente un reporte técnico que consolida e integra de forma muy completa los avances y trabajos existentes de la técnica de mutation testing. Allí se puede observar el crecimiento que tuvo esta técnica a lo largo de los años (ver figura 3) y analizar las distintas optimizaciones utilizadas para mejorar los tiempos de ejecución (ver figura 4).

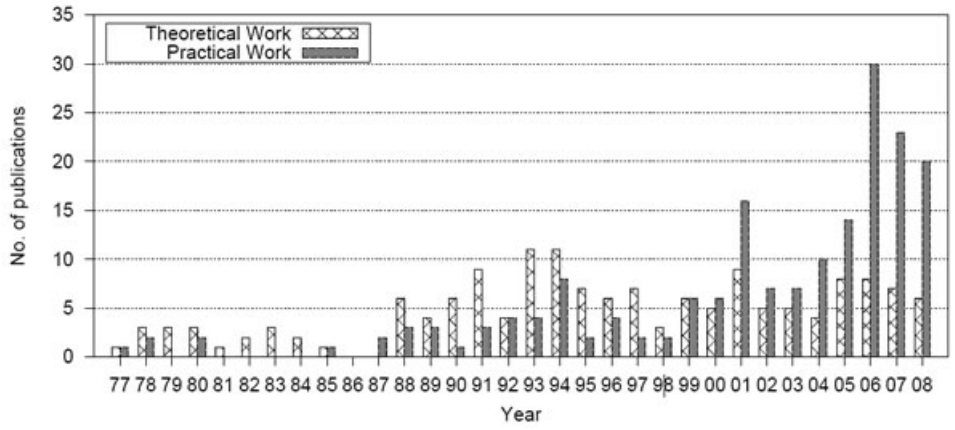

Figura 3: Trabajos de Mutation Testing publicados por año [JH09]

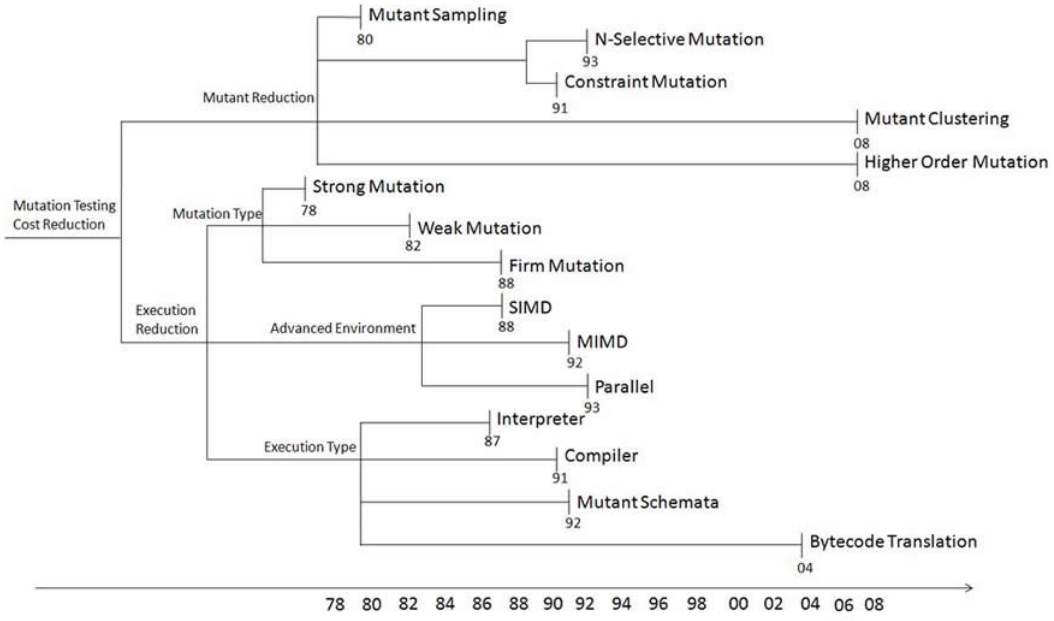

Figura 4: Clasificación de optimizaciones en orden cronológico [JH09]

Como se puede observar en la figura 4 a lo largo de los años se ha intentado optimizar los tiempos de ejecución de la técnica. La optimizaciones realizadas pueden ser resumidas en 4 principales categorías: *Reducción de operadores*, *Reducción de Ejecución a través del Tipo de Mutante*, *Reducción de Ejecución a través del Ambiente* y *Reducción de Ejecución a través del Tipo de Ejecución*. Veamos un resumen de cada uno de las categorías.

Reducción de operadores

Para reducir los tiempos de ejecución se puede reducir la cantidad de operadores de mutación a utilizar. Muchos de los operadores utilizados suelen generar mutantes equivalentes que demoran la ejecución de la técnica. Reducir la cantidad de operadores permitiría reducir el tiempo de generación de mutantes y el tiempo de ejecución ya que no se estarían generando mutantes equivalentes.

Una de las ideas que fueron desarrolladas bajo esta premisa fue la de determinar y utilizar un conjunto de operadores selectos (*Selective Mutation*), que sean distintos entre ellos y que generen la mayor cantidad mutantes posibles tratando de minimizar la cantidad de mutantes equivalentes. En [OLR+94] se hace un estudio comparando el uso de operadores selectos en lugar de utilizar todos los operadores y mediante este estudio se determinó que de los 22 operadores utilizados por la herramienta Mothra (ver sección herramientas), 5 resultan ser los claves y cubren casi la misma cantidad de mutantes.

A partir de ese trabajo uno podría pensar que lo ideal sería poder determinar qué mutantes son equivalentes con exactitud y descartarlos para minimizar los tiempos. Para ello, se investigaron heurísticas que utilizan técnicas de optimización de compiladores con el objetivo de atacar este problema ([OC94]), teniendo en cuenta que determinar mutantes equivalentes es un problema no decidible ([BA82]).

Reducción de Ejecución a través del Tipo de Mutante

Otra técnica para reducir los tiempos es mejorar la ejecución de acuerdo al tipo de mutante generado. Para ello, en algunos trabajos se propuso un modelo alternativo llamado *Weak Mutation* en lugar del clásico, al que se llamó *Strong Mutation* [OU00]. En Weak Mutation, en lugar de verificar si los tests pasan cuando se genera un mutante, se comparan estados intermedios del mutante con el programa original inmediatamente después de la ejecución de la parte del programa que tiene la mutación. Aunque pueda resultar más eficiente, con este modelo es posible verificar que la condición necesaria para que el mutante esté vivo se cumpla, pero no la condición suficiente. Es decir, no se garantiza que si el estado es diferente, el mutante vaya a quedar muerto. Weak Mutation entonces no mide precisamente la calidad de los tests y sus aserciones.

Veamos esto con un ejemplo. Supongamos el siguiente método que al evaluarse indica si un número es impar:

```
Number>>isOdd
    remainder := self - (2 * (self/2) truncated)
    ^remainder ~= 0.
```

y supongamos también que tenemos el siguiente caso de test:

```
NumberTest>>testOdd
    self assert: 3 isOdd.
```

Consideremos el mutante resultante de cambiar el - por + en la primer línea:

```
Number>>isOdd
    remainder := self + (2 * (self/2) truncated)
    ^remainder ~= 0.
```

Con Weak Mutation testing, corriendo el test se podría *matar* al mutante al determinar que, luego de la asignación, *reminder* referencia a un 5 en lugar de un 1 como en el método original. Es decir, se mata al mutante porque hay una diferencia en ese punto entre el estado del programa original y el estado del mutante.

Por otro lado, está claro que con Strong Mutation no se estaría matando al mutante porque la aserción sigue valiendo y por lo tanto el test pasa. Con Strong Mutation al dejar vivo al mutante se pone en evidencia que falta por lo menos un caso de prueba, mientras que con Weak Mutation, esto pasaría inadvertido.

Reducción de Ejecución a través del Ambiente

También se experimentó con la concurrencia, para esto, se hicieron implementaciones para aprovechar arquitecturas SIMD o MIMD y hacer que el análisis se ejecute en forma paralela [OU00]. Como cada mutante es independiente de los demás, los costos de comunicación son bajos y la concurrencia es factible y muy aprovechable.

En [OPFK92] por ejemplo, utilizando HyperMothra, una implementación basada en Mothra para explotar el paralelismo, se muestra que para modelos de software lo suficientemente grandes se pueden obtener hasta mejoras de tiempo lineales en función de la cantidad de procesadores. Esta mejora depende también de la estrategia con que se distribuya la ejecución entre los diferentes procesadores.

Reducción de Ejecución a través del Tipo de Ejecución

Con respecto al tiempo de compilación, una forma propuesta de minimizar el mismo consiste en utilizar otro modelo denominado *Mutant Schema Generation* [UOH93]. Este modelo codifica todos los mutantes en un solo *metaprograma* que es compilado una única vez. Luego, mediante una variable global, por ejemplo, puede determinarse qué mutante es el que tiene que estar *activo*. Éste método puede obtener ventajas significativas. Sin embargo, en casos en que se quieren hacer análisis con pocos mutantes se sigue pagando un costo considerable en la compilación, ya que el tiempo de compilación del metaprograma suele ser superior al de algunos pocos métodos de un lenguaje como Smalltalk.

Otra forma de ahorrar este tiempo consiste en evitar hacer la compilación completa y modificar el bytecode para aplicar las mutaciones, logrando una compilación optimizada a bajo nivel. Esta idea fue implementada en herramientas como MuJava ([sMOK05a]). Sin embargo, se aplica a lenguajes que compilan a código intermedio como Java, y tiene la desventaja de ser dificil de agregar nuevos operadores y de tener un acoplamiento con la codificación en bytecodes.

En [OU00] se hace una clasificación de todas estas optimizaciones en las siguientes categorías: *do fewer*, *do smarter* y *do faster*. En *do fewer* se consideran todas las formas de hacer el análisis con la menor cantidad de mutantes posibles sin incurrir en perdida de información como por ejemplo en el caso de utilizar operadores selectos. En los casos de *do smarter* se intenta distribuir el costo computacional en varias computadoras, factorizar el costo en varias ejecuciones conservando información entre cada corrida o evitar tener que correr el ejecuciones completas. En esta categoría entrarían los trabajos de concurrencia y Weak Mutation. Con *do faster* se intenta generar y evaluar cada mutante tan rápido como sea posible como, por ejemplo, en el caso de Mutant Schema Generation.

3.1. Mutation Testing en Programación Orientada a Objetos

Con respecto a la investigación de la técnica en la programación orientada a objetos, pueden nombrarse trabajos como [sMOK05a], donde se presenta y analiza una lista de operadores sintetizada de trabajos anteriores. Dado que los operadores fueron creados para una herramienta de Java, muchos de ellos tienen que ver con chequeos estáticos que realiza el compilador, como por ejemplo, cambiar el tipo de una variable o modificar sus niveles de acceso (private, protected o public). También se presentan varios otros que en general podrían utilizarse en otros ambientes de objetos (algunos exclusivamente en ambientes con subclasificación). En la mayoría de estos casos, estos son operadores generales que

tienen un criterio de fuerza bruta, como por ejemplo, borrar una variable, un método, una asignación o cambiar el orden de los parámetros de un método (o un envío de mensaje).

Según lo analizado, no existen hasta el momento trabajos en donde se desarrollen operadores que apliquen mutaciones para cambiar el comportamiento en la utilización de objetos de uso común, como por ejemplo instancias de la jerarquía de colecciones. Además, los operadores implementados no suelen estar desarrollados con criterios que permitan determinar los casos de test que verifican sino que, como se mencionó anteriormente, simulan fallas aleatorias sin un objetivo concreto.

3.2. Herramientas de Mutation Testing

Diferentes herramientas fueron implementadas a lo largo de los años con el objetivo de aplicar la técnica de Mutation Testing. Inicialmente se desarrollaron PIMS (para programas en Fortran IV) y Mothra. Esta última se caracterizó por cierta modularidad en su diseño, lo que permitió mayor flexibilidad para la experimentación, permitiendo agregar y/o modificar las formas de procesamiento.

Otras herramientas implementadas son:

- MuJava: Implementada para Java, se desarrollaron versiones para utilizar *Weak Mutation*, *Mutant Schema Generation* y *Selective Mutation*. Utiliza además de los operadores clásicos procedurales, los de clases, nombrados con anterioridad. Como se mencionó anteriormente, también se desarrolló para MuJava una técnica de traducción a bytecode de los mutantes para evitar los tiempos de compilación.
- CREAM: Es una implementación para C#. Implementa cinco operadores de mutación orientados a objetos.
- JESTER: Fue la primera herramienta open source de Mutation Testing para Java. Esta solamente provee dos operadores de mutación, uno cambia 0 por 1 y el otro reemplaza predicados con TRUE y FALSE. Existe una Version para Pyton llamada Pester y otra para C# llamada Nester.
- HECKLE: Es una implementación open source para Ruby que se utiliza desde línea de comandos. Actualmente soporta mutación de booleanos, numeros, strings, simbolos, rangos, expresiones regulares y branches para clases enteras o metodos individuales. Después de correr un análisis provee un reporte básico resumiendo los resultados estadísticos.

Cabe mencionar que en el muy reciente trabajo [SDZ09] se menciona el desarrollo de una nueva herramienta llamada JAVALENCHE, en el mismo se propone la utilización de información de coverage para minimizar la cantidad de tests a correr por cada mutante. Esta idea fue desarrollada en esta tesis previamente a dicha publicación como una de las estrategias de optimización. En la sección 5.1 de este informe se presenta un análisis más detallado de la misma.

4. Objetivos

La integración de tests automatizados en el proceso de desarrollo de software se ha vuelto más común en los últimos años. El incremento en la complejidad del software, la necesidad continua de cambios y la búsqueda de calidad han resultado en una mayor inversión en el desarrollo de los mismos. Más aún si consideramos el incremento del uso de la técnica de Test-Driven Development, en donde la generación de tests es una parte esencial del proceso de desarrollo.

Sin embargo, no mucho se ha hecho por buscar técnicas y herramientas que permitan trabajar sobre la calidad de los tests, lo que en la práctica implicaría una mejor calidad en el modelo computacional desarrollado.

En este contexto, la técnica de Mutation Testing podría aportar valor al desarrollo, siempre y cuando la misma se pueda integrar naturalmente en el proceso de desarrollo de software.

Por todo esto, la motivación en el presente trabajo radica, por un lado, en poder contar con una herramienta que permita utilizar y experimitentar la técnica de Mutation Testing en ambientes de objetos y, por otro lado, se busca que estas herramientas estén integradas al ambiente de desarrollo, que sean dinámicas y permitan obtener una rápida retroalimentación al programador. Además, se busca que éstas se adapten a un desarrollo ágil acompañando el mismo y que no se utilicen únicamente en etapas separadas.

Cabe desatacar que actualmente no existen implementaciones de este tipo, que se adapten al proceso de desarrollo fácilmente. Generalmente se cuenta con facilidades para realizar análisis de manera *batch*, pero no con formas de realizar análisis rápidos y ágiles, de porciones pequeñas del programa y en forma eficiente. En parte, este problema se debe a que en lenguajes estáticos es casi inevitable tener un tiempo de compilación ineludible, pero además, el hecho de no contar con ciertas capacidades reflexivas impide realizar mejoras importantes en la generación de mutantes y en la integración de esta técnica al proceso de desarrollo diario.

Un ambiente de objetos dinámico y reflexivo entonces podría resultar ideal en este sentido, ya que permite modificar y extender las herramientas de programación fácilmente y, a través de modelos simples, abiertos y flexibles, obtener una implementación eficiente que permita llevar a la práctica esta técnica con mayor dinamismo.

Como se presentará a continuación, en este trabajo se aprovecharon estas ventajas de los ambientes de objetos dinámicos y reflexivos, para obtener herramientas que faciliten el desarrollo combinando técnicas de testing como son Mutation Testing y Code Coverage, pero sobre todo mejorando los tiempos de respuesta y la integración con el proceso de software.

En conclusión, los objetivos a alcanzar son:

- Implementar herramientas que permitan utilizar Mutation Testing en un ambiente dinámico de objetos.
- Integrar la técnica de Mutation Testing a las herramientas existentes y al proceso de desarrollo.
- Investigar operadores de mutación y técnicas que faciliten y optimicen la utilización de Mutation Testing en ambientes de objetos dinámicos.
- Que las herramientas implementadas permitan mejorar los tests y el desarrollo de sistemas a través de la retroalimentación que la técnica de Mutation Testing provee.

5. MuTalk: Mutation Testing utilizando las capacidades reflexivas de un ambiente Meta-circular

Como trabajo de investigación se buscó implementar y optimizar los tiempos de la técnica de mutation testing en Smalltalk. Para ello se realizaron optimizaciones de tiempo en la generación y evaluación de mutantes; se buscaron criterios para la creación de operadores de mutación que permita obtener operadores que brinden mejor información y en lo posible más eficiente; y se crearon herramientas que facilitan la utilización de la técnica en Smalltalk. A continuación se explicará la forma que se buscó para atacar el primero punto, luego se mostrarán los distintos operadores junto con una heurística para la elección de los mismos, que atacaría el segundo punto y finalmente se detallarán las herramientas desarrolladas y se describirán las clases principales del modelo implementado.

5.1. Optimizaciones

Se puede decir que en principio Mutation Testing es una técnica de fuerza bruta. Si bien existen trabajos realizados para intentar optimizarla, en muchos casos las mismas se centran en lenguajes estáticos, en donde el tiempo de compilación es importante (por ejemplo los Mutant Schema Generation [UOH93]). En el caso de un ambiente dinámico estas optimizaciones aplicadas para mejorar los tiempos de compilación no tienen mucho sentido, por lo que se buscaron optimizaciones sobre otras etapas dentro del proceso de mutation testing (por ejemplo la etapa de generación o de evaluación de mutantes).

Como paso previo a buscar estas optimizaciones, sería bueno ver cuánto tiempo ocupa cada uno de estos pasos de un análisis de mutation testing (generación y evaluación de mutantes).

