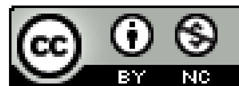
Mutación de Expresiones de Navegación para Testing y Reparación

por Simón Emmanuel Gutiérrez Brida

Presentado ante la Facultad de Matemática, Astronomía y Física
como parte de los requerimientos para la obtención del grado de
Doctor en Ciencias de la Computación de la
Universidad Nacional de Córdoba

Noviembre, 2018
@FaMAF - UNC 2018
Director: Nazareno Matías Aguirre



Mutación de Expresiones de Navegación para Testing y Reparación por Simón Emmanuel Gutiérrez Brida se distribuye bajo una Licencia Creative Commons Atribución-NoComercial 2.5 Argentina.

Resumen

Verificar que un sistema de *software* realiza correctamente las tareas para las cuales fue desarrollado es una de las actividades de mayor importancia en Ingeniería de Software, y concentra un significativo esfuerzo de investigación en esta área. El *testing*, el cual consiste en ejecutar un programa a evaluar en un conjunto de escenarios particulares y contrastar el comportamiento esperado del programa con el efectivamente obtenido, es una de las técnicas más utilizadas como forma de comprobación del correcto comportamiento del software.

Dada la inherente incompletitud de *testing*, resulta necesario realizar una selección adecuada de los escenarios bajos los cuales realizar la evaluación. Claramente, cómo esta evaluación es realizada va a afectar la confianza que genere el proceso de *testing*, cuando la ejecución real del software coincida con la esperada, y las chances de detectar defectos en el *software* durante este proceso. Concretamente, se espera que el conjunto de escenarios seleccionado para el proceso de *testing* sea un buen representante de todos los posibles escenarios de ejecución del software en cuestión. Los criterios de testing permiten medir la calidad de un conjunto de tests generando objetivos a ser cubiertos, y evaluando cuántos de estos son satisfechos (cubiertos) por los tests. *Mutation testing* es uno de estos criterios, y consiste en inyectar fallas artificiales en el software bajo evaluación, y evaluar la capacidad de detección, por parte de los tests, de estas fallas.

Las fallas generadas por mutation testing se basan en operadores de mutación. Estos operadores deben ser buenos representantes de fallas reales, y tradicionalmente involucran cambios simples, tales como el reemplazo de operadores aritméticos y relacionales. Algunos estudios recientes muestran sin embargo que algunas clases de fallas no se ven representadas por los operadores de mutación tradicionales, motivando así la introducción de nuevos operadores.

En esta tesis presentaremos un nuevo operador de mutación, que aplica a *expresiones de navegación*, un tipo de expresiones ampliamente utilizadas en programas orientados a objetos, y que no se ven afectadas por los operadores de mutación clásicos. Daremos