Antes de ver esto es importante aclarar que, tanto para este como para los demás análisis de esta sección, se experimentó aplicando la herramienta implementada (ver sección 5.4) sobre los modelos presentados en la siguiente tabla, repitiendo el experimento tres veces y promediando en el caso de gráficos de tiempo.

Modelos analizados	
Aconcagua	Modelo de medidas open source
Network	Toolkit de protocolos de red implementados en Squeak/Pharo Smalltalk
VB-Regex	Implementación de expresiones regulares
Magritte	Framework de Meta-Descripciones
Pier	CMS implementado en Smalltalk basado en el framework seaside

Las pruebas se hicieron en una computadora AMD Sempron 3000+ con 1GB de memoria RAM y con la versión 1.0 de Pharo Smalltalk. En todos los casos se utilizaron 55 operadores de mutación.

Cabe mencionar que se hizo el análisis con paquetes y subpaquetes open source disponibles en Squeak Source. Algunos de ellos son subpaquetes de otro paquete más general: Por ejemplo, *Aconcagua-ArithmeticModel* es un subpaquetes incluido en *Aconcagua*.

Lo que se pudo notar en un análisis sin aplicar ninguna optimización es que en la mayoría de los casos el tiempo de evaluación de cada mutante es superior con respecto a la generación de los mismos. Con la evaluación nos referimos a la etapa del análisis en que se intentan matar los mutantes y con la etapa de generación nos referimos a cuando se crean los mismos.

En la figura 5 se muestra la relación del tiempo ocupado por cada uno de los pasos de la técnica para algunos de los paquetes analizados. Allí se puede ver como el tiempo evaluación es generalmente superior respecto a los tiempos de generación. Esto significa que, en principio, seria conveniente lograr optimizar el tiempo de evaluación. Es importante tener en cuenta que si bien la forma en que se evalúa cada mutante es importante para reducir este tiempo, también hay que tener en cuenta la cantidad de mutantes que se generan y, como consecuencia, se evalúan. Es decir, el tiempo de evaluación es también relativo a la cantidad de mutantes generados.

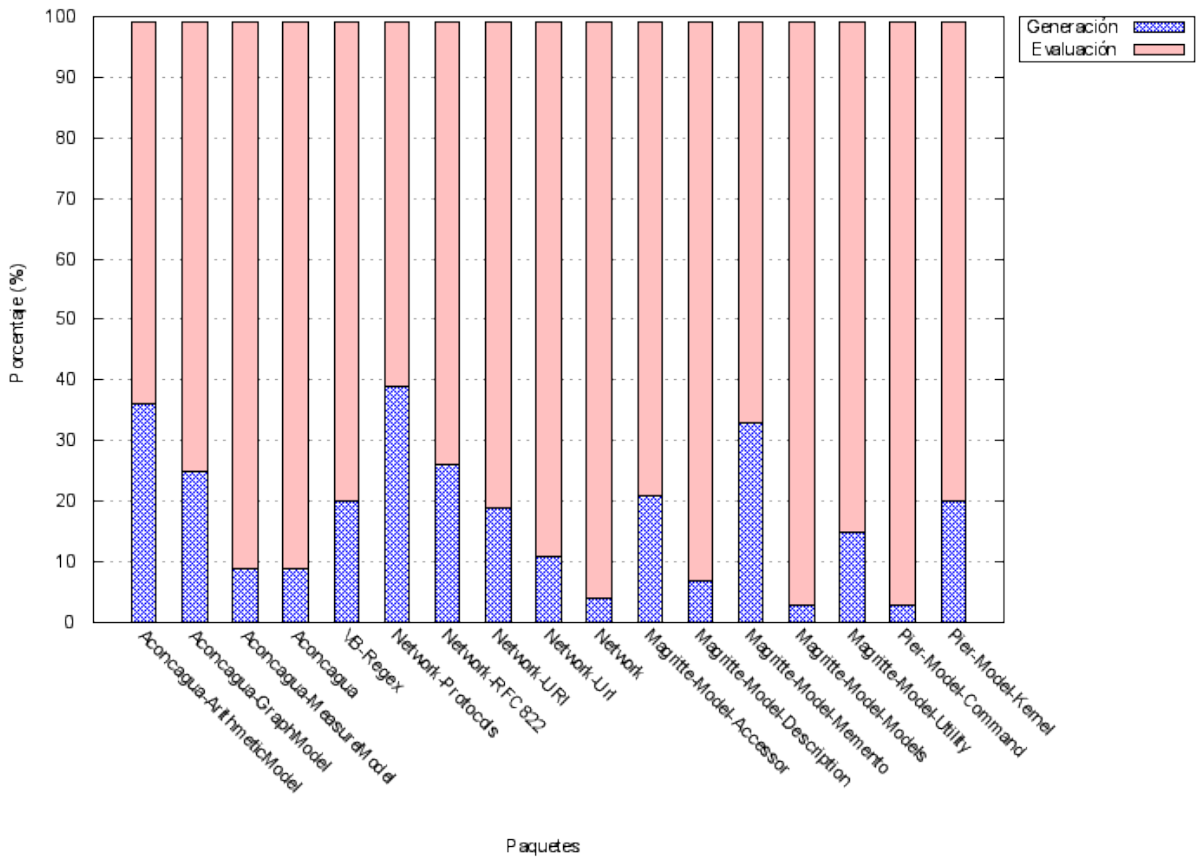

Figura 5: Comparación de tiempos entre generación y evaluacion de mutantes

5.1.1. Estrategias utilizando coverage

Con la idea de atacar estos puntos, se pensó en la posibilidad de utilizar información obtenida a partir de un análisis de coverage y, con esta, reducir tanto la cantidad de mutantes cómo la cantidad de tests que se corren para evaluar cada mutante.

Para entender cómo se utilizaría esta información, podemos ver el siguiente ejemplo.

Supongamos que tenemos tres casos de test que testean un modelo determinado y tomemos cuatro métodos de este modelo. En la figura 6 puede verse el cubrimiento de estos casos de test con respecto a estos métodos. Esto es, podemos ver que el test *testCase1* cubre los metodos *method1* y *method2*, es decir, cuando se corre *testCase1*, se evalúan los métodos *method1* y *method2*. En el caso de *testCase2*, este cubre los métodos *method1* y *method3* y por último, *testCase3* cubre solo *method3*. Por otro lado, podemos ver que *method4* no es cubierto por ningún test.

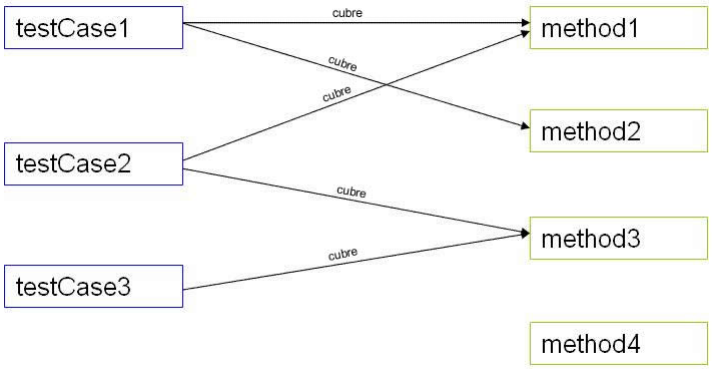

Figura 6: Ejemplo de cubrimiento

Imaginemos ahora que se decide realizar un análisis de Mutation Testing sobre estos tests y estos métodos y tomemos, por ejemplo, un mutante que se genera modificando *method1*. Si se quisiera evaluar este mutante, es decir, intentar matarlo, ¿qué tests habría que correr? Una posibilidad sería correr todos los tests, pero esto no tendría sentido y sería más eficiente correr solamente *testCase1* y *testCase2*. Es decir, podría correrse también *testCase3*, pero este no podría matar nunca al mutante porque el método mutado no es evaluado como consecuencia de correr este test. Entonces, el tiempo de ejecución del *testCase3* podría ahorrarse. Lo mismo se podría pensar al aplicarse una mutación sobre *method2* y *method3*. Si bien este tiempo podría llegar a ser despreciable, cuando se tiene una suite de test compleja la suma de estos tiempos podría resultar en un ahorro significativo.

Por otro lado, si tomamos el caso de *method4*, ¿tiene sentido mutarlo? Si se realiza un mutante modificando este método, este nunca podría ser matado ya que no se cuenta con un test que lo cubra. Por lo tanto, toda mutación del mismo generaría un mutante que no es posible matar a menos que se agregue el correspondiente caso de test que provoque una evaluación del mismo. En este caso, entonces, lo ideal sería detectar previamente esto por un análisis de coverage, que suele tomar menos tiempo, agregar casos que cubran el método en cuestión y entonces si tendría sentido mutarlo.

De este razonamiento se puede decir entonces que, tomando esta información de coverage, se podría elegir cómo correr un análisis de mutation testing: si corriendo todos los tests por cada mutante o solo los que lo cubren; y si mutar todos los métodos o solo los cubiertos por tests.

Para poder elegir estas alternativas al momento de correr un análisis se implementaron dos tipos de estrategias que pueden ser combinadas: *estrategia de evaluación (correr todos los tests o solo los que cubre)* y *estrategia de generación (mutar todos los métodos o solo los cubiertos)*.

La combinación resultante sería entonces la siguiente:

1. **Mutar todos los métodos, correr todos los tests para cada mutante (All-All)**
2. **Mutar todos los métodos, correr tests que cubren el método mutado (All-Covering)**
3. **Mutar métodos cubiertos, correr todos los tests para cada mutante (Covered-All)**
4. **Mutar métodos cubiertos, correr tests que cubren el método mutado (Covered-Covering)**

De ahora en más nos referiremos a las diferentes combinaciones de estrategias como: All-All, All-Covering, Covered-All y Covered-Covering respectivamente; en donde la parte anterior al guión se refiere a la estrategia con la que se seleccionan los métodos a mutar (All o Covered) y la parte posterior al guión se refiere a la estrategia de selección de tests a evaluar (All o Covering). En el primer caso, *All* significa que se muten todos los métodos y *Covered* que sólo se muten los método cubiertos por los tests (methods covered by tests); en el segundo caso, *All* significa que se corran todos los tests y *Covering* que sólo se corran los tests que cubran al método mutado (tests covering mutated method).

Siguiendo con el ejemplo, en la tabla 1 se muestra, para cada una de las combinaciones de estrategias, los métodos que se mutarían y los casos de test que se correrían para cada mutante respectivo.

Estrategias	Métodos Mutados	Test Corridos por Mutante
1. All-All	method1	testCase1, testCase2, testCase3
	method2	testCase1, testCase2, testCase3
	method3	testCase1, testCase2, testCase3
	method4	testCase1, testCase2, testCase3
2. All-Covering	method1	testCase1, testCase2
	method2	testCase1
	method3	testCase2, testCase3
	method4	
3. Covered-All	method1	testCase1, testCase2, testCase3
	method2	testCase1, testCase2, testCase3
	method3	testCase1, testCase2, testCase3
4. Covered-Covering	method1	testCase1, testCase2
	method2	testCase1
	method3	testCase2, testCase3

Cuadro 1: Métodos mutados y tests corridos para el ejemplo de la figura 6

Comparación de estrategias de evaluación de mutantes

Empezaremos por analizar las diferencias considerando la optimización de correr solo los tests que cubren un mutante. Para esto observaremos la figura 7 y la tabla 2 en donde puede verse la comparación de tiempos entre las estrategias All-All y All-Covering.

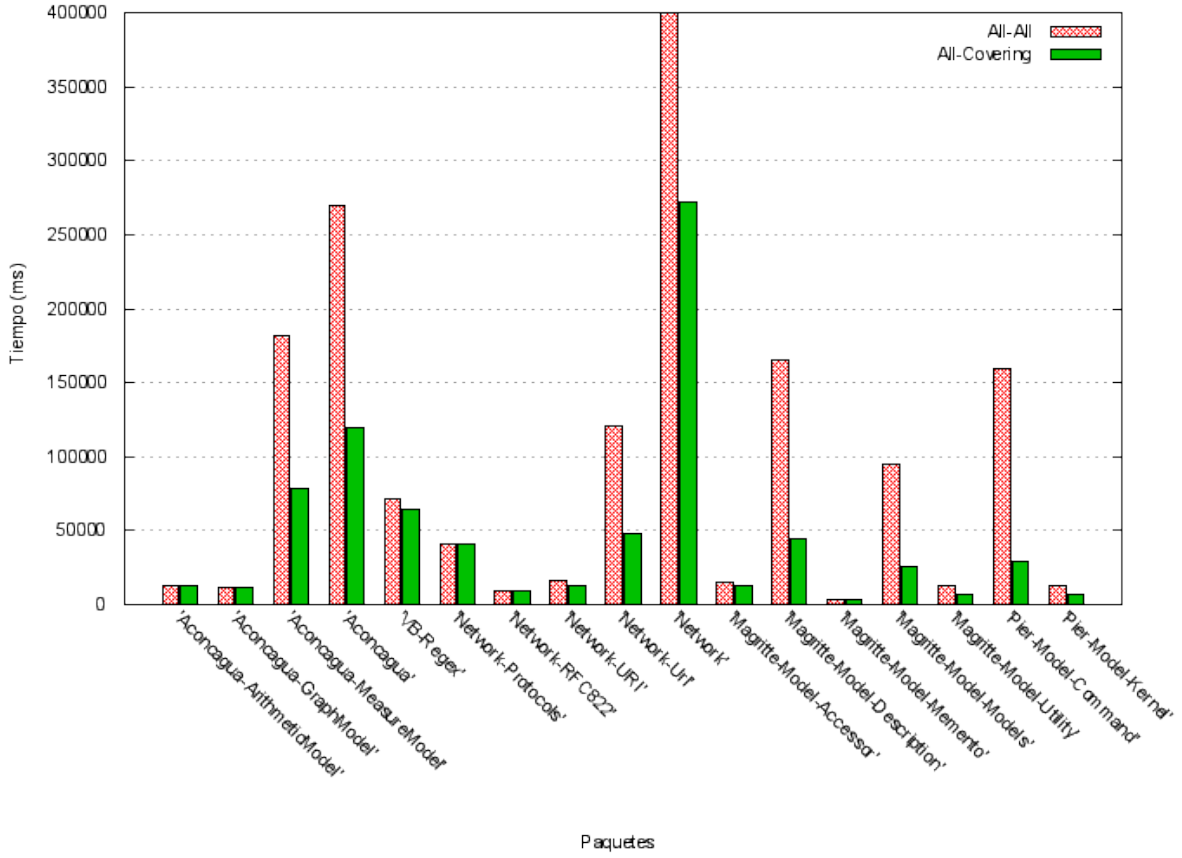

Figura 7: All-All vs All-Covering con escala recortada

Dado que en el caso del paquete *Network* la diferencia es excesiva (tiempo de All-All aproximadamente 6.2 veces el tiempo de All-Covering), se decidió recortar la escala en la figura 7 para mostrar con

Paquete	All-All				All-Covering				Reducción de tiempos		
	Coverage	Generación	Evaluación	Total	Coverage	Generación	Evaluación	Total	Evaluación (%)	Total (%)	total(ms)
Aconcagua-ArithmeticModel	144	4885	8426	13455	155	4956	6979	12090	17,17	10,14	1365
Aconcagua-GraphModel	80	2942	8422	11444	113	3212	8747	12072	-3,86	-5,48	-628
Aconcagua-MeasureModel	1205	18091	164258	183555	1306	19634	55464	76404	66,23	58,38	107151
Aconcagua	1497	26667	240721	268885	1637	30441	86084	118163	64,24	56,05	150722
VB-Regex	92	14489	56437	71018	98	15542	49167	64807	12,88	8,75	6211
Network-Protocols	16	16130	24282	40428	17	16181	25029	41227	-3,07	-1,98	-799
Network-RFC822	9	2413	6524	8946	9	2796	6521	9326	0,04	-4,25	-380
Network-URI	79	3373	13583	17035	84	3853	9143	13080	32,69	23,22	3956
Network-Url	100	14514	107840	122454	107	15162	37386	52654	65,33	57	69800
Network	228	85443	1660591	1746262	266	92983	189209	282458	88,61	83,82	1463803
Magritte-Model-Accessor	47	2998	11220	14265	48	2985	8590	11623	23,44	18,52	2642
Magritte-Model-Description	608	12483	151193	164285	642	13099	29251	42992	80,65	73,83	121293
Magritte-Model-Memento	34	1111	2171	3316	35	1107	1950	3092	10,15	6,75	224
Magritte-Model-Models	355	3295	97381	101030	375	3598	20857	24830	78,58	75,42	76201
Magritte-Model-Utility	92	1976	10743	12812	96	2279	6200	8575	42,29	33,07	4236
Pier-Model-Command	595	4929	153988	159513	674	5190	23629	29492	84,66	81,51	130021
Pier-Model-Kernel	84	2586	9756	12426	86	2757	4670	7512	52,13	39,54	4913
PROMEDIO									41,89	36,13	

Cuadro 2: Tiempos All-All vs All-Covering

mayor claridad los tiempos en los demás paquetes.

Para analizar en particular esta diferencia tan notable en el paquete *Network*, observemos la figura 8 en donde pueden verse los diversos porcentajes de coverage y la tabla 3 que muestra la cantidad de mutantes y tests por paquete. En este caso se observa que si comparamos con los demás, la cobertura de este paquete es muy inferior. Al ser entonces este un modelo con tan poco cubrimiento y, además, con numerosas clases y métodos (lo que implica muchos mutantes), sucede que en la primer estrategia (All-All) se corren todos los tests para una cantidad de mutantes mucho mayor, además del *overhead* adicional que esto implica, teniendo como resultado un tiempo mucho mayor para el análisis.

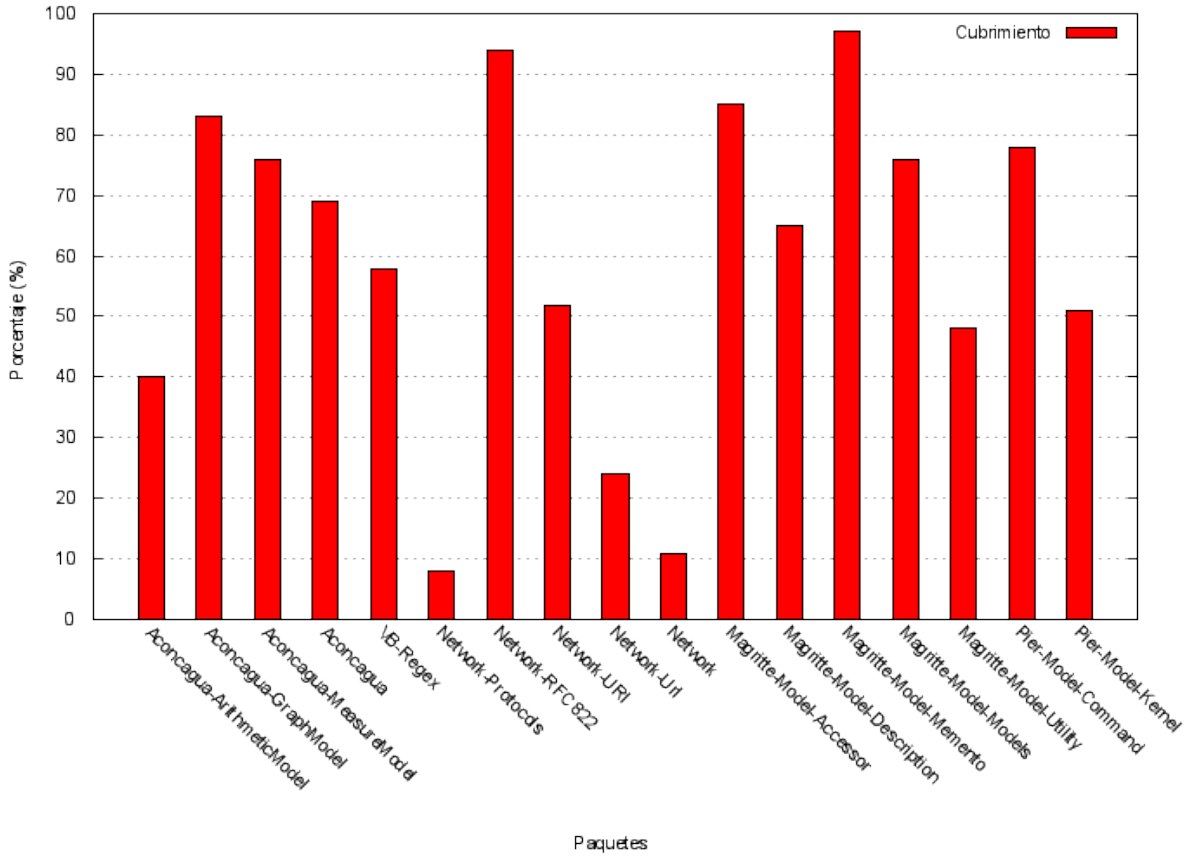

Figura 8: Cubrimiento de paquetes analizados

Paquete	Mutantes	Métodos de test
Aconcagua-ArithmeticModel	217	40
Aconcagua-GraphModel	210	42
Aconcagua-MeasureModel	1020	585
Aconcagua	1493	667
VB-Regex	1018	15
Network-Protocols	649	1
Network-RFC822	184	1
Network-URI	212	47
Network-Url	879	37
Network	4966	94
Magritte-Model-Accessor	171	163
Magritte-Model-Description	581	1415
Magritte-Model-Memento	57	73
Magritte-Model-Models	184	39
Magritte-Model-Utility	98	36
Pier-Model-Command	278	329
Pier-Model-Kernel	109	32

Cuadro 3: Cantidad de mutantes y tests por paquete

Por otra parte, como se puede notar en general, la diferencia de tiempos entre correr todos los tests para cada mutante o correr solo los que cubren se hace más notable en paquetes más grandes, con más métodos (lo que implica más mutantes) y más tests.

Es razonable que esto sea así, ya que en paquetes grandes tiende a haber tests más diversos para testear partes del modelo independientes, entonces es más probable que al generarse un mutante este tenga que ser evaluado con un subconjunto de tests bastante menor del total.

Por ejemplo, en *Aconcagua-MeasureModel* que es un paquete más grande y con más mutantes que *Aconcagua-ArithmeticModel* o *Aconcagua-GraphModel*, la reducción de tiempo total fue de 56 % versus 10.1 % y -5.5 % respectivamente. Es decir que incluso en el último de estos tres casos se aumentó el tiempo. Este aumento se explica, además de eventuales errores en la medición al ser los tiempos no muy diferentes, en el hecho de que con ambas estrategias el número de tests por mutante sea muy parecido, pero que en el caso de *All-Covering* se agregue el overhead que implica para cada mutante la búsqueda de los tests que lo cubren. Sin embargo, si pensamos en términos absolutos en el tiempo adicional (628 ms) que se suma en este paquete relativamente chico, podríamos decir que es una diferencia casi imperceptible.

Por lo tanto, vemos que generalmente, comparando la estrategias *All-All* con *All-Covering*, en casos de paquetes grandes se puede llegar a reducciones del tiempo total de hasta casi un 84 % y en paquetes chicos la diferencia es menor e incluso puede haber un ligero aumento de tiempo que suele ser casi imperceptible.

Una ventaja interesante que brinda el hecho de correr solo los tests que cubren al método mutado es que puede utilizarse para dar mayor agilidad al programador. Por ejemplo, supongamos que se está trabajando sólo con una clase de *Aconcagua* y se quiere aplicar Mutation Testing sobre ella. Una alternativa es escoger cuidadosamente los tests que deberían correrse para analizar esta clase (para que el análisis no tome un tiempo excedido), pero otra alternativa sería tomar todos los tests del modelo y que la herramienta de mutation testing haciendo un análisis previo de *coverage* decida sola que tests correr, utilizando la estrategia correspondiente. Esta facilidad fue implementada entre las herramientas que se proveen y permite agilizar la utilización de la técnica.

Algo que no se mencionó, es que en *All-All* se está sumando el tiempo de coverage, pero que en realidad este análisis no sería necesario. Sin embargo, si bien en este caso no se necesita conocer el cubrimiento, en esa misma etapa, por un lado, se chequea la precondición de que los tests corran sin fallar, y por otro, se calcula el tiempo en correr los tests, que como veremos más adelante, se utiliza para determinar cuando terminar un mutante por la posible generación de un ciclo infinito en el programa. Es decir, para estos dos propósitos es necesario correr los tests una vez, y es por esto que no se podría

suprimir este tiempo.

Además, es importante mencionar que como el tiempo que toma el análisis de coverage es equivalente al de correr los tests una sola vez, por lo general es despreciable en términos relativos si se lo compara con el tiempo que lleva el total del análisis (ver tabla 2).

Comparación de estrategias de generación de mutantes

Analicemos ahora que pasaría con los tiempos si utilizamos la estrategia de generar mutantes sólo sobre los métodos cubiertos (All-Covering y Covered-Covering). Para esto podemos ver el gráfico de la figura 9 y la tabla 4.

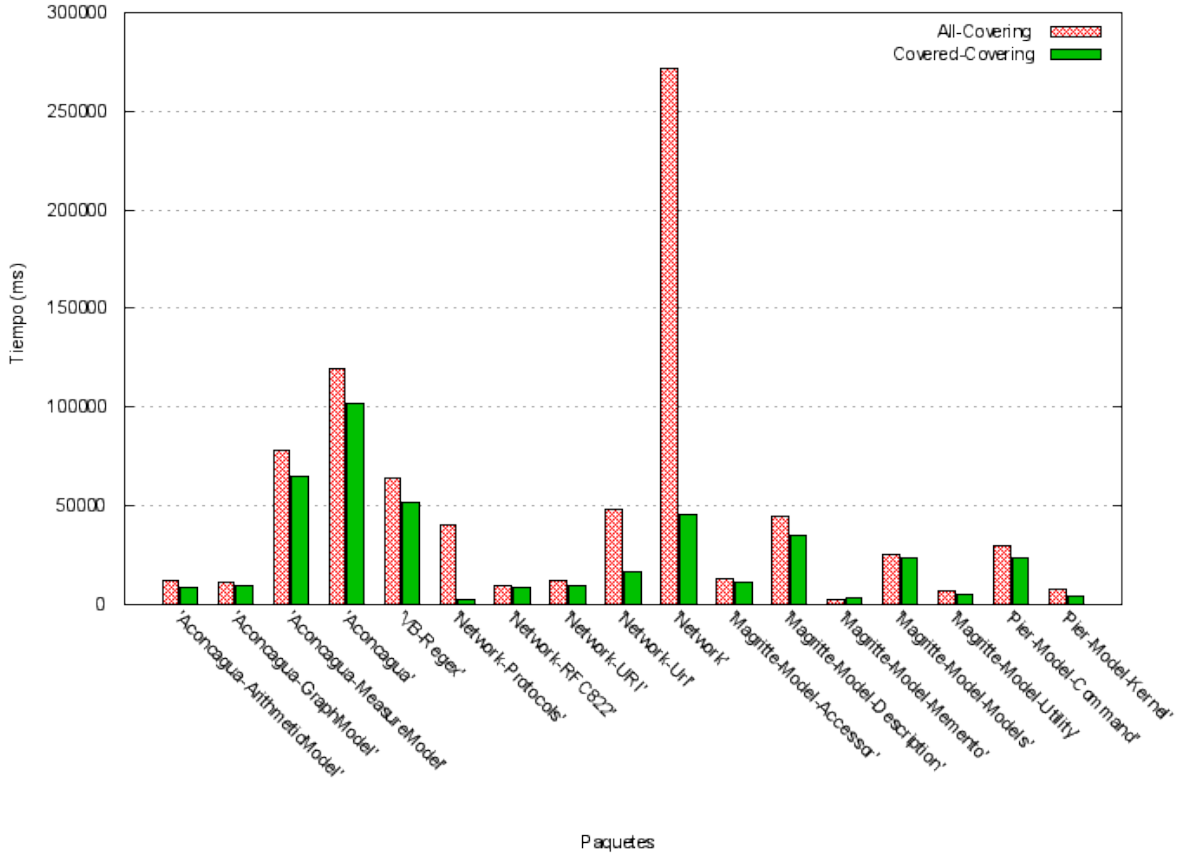

Figura 9: All-Covering vs Covered-Covering

Podemos observar nuevamente que hay una diferencia muy grande en el paquete *Network*. En este caso la tan distante cantidad de mutantes evaluados (4966 frente a 663) es la que determina el tiempo adicional, pero en esta ocasión este tiempo es ocupado en la generación y mayormente en el *overhead* que toma evaluar los mutantes adicionales, porque **para estos casos de mutantes de código no cubierto, no hay que correr ningún test.**

Esto explica por qué en los demás paquetes, si bien hay una reducción en los tiempos que en promedio es aproximadamente un 15 %, esta mejora no es muy pronunciada. Es decir, el hecho de que el número de mutantes a evaluar no sea tan diferente en ambos casos implica que el tiempo adicional es el de la generación y el del *overhead* de los mutantes de mutaciones no cubiertas, que en proporción no son tantos.

Para entender mejor esto volvamos al ejemplo de la figura 6 y miremos la tabla 1. En el gráfico puede verse que *method4* no es cubierto por ningún test, por lo tanto la diferencia es que mientras que con *All-Covering* sí se generan mutantes sobre este método, con *Covered-Covering* no. En consecuencia, podemos decir que en el primer caso existe el tiempo adicional que toma generar mutantes sobre

Paquete	All-Covering				Covered-Covering				Reducción de tiempos			
	Coverage	Generación	Evaluación	Total	Coverage	Generación	Evaluación	Total	Generación(%)	Evaluación(%)	Total (%)	total(ms)
Aconcagua-ArithmeticModel	155	4956	6979	12090	153	2592	6163	8908	47,69	11,69	26,32	3182
Aconcagua-GraphModel	113	3212	8747	12072	92	3048	7416	10556	5,1	15,22	12,56	1516
Aconcagua-MeasureModel	1306	19634	55464	76404	1375	16272	50057	67704	17,12	9,75	11,39	8700
Aconcagua	1637	30441	86084	118163	1664	22550	87617	111832	25,92	-1,78	5,36	6331
VB-Regex	98	15542	49167	64807	101	10124	42929	53154	34,86	12,69	17,98	11653
Network-Protocols	17	16181	25029	41227	17	874	1600	2491	94,6	93,61	93,96	38737
Network-RFC822	9	2796	6521	9326	10	2723	6936	9668	2,63	-6,35	-3,66	-342
Network-URI	84	3853	9143	13080	88	2510	7928	10525	34,86	13,29	19,53	2554
Network-Url	107	15162	37386	52654	112	3929	14964	19005	74,08	59,97	63,91	33650
Network	266	92983	189209	282458	263	10342	36622	47227	88,88	80,64	83,28	235232
Magritte-Model-Accessor	48	2985	8590	11623	84	2661	10485	13230	10,85	-22,06	-13,82	-1607
Magritte-Model-Description	642	13099	29251	42992	678	9193	26518	36389	29,82	9,34	15,36	6603
Magritte-Model-Memento	35	1107	1950	3092	35	1026	2063	3124	7,32	-5,76	-1,01	-31
Magritte-Model-Models	375	3598	20857	24830	357	2585	19571	22513	28,16	6,17	9,33	2317
Magritte-Model-Utility	96	2279	6200	8575	95	1054	3912	5061	53,75	36,9	40,98	3514
Pier-Model-Command	674	5190	23629	29492	690	3621	20404	24716	30,22	13,65	16,19	4776
Pier-Model-Kernel	86	2757	4670	7512	88	1273	3460	4820	53,83	25,91	35,84	2692
PROMEDIO									37,63	20,76	25,5	

Cuadro 4: Tiempos All-Covering vs Covered-Covering

method4, pero a la hora de evaluar uno de estos mutantes, se buscan los casos de test que cubren a este método, y como este conjunto es vacío, la evaluación se termina ahí. Es decir, el incremento de tiempo lo tendríamos en la generación de mutantes sobre el método no cubierto y en el overhead de buscar el conjunto (vacío) de casos de test para estos mutantes. En el caso de *Covered-Covering*, al no generarse mutantes sobre *method4*, no hay una etapa de evaluación para estos y en consecuencia no existe esta búsqueda.

Sin embargo, es bueno contar con ambas posibilidades. Por un lado, se podría querer contar con información más completa, queriendo conocer todos los mutantes que no pueden ser matados, independientemente de su cubrimiento. Más aún si vemos que en la mayoría de los casos la espera adicional no es tan grande. Por otro lado, se podría querer contar con información más acotada y precisa, evitando tener que lidiar con una gran cantidad de mutantes que podrían ignorarse a conciencia. Un ejemplo de esto sería el caso de código autogenerados por herramientas, como las interfaces gráficas donde no tiene mucho sentido generar mutaciones.

Además, la estrategia de mutar solo los métodos cubiertos es utilizada por una opción que se agregó al browser de desarrollo estándar de Smalltalk y que permite, al navegar métodos de tests, seleccionar uno o más de estos métodos y correr mutation testing utilizándolos sin necesidad de elegir que partes del modelo mutar. Es decir, al seleccionar únicamente los tests que se desean utilizar y teniendo en cuenta el modelo testeado, la estrategia obtiene automáticamente los métodos que son cubiertos por estos tests seleccionados para aplicarle mutaciones. Esta opción resulta de utilidad para analizar de una manera más ágil los tests mientras se implementan. En la sección de herramientas de este informe se verá esta opción con más detalle.

Resultados

Por último, podemos observar la figura 10 y la tabla 5 en donde se puede ver la reducción total de tiempo si se compara hacer el análisis sin optimizaciones (All-All) con hacer el análisis con ambas optimizaciones (Covered-Covering). Por un lado, se puede ver una reducción de un 97.3 % en el caso especial de Network, es decir una velocidad de respuesta aproximadamente 37 veces mayor. Si consideramos los demás casos, en promedio hay una mejora del 50 % lo que significa el doble en velocidad. Podemos decir entonces, que esta mejora de tiempos dependerá de muchos factores, entre ellos: la cantidad de métodos del modelo, el nivel de cubrimiento, la cantidad de tests y el tiempo que llevan estos.

En la figura 11 puede verse el tiempo resultante ocupado por las etapas del análisis en la estrategia *Covering-Covered*. En primer lugar se puede notar que el tiempo tomado para evaluar el coverage es relativamente despreciable frente al resto. Pero lo que puede notarse además, es que el tiempo de evaluación sigue siendo mayor al de generación, lo que implicaría, a priori, que en caso de seguir buscándose optimizaciones, deberían buscarse por el lado de la evaluación de los mutantes.

Paquete	All-All				Covered-Covering				Reducción de tiempos		
	Coverage	Generación	Evaluación	Total	Coverage	Generación	Evaluación	Total	Evaluación (%)	Total (%)	total(ms)
Aconcagua-ArithmeticModel	144	4885	8426	13455	153	2592	6163	8908	26,85	33,79	4547
Aconcagua-GraphModel	80	2942	8422	11444	92	3048	7416	10556	11,95	7,76	888
Aconcagua-MeasureModel	1205	18091	164258	183555	1375	16272	50057	67704	69,53	63,12	115851
Aconcagua	1497	26667	240721	268885	1664	22550	87617	111832	63,6	58,41	157053
VB-Regex	92	14489	56437	71018	101	10124	42929	53154	23,93	25,15	17864
Network-Protocols	16	16130	24282	40428	17	874	1600	2491	93,41	93,84	37937
Network-RFC822	9	2413	6524	8946	10	2723	6936	9668	-6,31	-8,07	-722
Network-URI	79	3373	13583	17035	88	2510	7928	10525	41,64	38,21	6510
Network-Url	100	14514	107840	122454	112	3929	14964	19005	86,12	84,48	103449
Network	228	85443	1660591	1746262	263	10342	36622	47227	97,79	97,3	1699035
Magritte-Model-Accessor	47	2998	11220	14265	84	2661	10485	13230	6,55	7,26	1035
Magritte-Model-Description	608	12483	151193	164285	678	9193	26518	36389	82,46	77,85	127896
Magritte-Model-Memento	34	1111	2171	3316	35	1026	2063	3124	4,98	5,8	192
Magritte-Model-Models	355	3295	97381	101030	357	2585	19571	22513	79,9	77,72	78518
Magritte-Model-Utility	92	1976	10743	12812	95	1054	3912	5061	63,58	60,49	7750
Pier-Model-Command	595	4929	153988	159513	690	3621	20404	24716	86,75	84,51	134797
Pier-Model-Kernel	84	2586	9756	12426	88	1273	3460	4820	64,54	61,21	7606
PROMEDIO									52,78	51,11	

Cuadro 5: Tiempos All-All vs Covered-Covering

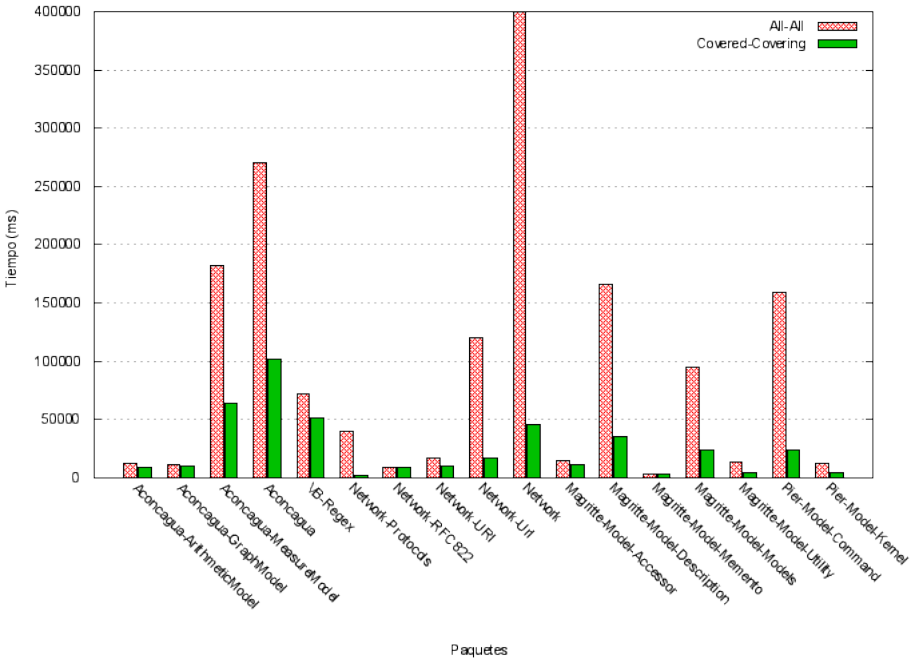

Figura 10: All-All vs Covered-Covering

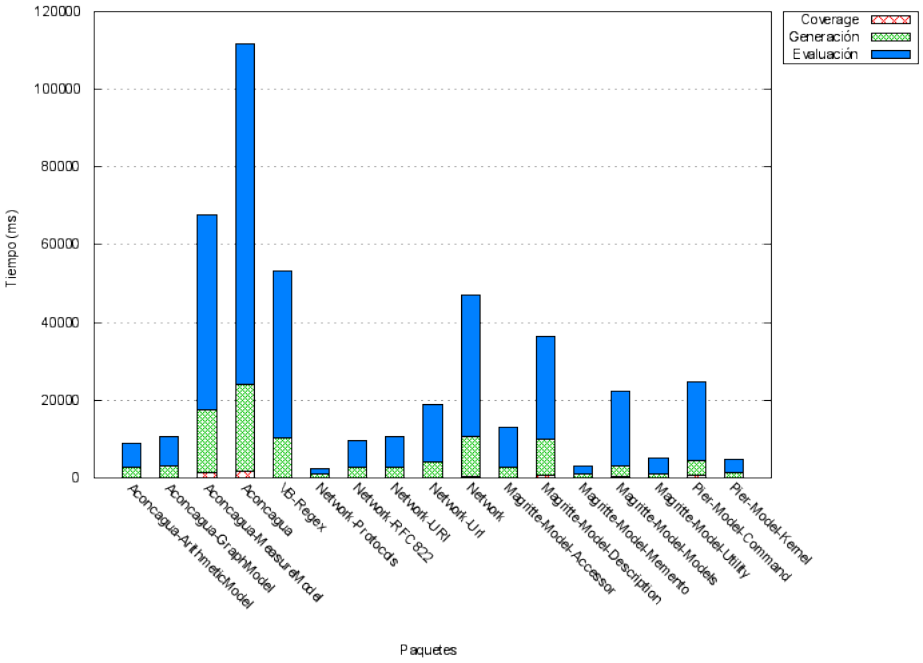

Figura 11: Covered-Covering detallado

Para terminar y luego de analizar los resultados, en el cuadro 6 se sintetiza y se completa una comparación de las diferentes combinaciones de estrategias.

Generación \Evaluación	All	Covering
All	▪ Es la combinación menos eficiente en la mayoría de los casos. ▪ Puede resultar más eficiente en paquetes chicos, con pocos métodos y tests, porque no tiene el overhead de utilizar *coverage*. ▪ Cuando es más eficiente, la diferencia suele ser imperceptible en términos absolutos.	▪ En los modelos no muy chicos es más eficiente correr los tests que cubren el mutante en lugar de correr todos. ▪ En casos de paquetes grandes se puede llegar a reducciones del tiempo total de hasta casi un 84 % ▪ Los resultados del análisis son los mismos que con All-All, salvo que los tests sean erráticos y tengan cierta dependencia entre sí, lo cual es algo no deseado para un conjunto de tests [Mes07]. ▪ Puede utilizarse para dar mayor agilidad al programador, permitiendo hacer Mutation Testing sobre una porción del modelo sin tener que seleccionar los tests que cubren esa porción ya que esto se calcula automáticamente. ▪ En el caso de mutantes de métodos no cubiertos, no corre ningún test para evaluarlos.
Covered	▪ Toma menos tiempo que *All-All* porque hay que generar y evaluar menos mutantes (solo los cubiertos). ▪ El resultado no necesariamente es el mismo que en *All-All* o en *All-Covered* ya que no existen mutantes no cubiertos, mientras que en los otros casos sí. Por ejemplo, en el caso de la figura 1 no existirían mutantes de *method4*, mientras que con la otra estrategia de generación si (y quedarían vivos).	▪ En los paquetes no muy chicos es la más eficiente de las combinaciones. ▪ En los casos analizados hubo reducciones de tiempo de hasta un 97 % y en promedio aproximadamente 50 % con respecto a *All-All* ▪ En la mayoría de los casos analizados, la reducción de tiempo con respecto a *All-Covered* no es muy pronunciada (en promedio aproximadamente 15 %). Esto se debe a que en *All-Covered* no hay que correr tests para mutantes no cubiertos (la diferencia de tiempos está en la generación y en la búsqueda de tests para esos mutantes). ▪ Es la estrategia *default* en las herramientas implementadas.

Cuadro 6: Combinaciones de estrategias

5.1.2. Optimización de Timeout

Un problema adicional que hubo que resolver durante el análisis es el de determinar cuánto tiempo tiene que transcurrir para que la evaluación de un mutante sea terminada. Esto debe hacerse debido a que al aplicarse una mutación podría generarse un ciclo o una recursión infinita, y en este caso, la

evaluación de los tests podría no finalizar.

Podríamos ver un ejemplo de esto con el método factorial de enteros, que tiene la siguiente definición:

```
Integer>>factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

Suponiendo que tenemos un test como el siguiente:

```
Integer>>testFactorial
    self assert: 3 factorial = 6
```

Si tomamos el siguiente mutante, que resulta de cambiar el mensaje - por un +, obtendríamos:

```
Integer>>factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self + 1) factorial].
    self error: 'Not valid for negative integers'
```

Es claro que si se corriera el test con este mutante, se produciría una recursión infinita, porque nunca se llegaría al caso base de 0.

Para detectar este tipo de situaciones, la solución más simple y directa es utilizar un *timeout* fijo. Sin embargo, esta idea tiene el problema evidente de que no existe un tiempo óptimo que se adecúe a cualquier paquete y cualquier computadora. Es decir, correr los tests de un paquete toma distinto tiempo que los de otro, y además, un mismo conjunto de tests puede tomar diferentes tiempos en diferentes computadoras con distintas características de hardware.

Para resolver esta limitación, la solución propuesta consiste en tomar el tiempo de correr los tests y luego utilizar este tiempo cómo referencias. De esta forma, se termina la evaluación de un mutante cuando el tiempo supera que lleva la misma supera en cierta cantidad de veces al de la evaluación de los tests. O sea, la condición para decidir si se termina la evaluación sería: $tiempoDeEvaluacion > tiempoTests * factor$, en donde $factor$ es una constante fija que decidió fijarse en 3 luego de poner en la práctica esta solución y probarla con diferentes valores en los diferentes modelos utilizados para la experimentación.

Cabe mencionar que la medición del tiempo que toman los tests se hace en la primer etapa, cuando se corren los tests para cálculo de coverage, por lo que no implica un tiempo adicional en el análisis.

5.2. Operadores de Mutación

Muchos de los operadores de las herramientas ya existentes se basan en la idea de generar mutantes aleatorios en forma de fuerza bruta y sin un objetivo claro de qué es lo que el operador intenta detectar. Como consecuencia, con este tipo de operadores en caso de quedar vivos algunos mutantes se requiere de cierto esfuerzo intelectual para interpretar cómo matarlos y además no dan información muy precisa de cómo hacerlo. Por ello, se tuvo como objetivo modificar el concepto de Operadores de Mutación para que no sean únicamente un algoritmo de reemplazo sino que brinden al programador información más precisa para la corrección de los mutantes y, a su vez, permitan una distinción más clara de los casos de tests que pone a prueba cada operador.

Para ver concretamente cómo se persiguió este objetivo veremos el siguiente ejemplo concreto.

Supongamos que tenemos el siguiente método que define la igualdad de tarjetas de débito:

```
DebitCard>>= anotherDebitCard
    ^(type = anotherDebitCard type) and: [ number = anotherDebitCard number ]
```

Esto significa que una tarjeta es igual a otra si tienen igual tipo e igual número.

Una forma razonable de asegurar cierto grado de eficacia del conjunto de tests para este método, sería cerciorarse de estar probando cada uno de los resultados esperados al evaluarse en los siguientes escenarios:

1. Tarjetas con igual tipo y número.
2. Tarjetas que tengan igual tipo pero distinto número.
3. Tarjetas que tengan distinto tipo pero igual número.
4. Tarjetas que no tienen ni igual número ni igual tipo.

Sería bueno que mutation testing ayudara de alguna manera a ver si alguno de estos casos no está siendo probando.

Imaginemos que pasaría si se introdujera una mutación de manera tal que el método siguiera respondiendo igual en todos los escenarios listados anteriormente salvo en uno. Si corriéramos los tests contra esté mutante y ninguno de los tests fallara (mutante vivo), esto estaría indicando que los casos de prueba no son suficientes, ya que estaría faltando un test que pruebe que el comportamiento sea el correcto en ese escenario distintivo.

Por ejemplo, supongamos el primer escenario e imaginemos que generamos un mutante de la igualdad de tarjetas tal que funciona mal para este caso, pero funciona bien para los demás. Es decir, cuando dos tarjetas sean de igual tipo y número debería devolver *false* (comportamiento incorrecto) , pero además, en los demás casos también debería devolver *false* (comportamiento correcto).

Este mutante podría ser el siguiente:

```
DebitCard>>= anotherDebitCard
   ^false
```

Notar que se cumpliría lo que buscábamos anteriormente ya que siempre estaría evaluando a false.

Supongamos ahora que contamos con este caso de test:

```
DebitCardTest >>testDebitCardWithDifferentNumberShouldBeDifferent
   self deny: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 789).
```

Es claro que este test funcionaría correctamente, porque el método mutado siempre evalúa a *false*. Por lo tanto, este test no es suficiente para matar el mutante, y esto estaría indicando que faltan tests.

Se podría entonces agregar el siguiente caso de prueba para matar el mutante:

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual
   self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

De esta manera pudimos ver con este mutante que uno de los casos no estaba siendo testeado.

Si pasamos ahora al caso del segundo escenario, lo que se necesitaría sería un mutante en el cual el método devuelva *true* cuando las tarjetas son de igual tipo pero distinto número (comportamiento incorrecto), pero que también devuelva *true* para el primer caso y *false* para los restantes dos (comportamiento correcto)

Entonces el mutante podría ser el siguiente:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) and: [true]
```

Notar que para matar este mutante serviría el primer test pero no el segundo. Entonces tendríamos dos mutante diferentes que me aseguran que existan casos de prueba para las primeras dos situaciones.

Podemos seguir entonces con el mismo razonamiento y para los restantes dos escenarios se podrían generar los siguientes mutantes respectivamente:

```
DebitCard>>= anotherDebitCard
   ^true and: [ number = anotherDebitCard number ]
```

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) eqv: [ number = anotherDebitCard number ]
```

Donde el mensaje `#eqv:` define el operador lógico de equivalencia, es decir, el que es verdadero cuando las dos condiciones son verdaderas o las dos son falsas.

Para matarlos entonces se requerirían casos como los siguientes:

```
DebitCardTest >>testDebitCardWithDifferentTypeShouldBeDifferent
   self deny: (DebitCard masterNumbered: 123) = (DebitCard visaNumbered: 123).
```

```
DebitCardTest >>testDebitCardWithDifferentTypeAndDifferentNumberShouldBeDifferent
        self deny: (DebitCard masterNumbered: 123) = (DebitCard visaNumbered: 786).
```

Lo importante a ver en este ejemplo es que cada uno de los mutantes al quedar vivo estaría sugiriendo implementar cada caso de prueba en particular. O si se quitara solo uno de los tests, quedaría el respectivo mutante vivo (quedando el resto muertos). Esto significa que encontramos mutantes capaces de poner a prueba cada uno de los escenarios que sería deseable tener en la suite de tests.

A partir de este ejemplo, se podría generalizar un poco más la idea teniendo en cuenta el esquema general:

```
    condicion1 and: [condicion2]
```

Lo más probable es que esta condición esté determinando el comportamiento del programa de alguna manera. Es decir, si en algún momento de la ejecución se evalúa este código habrá un comportamiento

diferente dependiendo de si el resultado de la evaluación, y este resultado depende, a su vez, de la evaluación de *condicion1* y *condicion2*.

Un conjunto de tests suficientemente completo debería testear el comportamiento en todos los escenarios posibles y, en este caso, los escenarios posibles estarían dados por los resultados de evaluar *condicion1* y *condicion2*, cuyas combinaciones pueden verse en la tabla de verdad:

condicion1	condicion2	resultado
true	true	true
true	false	false
false	true	false
false	false	false

El primer escenario es un caso de evaluación en el que ambos son *true*. Sería bueno que algún test chequee el comportamiento del programa cuando esto sucede. Una forma de ver si éste test existe sería generando un mutante que tenga distinto comportamiento en este caso y sólo en este caso. La única forma de lograr esto sería reemplazando todo el código por *false*.

En el segundo caso, solo *condicion2* es false y *condicion1* es *true* y la forma de ver que este caso se esté probando con los tests es dejando solo *condicion1*, o lo que es lo mismo, aplicando la siguiente mutación:

```
condicion1 and: [true]
```

Es importante notar que solo en los demás escenarios el comportamiento sería el mismo ya que la tabla de verdad sigue siendo igual.

Y siguiendo el mismo razonamiento, se infiere que los mutantes requeridos para testear los restantes dos escenarios serían:

```
true and: [condicion2]
```

y

```
true eqv: [condicion2]
```

De esta manera, tendremos cuatro operadores de mutación que testean diferentes escenarios con cierta precisión.

Cabe aclarar que en estos escenarios estamos haciendo una simplificación y no estamos considerando la posibilidad de que para evaluar *condition1* o *condition2* se esté haciendo una recursión. Si este fuera el caso, este análisis no sería tan acertado porque el comportamiento sería más complejo de analizar. Además, el comportamiento a nivel global del programa podría cambiar de manera, a priori, impredecible.

Sin embargo en la práctica se pudo ver que en general, implementar los operadores de esta manera resulta en más mutantes vivos pero dando información más acertada. Como veremos más adelante esta información se mostrará en las herramientas en forma de sugerencia que permitirá al programador corregir y agregar tests o modificar el programa con el objetivo de eliminar estos mutantes.

5.2.1. Heurística

Se puede generalizar aún más esta idea e intentar extrapolarla para generar mutaciones de otros tipos. Partiendo de un esquema general de código (como por ejemplo el envío de un mensaje en particular) la idea general para generar operadores estaría dada entonces por la **heurística** de los siguientes dos pasos:

1. Plantear los diferentes escenarios que deberían ser testeados para un código con este esquema.

2. Para cada uno de los escenarios, crear un operador de mutación, para que, en el código mutado el comportamiento del programa sea distinto en el escenario respectivo, pero igual en los demás.

No obstante, no siempre es posible llevar a cabo el paso dos para todos los escenarios, porque en la práctica podría resultar imposible generar un mutante que diferencie el escenario en cuestión. En la sección 5.2.5 puede verse un ejemplo donde no siempre es posible generar un operador por cada uno de los escenarios.

Los operadores de mutación que se implementaron pueden clasificarse en **lógicos**, de **magnitud**, de **control de flujo**, de **excepciones**, de **colecciones** y **artiméticos**. A continuación veremos por cada una de éstas categorías, los principales operadores que se implementaron, mostrando en cada caso la utilidad de cada uno de ellos y los escenarios que estarían siendo testeados de haberse utilizado la heurística anterior.

5.2.2. Operadores lógicos

Anteriormente mostramos y analizamos los escenarios posibles para mensaje `#and:`. Análogamente se puede realizar el mísmo análisis para el mensaje `#or:`. En la siguiente tabla se listan los operadores lógicos a aplicar para ambos mensajes:

Operador	Sugerencia para matar mutante
Reemplazar `#and:` por *false*	Agregar caso de test en el que ambas condiciones se cumplen
Reemplazar argumento de `#and:` por *[true]*	Agregar caso de test en el que se cumple la primera condición pero no la segunda
Reemplazar receptor de `#and:` por *true*	Agregar caso de test en el que se cumple la segunda condición pero no la primera
Reemplazar mensaje `#and:` por `#eqv:`	Agregar caso de test en el que no se cumplen ninguna de las dos condiciones
Reemplazar `#or:` por *true*	Agregar caso de test en el que no se cumplen ninguna de las dos condiciones
Reemplazar argumento de `#or:` por *[false]*	Agregar caso de test en el que se cumple la segunda condición pero no la primera
Reemplazar receptor de `#or:` por *false*	Agregar caso de test en el que se cumple la primera condición pero no la segunda
Reemplazar mensaje `#or:` por `#xor:`	Agregar caso de test en el que ambas condiciones se cumplen

Algo a tener en cuenta de estos operadores, en especial de los que sugieren testear el caso en que una de las condiciones es *true* y la otra no, es que a veces pueden generar falsos positivos porque no hay forma de generar el test con el caso correspondiente.

Por ejemplo, se encontró el siguiente método que define la igualdad en una clase de Aconcagua:

```
ArithmeticObjectInterval>> = aCollection
   ^ self == aCollection
        or: [self class = aCollection class
            and: [from = aCollection from
                and: [by = aCollection by
                    and: [to = aCollection to]]]]
```

El método significa que dos intervalos son iguales si son el mismo objeto, esto es, si son la misma instancia ocupando la misma posición de memoria, o, en otro caso, si tienen el mismo elemento inicial, final y distancia entre elementos.

Si consideramos el siguiente mutante:

```
ArithmeticObjectInterval>> = aCollection
   ^false
         or: [self class = aCollection class
             and: [from = aCollection from
                 and: [by = aCollection by
                     and: [to = aCollection to]]]]
```

y evaluando el mutante para la misma instancia, se evaluaría la segunda condición del `#or:` resultando tambien válida.

Esto se debe a que el método esta aprovechando las características del *lazy evaluation* para que la comparación sea más eficiente y no siga la comparación en el caso de que sean el mismo objeto. Se puede observar, entonces, que estaríamos en presencia de un mutante equivalente al programa original y, por consiguiente, ante un falso positivo.

A su vez, existen casos similares a este en otros paquetes.

5.2.3. Operadores de magnitud

Con los mensajes de comparación de magnitudes (`>`,`<`, `<=`, `>=`) también se pueden plantear escenarios y utilizar la heurística. Por ejemplo, si tenemos:

```
   a <= b
```

Los casos serían:

1. a es igual a b
2. a es menor a b
3. a es mayor a b

Análogamente puede aplicarse el mísmo criterio para los operadores `>`,`<`, y `>=` lo que resultaría en los operadores de la siguiente tabla:

Operador	Sugerencia para matar mutante
Reemplazar `#<=` por `<`	Agregar caso de test en el que ambos valores son **iguales**
Reemplazar `#<=` por `=`	Agregar caso de test en el que el primer valor es **menor** al segundo
Reemplazar `#<=` por true	Agregar caso de test en el que el primer valor es **mayor** al segundo
Reemplazar `#<` por `<=`	Agregar caso de test en el que ambos valores son **iguales**
Reemplazar `#<` por false	Agregar caso de test en el que el primer valor es **menor** al segundo
Reemplazar `#<` por `!=`	Agregar caso de test en el que el primer valor es **mayor** al segundo
Reemplazar `#>=` por `>`	Agregar caso de test en el que ambos valores son **iguales**
Reemplazar `#>=` por true	Agregar caso de test en el que el primer valor es **menor** al segundo
Reemplazar `#>=` por `=`	Agregar caso de test en el que el primer valor es **mayor** al segundo
Reemplazar `#>` por `>=`	Agregar caso de test en el que ambos valores son **iguales**
Reemplazar `#>` por `!=`	Agregar caso de test en el que el primer valor es **menor** al segundo
Reemplazar `#>` por false	Agregar caso de test en el que el primer valor es **mayor** al segundo

5.2.4. Operadores de control de flujo

También se puede aplicar la heurística a mensajes cómo `#ifTrue:` e `#ifFalse:`.

Por ejemplo, si tuviéramos el siguiente método que devuelve el primer elemento de una coleccion ordenada:

```
OrderedCollection>>first
    self isEmpty ifTrue: [self error: 'the collection is empty!'].
    ^self at:1.
```

Considerando la línea del `#ifTrue:`, los escenarios a probar serían dos:

- la colección está vacía.
- la colección no esta vacía.

Los mutantes para distinguir estos dos casos serían entonces:

```
OrderedCollection>>first
    false ifTrue: [self error: 'the collection is empty!'].
    ^self at:1.
```

y

```
OrderedCollection>>first
    true ifTrue: [self error: 'the collection is empty!'].
    ^self at:1.
```

Los operadores resultantes serían entonces los de la siguiente tabla.

Operador	Sugerencia para matar mutante
Reemplazar receptor de `#ifTrue:` por *true*	Agregar caso de test en el que el receptor es false
Reemplazar receptor de `#ifTrue:` por *false*	Agregar caso de test en el que el receptor es true
Reemplazar receptor de `#ifFalse:` por *true*	Agregar caso de test en el que el receptor es false
Reemplazar receptor de `#ifFalse:` por *false*	Agregar caso de test en el que el receptor es true

Otro caso para el control de flujo de ejecución que se implementó es el de la devolución de objetos de un método. Es decir, el operador ^ de Smalltalk.

El único escenario a verificar para este operador sería que se esté devolviendo el objeto en cuestion. En este caso, si se implementa un operador que quite el ^, se estaría poniendo a prueba que existan casos de test que verifiquen que al evaluarse el método se devuelva el objeto correspondiente.

El operador resultante sería entonces:

Operador	Sugerencia para matar mutante
Remover ^	Agregar caso de test en el que se verifique que el objeto retornado sea el que corresponda

5.2.5. Operadores de Excepciones

Otro operador que se generó fue el que pone a prueba el manejo de excepciones. Veamos esto con el siguiente ejemplo de método con un *handler* de excepciones:

```
MeasureConverter>>canConvert: aSourceMeasure to: aTargetUnit
        [self findPathAndConvert: aSourceMeasure to: aTargetUnit]
                on: GraphPathNotFoundException
                do: [:ex | ^ ex return: false].
        ^ true
```

Aquí los escenarios serían los siguientes:

- Que se produzca la excepción y por consiguiente se evalúe el handler.
- Que no se produzca la excepción.

El operador que pone a prueba el primer escenario se implementó de manera tal que quite por completo el *handler*. Volviendo al ejemplo, el mutante resultante sería el siguiente:

```
MeasureConverter>>canConvert: aSourceMeasure to: aTargetUnit
        [self findPathAndConvert: aSourceMeasure to: aTargetUnit] value.
        ^ true
```

Para matar este mutante haría falta un test que valide un caso en que no haya posible conversión entre dos unidades y que el mensaje `#canConvert:to:` devuelva false por no encontrar camino en el grafo.

Habría que crear entonces el otro operador para el segundo escenario, cuyo método resultante generaría siempre la excepción. Pero el problema para construir este operador es que no siempre la instanciación de las excepciones se realiza de igual manera. En este caso, no se sabe si la excepción tendría que ser creada únicamente evaluando `#GraphPathNotFoundException signal` o si es necesario construirla utilizando determinados colaboradores.

5.2.6. Operadores de colecciones

Para el caso de las colecciones se implementaron operadores para mutar los mensajes más comunes que pueden recibir este tipo de objetos.

Un mensaje muy común que recibe todo tipo de colección es el que permite obtener otra colección con los objetos que cumplen con una condición. Estos son: `#select:` y `#reject:`. Para estos dos casos, se hicieron operadores que invalidan los filtro, devolviendo una colección igual o una vacía.

Por otro lado, algo que obviamente con frecuencia se hace con las colecciones es iterarlas. Para esto, en Smalltalk se envía el mensaje `#do:`, pasandole como argumento un bloque que se evaluará con cada uno de los objetos de la colección. Para estos casos se implementó un operador que invalida la iteración, reemplazando el bloque original por uno que no hace nada. Estos son entonces los operadores más relevantes que se implementaron para colecciones:

Operador	Sugerencia para matar mutante
Reemplazar argumento de `#select:` por `[:each | true]`	Agregar caso de test en el que se tenga que cumplir que un objeto no esté incluido en la colección devuelta por no cumplir la condición
Reemplazar argumento de `#select:` por `[:each | false]`	Agregar caso de test en el que se tenga que cumplir que un objeto esté incluido en la colección devuelta por cumplir la condición
Reemplazar argumento de `#reject:` por `[:each | true]`	Agregar caso de test en el que se tenga que cumplir que un objeto esté incluido en la colección devuelta por no cumplir la condición
Reemplazar argumento de `#reject:` por `[:each | false]`	Agregar caso de test en el que se tenga que cumplir que un objeto no esté incluido en la colección devuelta por cumplir la condición
Reemplazar argumento de `#do:` por `[:each |]`	Agregar caso de test que falle si no se itera la colección

5.2.7. Operadores Aritméticos

Para el caso de los operadores aritméticos simplemente se implementaron los operadores clásicos, que reemplazan una operación artimética por otra: + por - y viceversa, * por / y viceversa.

5.3. Aplicación de la Heurística

Una vez presentada la heurística y aplicada a algunas de las clasificaciones de los operadores, veremos un ejemplo donde se podrá conocer en detalle las ventajas de los operadores de mutación desarrollados.

En la sección 2.1.3 se vio que teniendo el siguiente método:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
         and: [ number = anotherDebitCard number ]
```

y con los siguientes casos de tests:

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual

        self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

```
DebitCardTest>>testDebitCardWithDifferentNumberShouldBeDifferent

        self deny: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 789).
```

si se aplica el operador de mutación que reemplaza el mensaje `#and:` por `#or:` se obtendría el siguiente mutante:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
         or: [ number = anotherDebitCard number ]
```

este operador de reemplazo de `#and:` por `#or:` genera, entonces, un único mutante que es eliminado por los tests anteriores.

Se vio también que si en lugar de utilizar el operador de mutación que reemplaza el mensaje `#and:` por `#or:` se utilizan los operadores:

- Reemplazar `#and:` por *false*
- Reemplazar argumento de `#and:` por *[true]*
- Reemplazar receptor de `#and:` por *true*
- Reemplazar mensaje `#and:` por `#eqv:`

se obtendrían los siguientes mutantes:

```
DebitCard>>= anotherDebitCard
   ^false
```

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) and: [true]
```

```
DebitCard>>= anotherDebitCard
   ^true and: [ number = anotherDebitCard number ]
```

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) eqv: [ number = anotherDebitCard number ]
```

de los cuales dos de ellos no son posible eliminarlos con los tests antes listados. Para poder eliminarlos se debería generar los siguientes casos de tests:

```
DebitCardTest >>testDebitCardWithDifferentTypeAndDifferentNumberShouldBeDifferent
        self deny: (DebitCard masterNumbered: 123) = (DebitCard visaNumbered: 786).
```

```
DebitCardTest >>testDebitCardWithDifferentTypeShouldBeDifferent
   self deny: (DebitCard masterNumbered: 123) = (DebitCard visaNumbered: 123).
```

lo que derivaría en la mejora de la suite de test.

5.3.1. Resultados

Al aplicar esta mísma lógica sobre los paquetes de *Pharo*: *Aconcagua*, *Magritte*, *VB-Regex*, *Network* y *Pier* se obtuvieron los resultados presentados a continuación:

Operador	Mutantes	Vivos	M. Score(%)
Replace and: with or:	151	40	73,51
Replace and: with nand:	151	8	94,7
Replace and: argument with [true]	151	60	60,26
Replace and: receiver with true	151	80	47,02
Replace and: with bEqv:	151	67	55,63
Replace and: with false	151	13	91,39

Cuadro 7: Resultados de and: en Aconcagua, Magritte, VB-Regex, Network y Pier

En el cuadro 7 se puede observar como utilizando los operadores de mutación creados según la heurística (los últimos cuatro) se obtiene, en general, un Mutation Score más bajos. Con esta diferencia en el resultado se puede ver que los tests del sistema no son tan completos como parecen ser con los mutantes generados por los primeros dos operadores y que existen una mayor cantidad de casos que deberían considerarse para mejorar los tests.

Al aplicar la mísma lógica sobre los operadores de mutación que reemplazan el mensaje `#or:` se obtuvieron los resultados del cuadro 8:

Operador	Mutantes	Vivos	M. Score(%)
Replace or: with and:	55	17	69,09
Replace or: argument with [false]	55	22	60
Replace or: receiver with false	55	26	52,73
Replace or: with bXor:	55	40	27,27
Replace or: with true	55	6	89,09

Cuadro 8: Resultados de or: en Aconcagua, Magritte, VB-Regex, Network y Pier

Con los operadores de mutación que aplican sobre el mensaje `#or:` también se puede observar como los operadores generados mediante la aplicación de la heurística obtuvieron un Score más bajo.

En los cuadros 9, 10, 11 y 12 pueden verse los mísmos resultados aplicados sobre los mensajes `#<=`, `#ifTrue:`, `#reject:` y `#select:` respectivamente.

Operador	Mutantes	Vivos	M. Score(%)
Replace <= with >	17	0	100
Replace <= with <	17	8	52,94
Replace <= with =	17	2	88,24
Replace <= with true	17	5	70,59

Cuadro 9: Resultados de <= en Aconcagua, Magritte, VB-Regex, Network y Pier

Operador	Mutantes	Vivos	M. Score(%)
Replace ifTrue: with ifFalse:	205	34	83,41
Replace ifTrue: receiver with false	205	86	58,05
Replace ifTrue: receiver with true	205	52	74,63

Cuadro 10: Resultados de ifTrue: en Aconcagua, Magritte, VB-Regex, Network y Pier

Operador	Mutantes	Vivos	M. Score(%)
Replace reject: with select:	12	0	100
Replace reject: block with [:each | true]	12	0	100
Replace reject: block with [:each | false]	12	2	83,33

Cuadro 11: Resultados de reject: en Aconcagua, Magritte, VB-Regex, Network y Pier

Operador	Mutantes	Vivos	M. Score(%)
Replace select: with reject:	17	4	76,47
Replace select: block with [:each | false]	17	4	76,47
Replace select: block with [:each | true]	17	6	64,71

Cuadro 12: Resultados de select: en Aconcagua, Magritte, VB-Regex, Network y Pier

Podemos decir entonces que utilizando la heurística se crearon operadores de mutación que generan scores más bajos. Tener scores bajos significa que más mutantes quedaron vivos. Esto es deseable para los operadores porque implica que se detectan una mayor cantidad de casos de tests no creados. Esta reducción en los scores se debe a que cada operador de mutación generado por la heurística distingue un caso en particular, mientras que los otros operadores no generan esta distinción.

5.4. Herramientas

Con el objetivo de utilizar Mutation Testing en el desarrollo diario de aplicaciones en *Smalltalk*, se desarrollaron algunas herramientas que pueden ser descargadas y utilizadas en *Pharo* [Pha] y que facilitan la aplicación de la técnica de mutation testing para desarrollos de aplicaciones. Estas herramientas son:

- **Mutation Testing Runner**: Runner principal donde se permite configurar el paquete y las clases que se mutarán, los operadores de mutación a aplicar y la estrategia de coverage utilizada.
- **Mutation Result Browser**: Browser donde se lista una mutación en particular. Aquí se puede ver la mutación generada, la suite de test utilizada para intentar matar al mutante y una sugerencia para intentar matar al mutante en caso de necesitarlo.
- **Mutation Testing integrado en Browser**: Se agregaron opciones de menú y shortcuts para los browsers de desarrollo de forma tal que sea facil ejecutar la técnica sin necesidad de abrir el runner.
- **Mutation Testing Analysis Result**: Browser donde se permite observar el resultado de la ejecución de la técnica de mutation testing. Se listan los mutantes que fueron eliminados y aquellos que quedaron vivos junto a un resumen del resultado general de la ejecución.

A continuación se detallan cada una de las herramientas mencionadas.

5.4.1. Mutation Testing Runner

El Runner es similar al Runner de SUnit. Se utiliza para ejecutar la técnica de Mutation Testing pero con la posibilidad de configurar: las clases del modelo sobre la que se aplicarán las mutaciones, los operadores de mutación a utilizar y los tests utilizados para eliminar los mutantes. A su vez, se agregó la posibilidad de correr 4 estrategias de coverage distintas que permiten optimizar tiempos en las fases de generación o eliminación de mutantes.

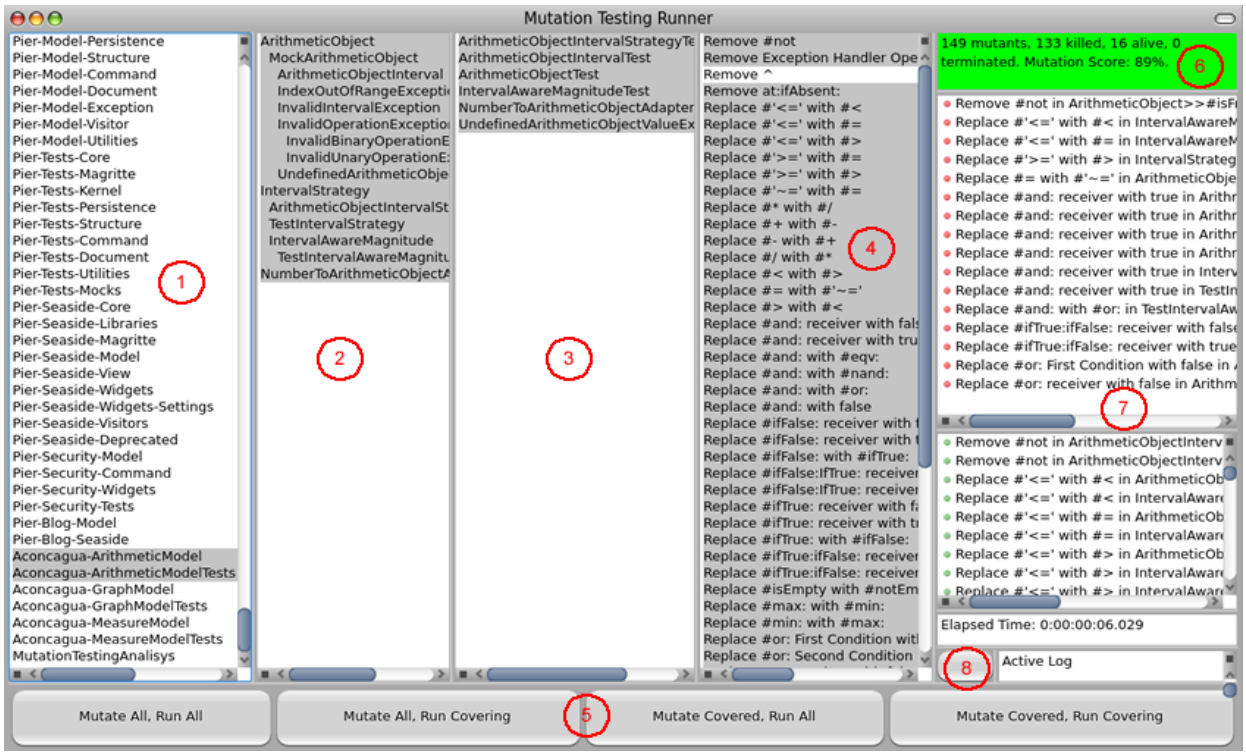

Figura 12: Mutation Testing Runner

El browser cuenta de 8 paneles o secciones principales (ver figura 12). En el panel 1 se listan todas las categorías (o paquetes) existentes en la imagen. Alli se permiten seleccionar una o más categorías sobre las que se desea realizar la ejecución. En base a la selección realizada, se listarán las clases de modelo en el panel 2 y las clases de tests en el panel 3. En el panel 4 se listan todos los operadores de mutación disponibles y se permite seleccionar los operadores a utilizar. Una vez seleccionadas las clases de modelo, los tests y los operadores de mutación a utilizar, se debe seleccionar cual de las 4 estrategias posibles de coverage se utilizará para generar y eliminar los mutantes (ver Sección 5.1). Estas estrategias se seleccionan con los botones disponibles en el panel 5.

Al ejecutar la técnica de mutation testing se devuelven los correspondientes resultados. En el panel 6 se puede observar un resumen del resultado de la ejecución. Alli se permite conocer la cantidad de mutantes generados y los resultados obtenidos al intentar eliminar dichos mutantes (numero de mutantes eliminados, numero de mutantes terminados y numero de mutantes que quedaron vivos). A su vez, es posible observar el Mutation Score (mutantes eliminados respecto a mutantes generados) correspondiente al resultado de la ejecución. Este panel es coloreado con distintos colores de acuerdo al Mutation Score obtenido:

- Verde si el Score supera el 80 %. Esto significa que el score es “aceptable”.
- Amarillo si el Score se encuentra entre 50 % y 80 %. Esto significa que el score no llega a ser aceptable.
- Rojo si el Score no supera el 50 %. Este Score se considera demasiado bajo y debería ser mejorado.

En el panel 7 se listan cada uno de los mutantes generados separando los eliminados y terminados de aquellos que quedaron vivos. Esto se debe a que los mutantes que quedaron vivos son aquellos mutantes que deberían ser atacados (agregando o corrigiendo tests) para tratar de eliminarlos. Es posible acceder a cada uno de los mutantes listados seleccionando el mutante en cuestión; al seleccionarlo se desplegará el *Mutation Result Browser* donde se podrá observar el cambio realizado y los tests ejecutados (ver sección 5.4.2). Los mutantes tienen iconos de colores que identifican el estado del mutante: el ícono rojo identifica a los mutantes que quedaron vivos, el icono verde significa que el mutante fue eliminado y el icono amarillo significa que fue necesario terminar la ejecución de los tests del mutante por tratarse de un mutante que probablemente se quedó ciclando.

Finalmente, en el panel 8 se listan los tiempos de ejecución utilizados para correr la técnica y se permite activar un log que escribirá en un archivo a medida que ejecuta cada mutante. Este log es utilizado en caso de que la ejecución bloquee la imagen y se necesite detectar en que ejecución falló.

5.4.2. Mutation Result Browser

El *Mutation Result Browser* permite detallar los componentes de un mutante. Aquí es posible ver el cambio que originó el mutante junto a los resultados de los tests que se corrieron con el objetivo de eliminarlo. A su vez se listan sugerencias para el caso de que el mutante deba ser eliminado y se permite acceder a ciertas funcionalidades que facilitan la modificación del modelo con el objetivo de eliminarlo.

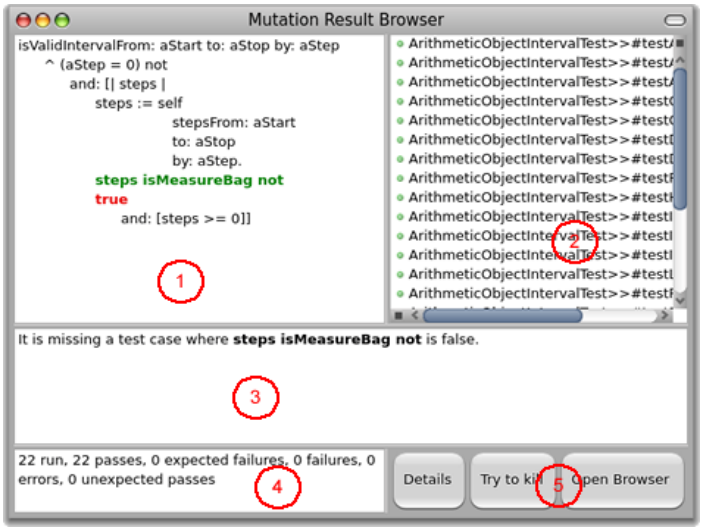

Figura 13: Mutation Result Browser

El browser cuenta principalmente de 5 paneles o secciones principales (ver figura 13). En el panel 1 se detalla el método modificado, en verde se puede observar el código original y en rojo el código insertado para generar el mutante. En el panel 2 se listan los tests evaluados al intentar eliminar el mutante (el color determina si el test fallo o no). En el panel 3 se pueden observar, en caso de que el mutante no haya sido eliminado, sugerencias que pueden ser aplicadas con el objetivo de eliminarlo. Generalmente las sugerencias tienen que ver con tests que estén faltando.

Finalmente en el panel 4 se puede observar un resumen de la cantidad de la suite de test ejecutada para eliminar el mutante (cantidad de tests ejecutados, fallidos, etc) y en el panel 5 se listan 3 botones:

- **Details** muestra información detallada del mutante en un Workspace (ver Figura 15).
- **Try to Kill** permite reejecutar el mutante y volver a correr los tests para ver si con nuevos tests o los mismos tests corregidos el mutante es eliminado. Aquí se despliegan 3 posibilidades (ver Figura 14): *Try to kill using Same Tests*, *Try to kill using Tests in Same Classes*, *Try to kill using All tests in same packages*. Cada una de estas posibilidades permite seleccionar nuevos tests para eliminar el mutante. Por ejemplo, *Same tests* se utilizaría en caso de que se haya modificado alguno de los tests listados; *Tests in Same Classes* se utilizaría en caso de que se haya agregado un nuevo tests en alguna de las clases de los tests listados; y *All Tests in same packages* se utilizaría en caso de que se haya agregado una nueva clase para eliminar el mutante.

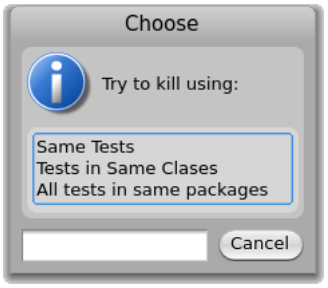

Figura 14: Try to Kill Selection

- **Open Browser** es un shortcut para desplegar el browser en la clase de test que se encarga de testear dicha funcionalidad (se selecciona de acuerdo a la mayoría de tests existente).

5.4.3. Mutation Testing integrado en Browser

Para facilitar la ejecución de la técnica durante el desarrollo de aplicaciones se crearon opciones de menú y shortcuts en el browser de clases. Estas nuevas opciones permiten ejecutar Mutation Testing

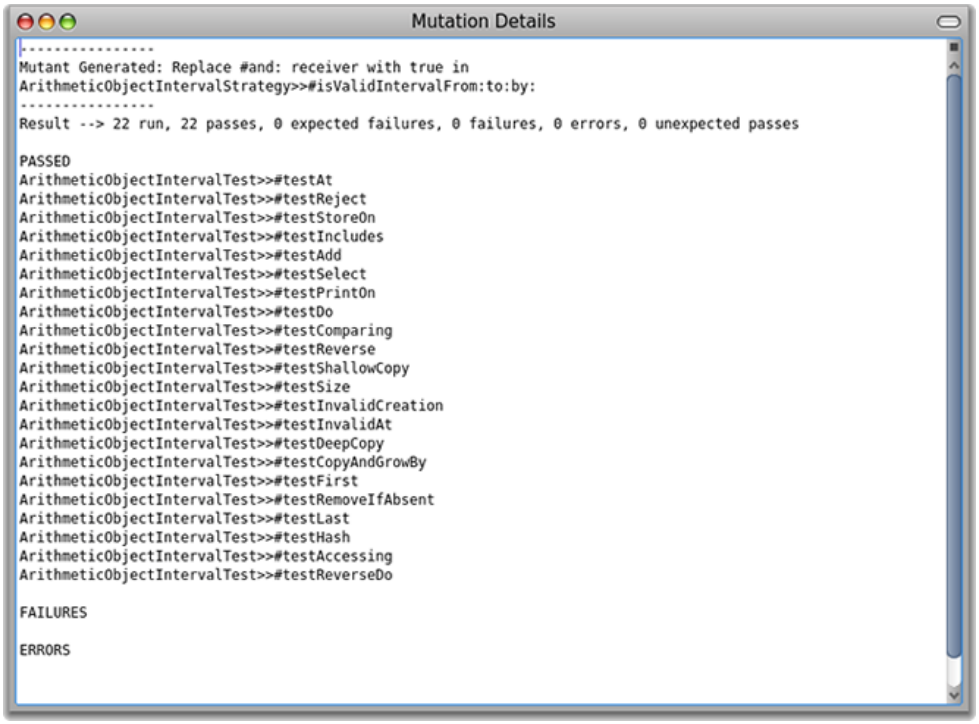

Figura 15: Mutation Details

desde clases de Tests. Para hacerlo, se debió tomar ciertas decisiones como a que clases del modelo se le aplicarán las mutaciones y que estrategia de coverage utilizar. Para ello se tomó la siguiente decisión: Al ejecutar la técnica de mutation testing desde el browser de clases se ejecuta la estratégia de Coverage *Covered-Covering* utilizando como clases de modelo las clases definidas en la misma categoria que el test en cuestión pero excluyendo la palabra "Tests´´.

Por ejemplo, si ejecutamos la técnica de Mutation Testing desde el browser de clases estando en una clase de test definida en la categoría *Aconcagua-ArithmeticModelTests* se ejecutará coverage sobre las clases definidas en la categoría *Aconcagua-ArithmeticModel* y se aplicará Mutation Testing sobre esos métodos cubiertos.

Al igual que como se utiliza SUnit, es posible ejecutar Mutation Testing para:

- un método en particular
- una categoría en particular
- toda una clase de test
- un paquete de test completo.

Cada una de estas opciones pueden ser ejecutadas desde el menu contextual que se despliega de cada uno de los paneles del browser (ver Figura 16).

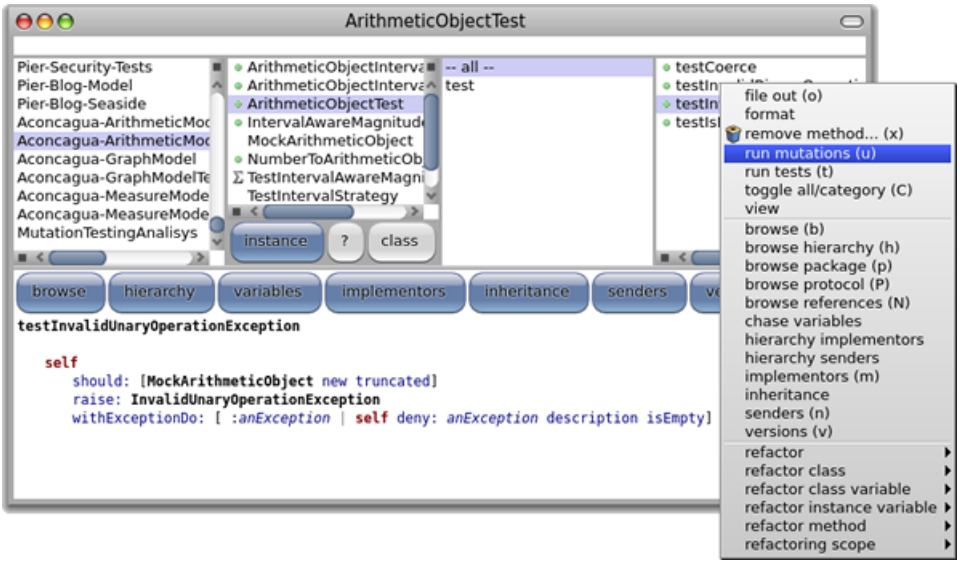

Figura 16: Mutation Testing integrado en Browser

Por último, al finalizar la ejecución se despliega una ventana donde es posible analizar el resultado obtenido.

5.4.4. Mutation Testing Analysis Result

El *Mutation Testing Analysis Result* es una ventana que detalla el resultado de la ejecución de la técnica de Mutation Testing al ser ejecutado desde el Browser de desarrollo. Esta ventana cuenta con algunos de los paneles presentados en el Mutation Testing Runner.

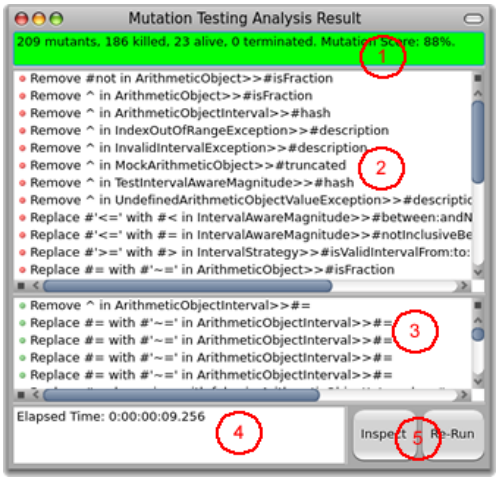

Figura 17: Mutation Testing Analysis Result

Como se puede ver en la figura 17, la ventana cuenta con 5 paneles principales. En el panel 1 se resume el resultado de la ejecución. En el panel 2 y 3 se listan los mutantes que sobrevivieron y aquellos que fueron eliminados o terminados respectivamente. En el panel 4 se muestra el tiempo utilizado para obtener el resultado y, en el panel 5 se presentan 2 botones (*Inspect* y *Re-Run*) que permiten inspeccionar el resultado y volver a ejecutar el análisis de Mutation Testing.

5.5. Implementación

5.5.1. Modelo

A continuación se presenta una descripción de las principales clases del modelo junto con un detalle de la interacción principal de los objetos.

La clase principal dentro del modelo de Mutation Testing desarrollado es *MutationTestingAnalysis*. Una instancia de esta clase es la que se encarga de ejecutar el análisis de Mutation Testing y recolectar los resultados que se listan en los browsers antes presentados. Para comprender la interacción entre los objetos, observemos el siguiente diagrama de secuencia:

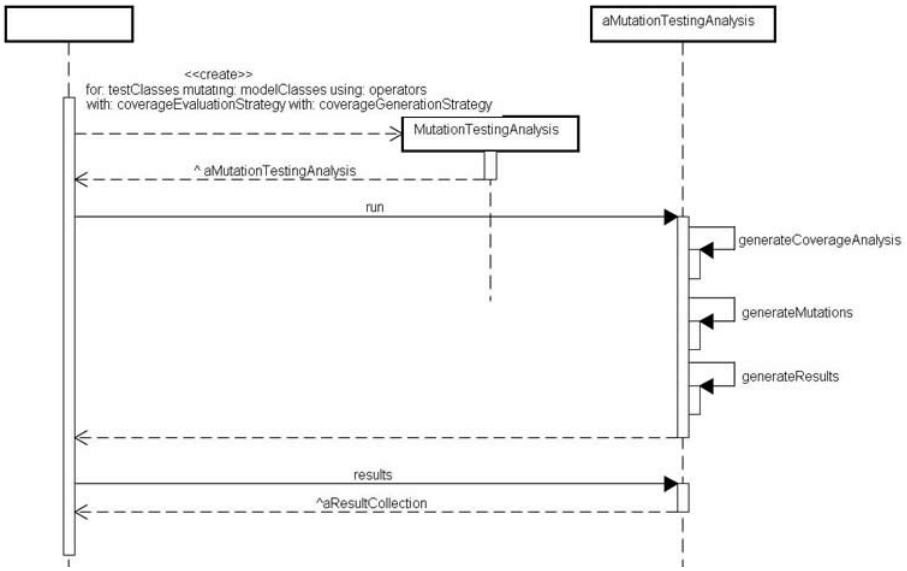

Figura 18: Diagrama de Secuencia de Mutation Testing Analysis

Como se puede observar en el diagrama 18, al ejecutar la técnica de Mutation Testing se crea una instancia de *MutationTestingAnalysis* tomando:

- Las clases de test que se utilizarán para evaluar los mutantes.
- Las clases de modelo sobre las que se construirán los mutantes y se evaluará el Coverage.
- El conjunto de operadores de Mutación a aplicar.
- La estrategia de coverage a utilizar para la evaluación de los mutantes.
- La estrategia de coverage a utilizar para la generación de los mutantes.

Una vez generada una instancia de *MutationTestingAnalysis*, se ejecuta el método *run*. La ejecución de la técnica se divide en 3 etapas: El análisis de Coverage, la geneneración de mutantes y la generación de los resultados.

Análisis de Coverage La primer etapa es la generación del análisis de Coverage (*generateCoverageAnalysis*) que será utilizada al momento de construir los mutantes y de generación de los resultados. Para esta etapa interviene la clase *CoverageAnalysis* ejecutando los tests y obteniendo un análisis de coverage por cada uno de los tests ejecutados.

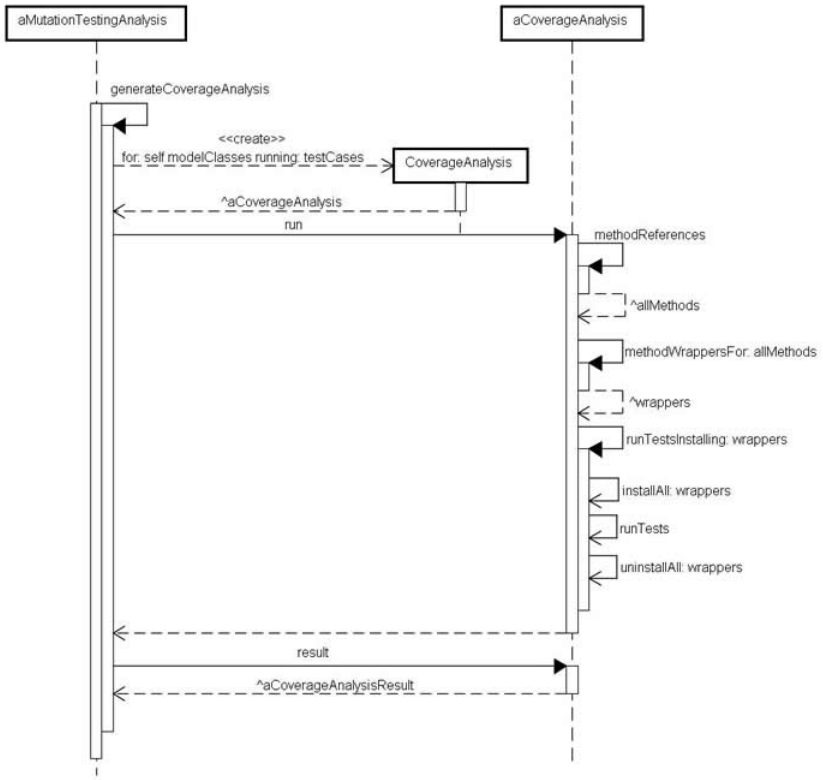

Figura 19: Diagramas de Secuencia del Análisis de Coverage

En la figura 19 se muestra como se obtiene el resultado de coverage. Al ejecutar el análisis de coverage, se obtienen todos los métodos del modelo y para cada uno se genera un *Method Wrapper* [BFJR98] (ver sección 5.5.2). Una vez obtenidos todos los wrappers de los métodos del modelo, se procede a instalarlos, evaluar los tests y, finalmente, desinstalarlos.

Al ejecutar los tests usando los wrappers es posible determinar cuales fueron los métodos cubiertos por cada uno de los tests. Este resultado es guardado en el análisis de mutation testing y es utilizada con las estrategias de generación de mutantes o las estrategias de ejecución.

Geneneración de Mutantes La segunda etapa es la de generación de mutantes. Esta etapa se lleva a cabo al evaluarse el método *generateMutations*. Como se puede observar en el diagrama 20, el análisis colabora con la instancia de la estrategia de generación de mutantes elegida y esta estrategia utiliza los métodos del modelo seleccionado para generar los mutantes.

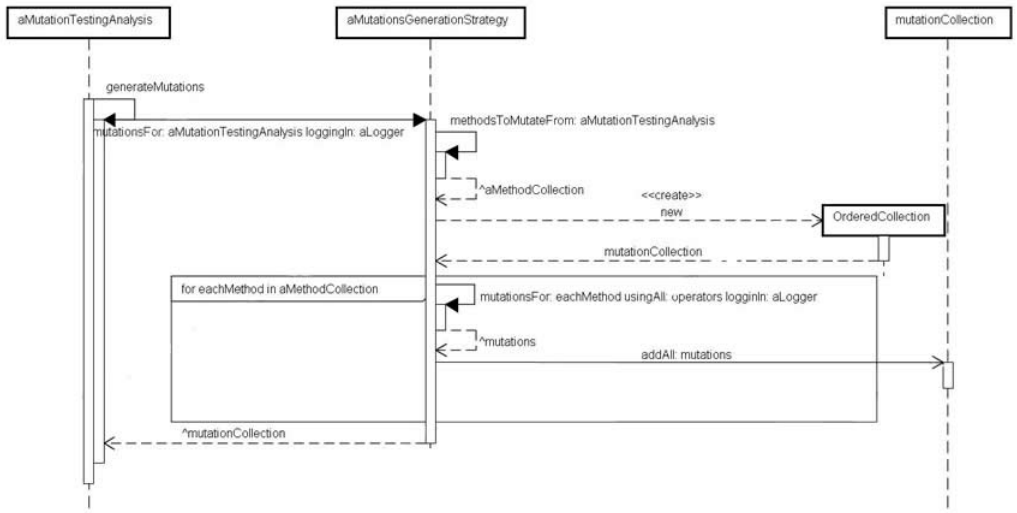

Figura 20: Diagrama de Secuencia de Generación de Mutantes

En la figura 21 se puede observar como colaboran las distintas estrategias de generación de mutantes (*AllMutationsGenerationStrategy* y *SelectingFromCoverageMutationsGenerationStrategy*) para obtener los métodos del modelo sobre los cuales se aplicarán los operadores de mutación.

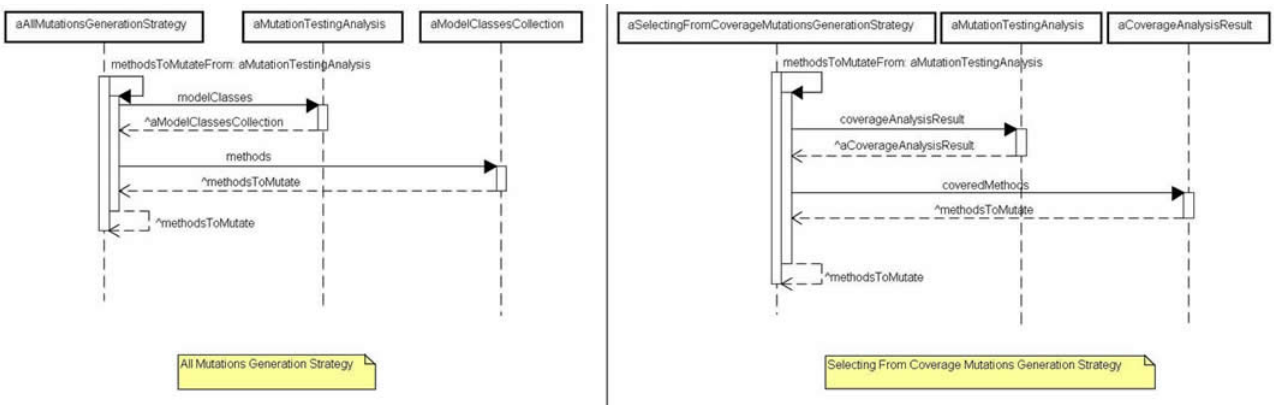

Figura 21: Diagramas de Secuencia de Estrategias de Generación de Mutantes

La estrategia *All Mutations Generation Strategy* obtiene todas las clases del modelo colaborando con el análisis y de cada clase del modelo obtiene los métodos de instancia y de clase. Esos métodos son los métodos que se consideran al utilizar esta estrategia y sobre los cuales trabajarán los operadores de mutación seleccionados para obtener los mutantes.

La estrategia *Selecting From Coverage Mutations Generation Strategy* obtiene del análisis el resultado del coverage aplicado y de este resultado obtiene los métodos cubiertos por los tests seleccionados. Esos métodos son los que serán usados para generar los mutantes.

Una vez obtenidos todos los métodos a mutar se procede a generar los mutantes. Para ello el Mutation Testing Analysis colabora con los operadores de mutación seleccionados aplicando los cambios correspondientes.

Los operadores de mutación son subclases de *MutantOperator* y responden principalmente a 4 mensajes:

- **description** Retorna la descripción del operador de mutación. Se utiliza al momento de listar los operadores. Debe ser lo suficientemente clara como para que sea fácil determinar que realiza dicho operador tan solo con leer la descripción. Por ejemplo:

```
ReplaceAndWithFalseOperator>>description
        ^'Replace #and: with false'.
```

- **expressionToReplace** Determina cual es la expresión que se desea reemplazar. Por ejemplo:

```
ReplaceAndWithFalseOperator>>expressionToReplace
        ^'''@object and: ''@aBlock'
```

  Este operador se aplicará siempre que se envíe a un objeto el mensaje *and:* pasandole como segundo argumento un bloque.

- **newExpression** Determina cual es la expresión que se insertará en lugar de la expresión a reemplazar. Por ejemplo:

```
ReplaceAndWithFalseOperator>>newExpression
        ^'false'
```

- **suggestionFor:Using:** Permite determinar *Sugerencias* a la hora de eliminar un mutante. Estas sugerencias son listadas al ver el mutante generado y su función es ayudar a detectar que tests o correcciones se deben realizar para eliminar el mutante. Por ejemplo, si se aplica el operador de mutación *Replace #and: with false* al método:

```
AClass>>aMethod
        ^condition1 and: [condition2]
```

  En caso de que el mutante no haya sido eliminado por ningún test, se desplegará la siguiente sugerencia:

  It is missing a test case where both conditions ***condition1*** *and* ***[condition2]*** *are true.*

  Esto se debe a que los tests que cubren ese método solo testean cuando alguna de las condiciones no se cumple pero no cuando ambas son verdaderas. De existirlo, con este mutante instalado el test fallaría. Por lo tanto, la aplicación sugiere que se agrege un test con ambas condiciones verdaderas (ver Sección 5.2).

Geneneración de Resultados La tercer etapa dentro del Análisis de Mutation Testing es la de generación de resultados. Una vez obtenidos todos los mutantes se procede a instalar cada mutante, ejecutar la suite de test correspondiente (utilizando la Estrategia de Evaluación seleccionada), obtener los resultados y finalmente desinstalarlo.

En la figura 22 se puede observar como colaboran los objetos que se encargan de generar los resultados.

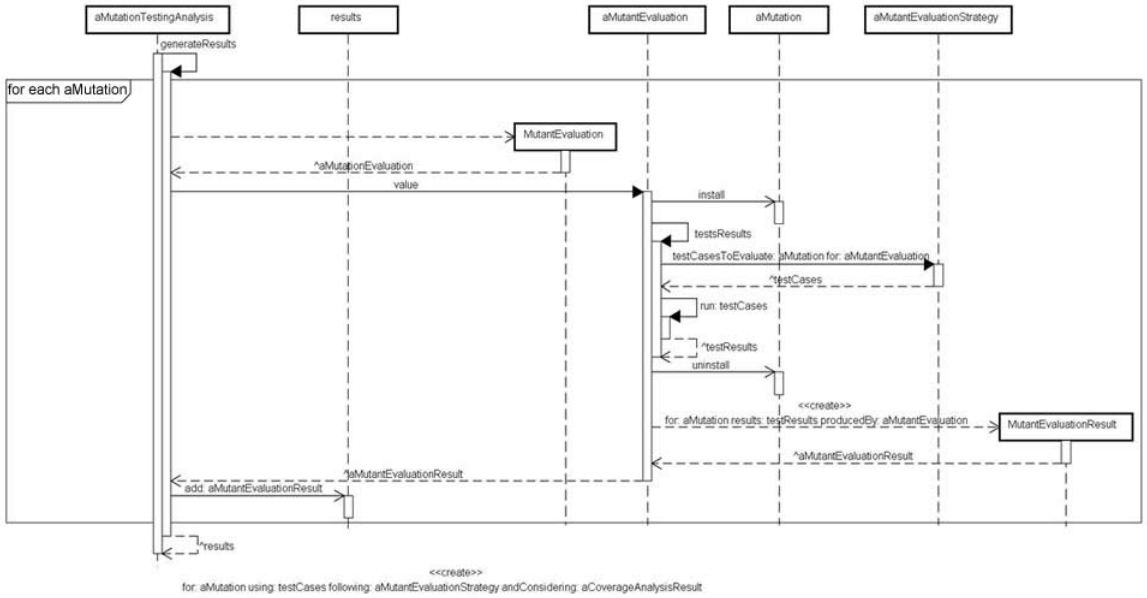

Figura 22: Diagramas de Secuencia de Generación de Resultados

Por cada mutante generado, se construye un *MutantEvaluation*. Esta evaluación se encarga de instalar el mutante; obtener los tests a correr colaborando con la estrategia de evaluación seleccionada (*AllTestsMethodsRunningMutantEvaluationStrategy* o *SelectingFromCoverageMutantEvaluationStrategy*); desinstalar el mutante y construir un resultado a devolver (*MutantEvaluationResult*) con los resultados de los tests ejecutados.

Las estrategias de evaluación son la siguientes:

- **AllTestsMethodsRunningMutantEvaluationStrategy**: Al colaborar con instancias de esta clase, retorna todos los tests de las clases de test seleccionadas.
- **SelectingFromCoverageMutantEvaluationStrategy**: Al colaborar con instancias de esta clase, retorna solo los tests cubiertos por el método mutado.

5.5.2. Meta-programación

Aprovechando las facilidades reflexivas del ambiente, se hizo uso de técnicas de meta-programación para la implementación de algunos de los pasos requeridos para llevar a cabo un análisis de Mutation Testing como el análisis de coverage, la generación de mutaciones y la instalación de las mismas para generar los mutantes.

Para el caso del análisis de coverage se utilizó una técnica llamada *Method Wrapper* [BFJR98] que permite agregar comportamiento adicional en la evaluación de un método, sin tener que recompilarlo. Básicamente, consiste en cambiar, en la clase en la que está definido, el método original por uno nuevo que lo *envuelve*. De esta manera, antes y/o después de evaluarse el método original, se evalúan colaboraciones que pueden servir para diferentes propósitos. Esto se puede lograr en partes, gracias a que los métodos son objetos del metamodelo.

En la primera implementación que se realizó, se generaba un method wrapper para cada método del modelo. Este wrapper servía para marcar el método una vez que se evaluaba al ejecutar un test. Una vez corrido el test, se recolectaban todos los métodos marcados y se reinstalaban los wrappers para correr el siguiente test. Si bien esta implementación sirvió en principio para seguir con el desarrollo, los tiempos utilizados para el análisis de coverage eran excesivos. Por esta razón, se realizó una segunda implementación en la que sólo se instalan los method wrappers una única vez antes de correr toda la suite de tests. Para construir el resultado de coverage, en lugar de *marcar* los métodos cubiertos se utiliza un conjunto en cada method wrapper que guarda los tests que cubren dicho método. De esta

manera se evita tener que iterar sobre tódos los métodos del modelo por cada test corrido y se evita tener que instalar todos los wrappers por cada una de estas iteraciones.

Para el caso de la generación de mutantes, se aprovechó nuevamente la ventaja de que los métodos son objetos del metamodelo (instancias de la clase *CompiledMethod*), y que saben responder al mensaje `#parseTree` con el árbol de parsing (árbol de análisis sintáctico) del método que representan. Entonces, como primer paso para mutar, por ejemplo, el envío de un mensaje en un método, se ubica en el árbol del método el nodo correspondiente al envío del mensaje en cuestión. Luego se genera un árbol similar pero con la modificación correspondiente del cual se puede obtener y compilar el nuevo código mutado. Para esto, una vez obtenido este código, se puede conseguir una nueva instancia de CompiledMethod colaborando con un objeto compilador, que también es parte del metamodelo.

La instalación de una mutación se realiza reemplazando el método original por el método mutado en la clase correspondiente. Para esto, el reemplazo se hace en un diccionario de la clase llamado *methodDictionary*[BFJR98], que tiene como valores a los métodos de la clase y como claves a los selectores de estos métodos. De esta manera, utilizando como clave el selector, se puede reemplazar el método original por uno mutado para la instalación, conservando la instancia de CompiledMethod original para permitir la posterior desinstalación.

6. Conclusiones

En este trabajo se obtuvieron resultados en dos aspectos: por un lado en las mejoras a la técnica de Mutation Testing y por el otro en las herramientas para el uso de la misma.

Con respecto a las mejoras a la técnica, en primer lugar, se lograron optimizaciones mediante las cuales se obtuvieron reducciones significativas en los tiempos totales de respuesta, que son un factor importante a la hora de llevar Mution Testing a la práctica. Estas optimizaciones se basan en la utilización de resultados de un análisis previo de Code Coverage para evitar evaluaciones de tests y/o generaciones de mutantes innecesarias. Cuantitativamente, gracias a estas optimizaciones, las reducciones de los tiempos originales fueron, en el promedio de los casos analizados, de un 50 %, llegando en los casos de modelos de software más grandes a reducciones de hasta un 97 %.

A su vez, también se logró una mejora en la calidad de los resultados que la técnica provee. Para lograr esta mejora, se creó y se utilizó una heurística para crear operadores de mutación. Esta heurística permite que, a partir de un esquema general de código (como por ejemplo el envío de un mensaje en particular) y luego de identificar los diferentes escenarios que deberían ser probados, se puedan crear operadores que pongan a prueba cada uno de estos escenarios por separado. Estos operadores generan mutantes que, en caso de quedar vivos, permiten identificar con mayor precisión los casos que faltarían ser probados.

Por otro lado, se implementaron herramientas integradas al ambiente de desarrollo que permiten utilizar la técnica de *Mutation Testing* de manera rápida y ágil, facilitando su uso en conjunto con el proceso de desarrollo y permitiendo la mejora continua del testing del sistema. A diferencia de las herramientas existentes para otros ambientes, en donde sólo se permiten análisis de forma batch, en este caso se agregaron además facilidades para llevar a cabo en forma eficiente análisis que apliquen mutaciones a porciones (clases o paquetes) o a la totalidad del modelo bajo análisis, opcionalmente desde las mismas herramientas en que se lleva a cabo el desarrollo normalmente. Permitiendo además que, una vez realizado un análisis, con cada mutante resultante por separado se pueda: inspeccionarlo, ver los casos de tests que lo cubren, agregar casos para eliminarlo y volver a evaluarlo dinámicamente.

El modelo reflexivo de Smalltalk, dado por su definición meta-circular, resulta de gran ayuda para trabajos como este. En este caso permitió la implementación de un modelo sencillo, eficiente y extensible que facilitó la resolución de problemas críticos de la implementación de una manera más simple, que en ambientes sin estas características hubiesen implicado modelos mucho más complejo. Ejemplos de esto son: la generación e instalación de mutantes, el análisis de coverage, la facilidad con la que se pueden agregar nuevos operadores de mutación y la manera mas o menos inmediata con que se pueden extender las herramientas existentes en el ambiente para integrar las herramientas de Mutation Testing. Además, gracias al dinamismo del ambiente, la instalación de mutantes resulta naturalmente eficiente por no tener el cuello de botella de compilación y enlace que suelen tener los ambientes estáticos, problema que en un principio contribuyó a imposibilitar el uso práctico de la técnica.

Sin embargo, una posible desventaja de un ambiente de este tipo en donde conviven objetos de diferentes niveles conceptuales (los objetos del meta-modelo, los que tienen que ver con el funcionamiento de la máquina virtual, los que definen las herramientas de desarrollo y los del nivel *base*, es decir, objetos que definen un modelo de negocio en particular) es la facilidad de cometer errores que desestabilicen el ambiente por completo. Por ejemplo, al aplicarse mutaciones en objetos que tienen que ver con el manejo de procesos, puede llegarse a interrumpir o a alterar el normal funcionamiento de la imagen. Este es un problema que, de hecho, se presentó en este trabajo al realizarse pruebas de Mutation Testing sobre modelos que resuelven problemas de bajo nivel. En conclusión, este tipo de herramientas deben ser utilizadas con precaución en ciertos casos.

Con respecto a la utilidad de Mutation Testing en general, puede decirse que es una técnica que resulta beneficiosa ya que no solo permite su utilización para la obtención de información estadística con respecto a la calidad de los tests sino que además genera una retroalimentación que sirve para asistir en la mejora y extensión de los casos de tests existentes. Esta idea se tuvo en cuenta a la hora de implementar operadores de mutación y al crear la heurística antes mencionada.

A su vez, algo a tener en cuenta de los operadores de mutación, es que si bien existen algunos que suelen poder aplicarse con frecuencia en casi todos los modelos, existen otros que son aplicables a

unos pocos. Esto se pudo ver en la práctica, por ejemplo, con algunos operadores que mutan el envío de mensajes aritméticos. Además, teniendo en cuenta la utilización de la técnica en un ambiente de objetos, a medida que se crean abstracciones nuevas que definen nuevos objetos, puede necesitarse implementar otros operadores que apliquen a los nuevos protocolos. Una idea que podría resultar interesante, es que cada modelo a ser reutilizado, venga acompañado por un conjunto de operadores particulares para éste.

Por otro lado, esta técnica no reemplaza a otras existentes como Code Coverage sino que la complementa. Como ya vimos en la introducción de este informe, tener una cobertura del 100 % no significa que estén todos los casos probados. Con un análisis de Mutation Testing se podrían detectar muchos casos no probados a pesar de tener todo los métodos cubiertos. En forma análoga, que todos los mutantes queden muertos no significa que todo el modelo esté cubierto ya que podrían existir métodos para los cuales no se aplicó ninguna mutación, por lo que no existiría ningún mutante que advierta esto. Una posibilidad para combinar ambas técnicas podría ser alcanzar cierto grado de cobertura en el modelo testeado para luego aplicar Mutation Testing profundizando el alcance de los tests. Los beneficios de tener un conjunto de tests completos pueden verse especialmente en el desarrollo de programas que cambien y evolucionen en el tiempo, en donde a la hora de introducir cambios, como por ejemplo modificaciones en el diseño o arquitectura del programa, se puede garantizar que el programa siga teniendo el comportamiento adecuado en un conjunto de casos más completo.

7. Trabajo Futuro

Existen diferentes aspectos en los que se debería continuar la investigación del trabajo actual. Por un lado, se podrían mejorar la usabilidad de las herramientas, optimizando la calidad de la información brindada, haciendolas más flexibles y agregando facilidades para mayor asistencia al proceso de desarrollo. Por otro lado, las mejoras podrían realizarse en cuestiones que tienen que ver con la aplicación de la técnica en sí, como la eficiencia en los tiempos o en la calidad de los resultados generados.

7.1. Herramientas

En primer lugar, sería bueno contar con alguna manera de documentar los falsos positivos, esto es, casos de mutantes que quedan vivos por ser equivalentes al programa original. Es decir, sería bueno que al ser detectados estos casos, se pueda indicar de alguna manera que son mutantes equivalentes y poder excluirlos de los resultados de los siguientes análisis, evitando tener que volver a lidiar con ellos reiteradas veces.

Otra idea a investigar es la posibilidad de contar con alguna forma de automatizar la creación de los tests que debieran agregarse a partir de un mutante vivo. Actualmente la herramienta brinda sugerencias que facilitan al programador la mejora de los tests pero no permite la generación de los mismos de alguna manera más automatizada. Una posibilidad a analizar es la de desarrollar alguna funcionalidad para que, a partir de los tests existentes y la información del mutante, la herramienta pueda inferir y luego asistir en la creación de los casos de prueba faltantes.

Con respecto a los operadores de mutación, sería deseable contar con clasificaciones de los mismos que permitan selecciones más flexibles de los mismos a la hora de realizar los análisis. Por ejemplo, se podrían clasificar los operadores por eficiencia, categoría (aritméticos, lógicos, de colección, etc.) o, incluso, permitir la posibilidad de definir conjuntos por parte de los usuarios de la herramienta. Esta opción permitiría una configuración más acorde a cada caso en que se estén usando las herramientas.

7.2. Técnica

Con respecto a la utilización de la técnica, uno de los principales problemas, es la cantidad de mutantes que que se producen, muchos de los cuales son equivalente entre sí. Reducir la cantidad de estos mutantes equivalentes permitiría contar con información menos redundante a la hora de analizar los resultados y, a su vez, contribuiría a mejorar los tiempos de los análisis. Para lograr este objetivo, habría que investigar heurísticas que permitan resolver el problema de los mutantes equivalentes que, como ya se mencionó, es un problema no decidible [BA82].

Otra de las alternativas a analizar para reducir la cantidad de mutantes es la de aplicar Selective Mutation [OLR+94], permitiendo correr los análisis seleccionando un subconjunto de operadores óptimos, es decir, operadores con un alto Mutation Score y que por lo tanto generen una mayor proporción de mutantes vivos con respecto al total. Para encontrar este subconjunto, debería hacerse un análisis estadístico de los operadores en diferentes modelos y, en base al resultado de este, incorporar en MuTalk la posibilidad de correr Mutation Testing con un grupo de operadores selectos. Pero además de considerar el Mutation Score, para analizar la eficiencia de los operadores se podría tener en cuenta la cantidad de mutantes terminados por generación de ciclos infinitos, ya que estos suelen insumir mayor tiempo en la ejecución del análisis.

Con respecto a la eficiencia en los tiempo del análisis, como ya se vió en [OPFK92], adaptando las herramientas para facilitar la ejecución concurrente se obtendrían mejoras de tiempo lineales en función de la cantidad de procesadores utilizados. Teniendo en cuenta que cada mutante es independiente de los demás y que los costos de comunicación serían bajos, la concurrencia resultaría factible y muy aprovechable.

También se podría investigar la posibilidad de reducir los tiempos de evaluación de los mutantes determinando el orden en que se corren los tests. Actualmente no existe ningún criterio que determine el orden en que estos se corren para la evaluación de un mutante, pero dado que esta evaluación se interrumpe ante el primer test que falle, se podrían analizar algunas posibilidades para determinar un

orden óptimo. Una posibilidad sería la de ordenar los tests de acuerdo al tiempo que toman y que los más cortos en duración sean corridos primero. Otra posibilidad, combinable con la anterior, sería la de registrar dinámicamente algún tipo de estadística que permita inferir la capacidad de matar mutantes de cada test, corriendo primero los que tengan mayor de esta capacidad.

Por otra parte, un problema que no se resolvió como parte de este trabajo es la posibilidad de aplicar Mutation Testing sobre cualquier modelo sin producirse inestabilidad o bloqueos de la imagen. Como se mencionó anteriormente, este problema puede ocurrir al aplicarse mutaciones que alteren el comportamiento de objetos que definen el meta-modelo o la máquina virtual, pero además con modelos de los cuales depende Mutation Testing (como las colecciones o las magnitudes de Smalltalk) o, incluso, el mismo modelo de Mutation Testing. Una solución a este problema podría ser clonar el modelo y los tests corriendo luego el análisis sobre los clones. Otra solución sería que de alguna manera existan distintos contextos de ejecución, que uno de ellos fuera exclusivo para la evaluación de los tests y que la mutación solo esté activada en ese contexto, de esta manera, se evitaría un comportamiento inadecuado en los otros contextos. Todo esto debería realizarse de forma transparente al usuario.

Por último, como vimos en la sección 5.1.2, en este trabajo se resolvió el problema de los posibles ciclos o recursiones infinitas en la evaluación de un mutante. Esta solución consiste en tomar el tiempo de correr todos los tests y luego utilizar este tiempo cómo referencia. De esta forma, se termina la evaluación de un mutante cuando el tiempo que lleva la misma supera en cierto factor fijo al de la evaluación de los tests. Si bien en la práctica esta solución resulta adecuada, podrían analizarse alternativas para minimizar aún más el tiempo de espera, haciendolo más acorde a cada uno de los tests. A priori, podría parecer más acertado tomar el tiempo de cada uno de los tests por separado, para luego utilizar este tiempo de referencia en la evaluación de cada uno de ellos. Sin embargo, el problema con esta solución alternativa es que el tiempo de correr un solo test suele ser muy corto, lo que implica una medición desacertada en muchos casos y como consecuencia, muchos mutantes terminados incorrectamente. Una alternativa a analizar sería la de tomar el máximo entre este tiempo y un tiempo fijo predeterminado, evitando terminar tantos mutantes antes de tiempo.

Referencias

[ABGJ02] Roger T. Alexander, James M. Bieman, Sudipto Ghosh, and Bixia Ji. Mutation of java objects, 2002.

[BA82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Inf.*, 18:31–45, 1982.

[Bec03] Kent Beck. *Test-Driven Development by Example.* Addison-Wesley, 2003.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. In *In Proceedings ECOOP '98, volume 1445 of LNCS*, pages 396–417. Springer-Verlag, 1998.

[Bou] Bouraqadi. Metaclasstalk: Reflection and meta-programming in smalltalk. `http://csl.ensm-douai.fr/MetaclassTalk/`.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 1972.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[JH09] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. Technical Report TR-09-06, September 2009.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA*, pages 147–155, 1987.

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, 2007.

[MO05a] Yu-Seung Ma and Jeff Offutt. Description of class mutation operators for java, 2005.

[MO05b] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for java, 2005.

[NBD+05] Oscar Nierstrasz, Re Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In *Proceedings of Software Composition 2005. LNCS*, pages 1–13, 2005.

[nFBD+08] Der Philosophisch naturwissenschaftlichen Fakultät, Der Universität Bern, Marcus Denker, Von Deutschland, Prof Dr, and O. Nierstrasz. Sub-method structural and behavioral reflection, 2008.

[OC94] A. Jefferson Offutt and W. Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[Off95] A. Jefferson Offutt. A practical system for mutation testing: Help for the common programmer, 1995.

[OLR+94] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5:99–118, 1994.

[OPFK92] A. Jefferson Offutt, Roy P. Pargas, Scott V. Fichter, and Prashant K. Khambekar. Mutation testing of software using a mimd computer. In *in 1992 International Conference on Parallel Processing*, pages 257–266, 1992.

[OU00] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal, 2000.

[Pha] Pharo. Pharo project. `http://www.pharo-project.org/`.

[RAD78] Frederick G. Sayward Richard A. DeMillo, Richard J. Lipton. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[SDZ09] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[sMOK05a] Yu seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15:97–133, 2005.

[sMOK05b] Yu seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15:97–133, 2005.

[Squ] Squeak. Squeak. `http://www.squeak.org/`.

[UOH93] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–148. Press, 1993.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. pages 227–242, 1987.

[Weba] MuJava Web. Mujava web. `http://cs.gmu.edu/~offutt/mujava/`.

[Webb] Mutation Testing Web. Mutation testing web. `http://www.mutationtest.net/twiki/bin/view/Resources/WebHome`.

A. Apendice

En la siguiente página puede verse el cuadro 13 con los resultados de los operadores detallados en cada paquete. De estos resultados se extrajeron los cuadros utilizados en la sección 5.3.1.

Operador \Paquete	Aconcagua			Magritte			VB-Regex			Network			Pier			Total		
	Mutantes	Vivos	M. Rate(%)	Mutantes	Vivos	M. Rate(%)	Mutantes	Vivos	M. Rate(%)	Mutantes	Vivos	M. Rate(%)	Mutantes	Vivos	M. Rate(%)	Mutantes	Vivos	M. Rate(%)
Replace and: with or:	34	8	76,47	44	12	72,73	19	4	78,95	15	5	66,67	54	16	70,37	151	40	73,51
Replace and: with nand:	34	0	100	44	2	95,45	19	1	94,74	15	2	86,67	54	5	90,74	151	8	94,7
Replace and: argument with [true]	34	11	67,65	44	19	56,82	19	5	73,68	15	6	60	54	25	53,7	151	60	60,26
Replace and: receiver with true	34	21	38,24	44	25	43,18	19	9	52,63	15	7	53,33	54	25	53,7	151	80	47,02
Replace and: with bEqv:	34	22	35,29	44	15	65,91	19	5	73,68	15	9	40	54	25	53,7	151	67	55,63
Replace and: with false	34	0	100	44	4	90,91	19	2	89,47	15	3	80	54	7	87,04	151	13	91,39
Remove Exception Handler Operator	2	1	50	4	0	100	0	0	-	1	1	0	1	0	100	7	1	85,71
Remove ∧	606	63	89,6	401	24	94,01	155	41	73,55	113	15	86,73	548	99	81,93	1710	227	86,73
Replace <= with >	8	0	100	2	0	100	0	0	-	0	0	-	7	0	100	17	0	100
Replace <= with <	8	5	37,5	2	0	100	0	0	-	0	0	-	7	3	57,14	17	8	52,94
Replace <= with =	8	2	75	2	0	100	0	0	-	0	0	-	7	0	100	17	2	88,24
Replace <= with true	8	4	50	2	1	50	0	0	-	0	0	-	7	0	100	17	5	70,59
Replace * with /	28	0	100	5	1	80	2	0	100	0	0	-	0	0	-	35	1	97,14
Replace + with −	24	0	100	1	0	100	3	2	33,33	25	1	96	35	1	97,14	63	3	95,24
Replace − with +	14	2	85,71	1	0	100	6	2	66,67	20	3	85	28	1	96,43	49	5	89,8
Replace / with *	27	2	92,59	0	0	-	0	0	-	0	0	-	0	0	-	27	2	92,59
Replace ifTrue: with ifFalse:	14	0	100	39	3	92,31	65	21	67,69	54	0	100	87	10	88,51	205	34	83,41
Replace ifTrue: receiver with false	14	2	85,71	39	18	53,85	65	47	27,69	54	14	74,07	87	19	78,16	205	86	58,05
Replace ifTrue: receiver with true	14	2	85,71	39	7	82,05	65	25	61,54	54	9	83,33	87	18	79,31	205	52	74,63
Replace ifTrue:ifFalse: receiver with false	33	13	60,61	6	2	66,67	43	22	48,84	28	7	75	30	10	66,67	112	47	58,04
Replace ifTrue:ifFalse: receiver with true	33	3	90,91	6	3	50	43	19	55,81	28	6	78,57	30	9	70	112	34	69,64
Replace isEmpty with notEmpty	9	0	100	14	0	100	12	7	41,67	18	0	100	21	4	80,95	56	11	80,36
Replace or: with and:	7	0	100	12	3	75	19	8	57,89	9	1	88,89	17	6	64,71	55	17	69,09
Replace or: argument with [false]	7	0	100	12	4	66,67	19	10	47,37	9	1	88,89	17	8	52,94	55	22	60
Replace or: receiver with false	7	2	71,43	12	6	50	19	10	47,37	9	5	44,44	17	8	52,94	55	26	52,73
Replace or: with bXor:	7	5	28,57	12	8	33,33	19	16	15,79	9	7	22,22	17	11	35,29	55	40	27,27
Replace or: with true	7	0	100	12	0	100	19	3	84,21	9	1	88,89	17	3	82,35	55	6	89,09
Replace reject: with select:	8	0	100	4	0	100	0	0	-	0	0	-	0	0	-	12	0	100
Replace reject: block with [:each | false]	8	2	75	4	0	100	0	0	-	0	0	-	0	0	-	12	2	83,33
Replace reject: block with [:each | true]	8	0	100	4	0	100	0	0	-	0	0	-	0	0	-	12	0	100
Replace select: with reject:	5	0	100	4	0	100	0	0	-	0	0	-	8	4	50	17	4	76,47
Replace select: block with [:each | false]	5	0	100	4	0	100	0	0	-	0	0	-	8	4	50	17	4	76,47
Replace select: block with [:each | true]	5	0	100	4	1	75	0	0	-	0	0	-	8	5	37,5	17	6	64,71
Replace do: block with [:each |]	13	2	84,62	9	2	77,78	2	1	50	5	1	80	45	13	71,11	69	18	73,91

Cuadro 13: Tabla de Resultados de Operadores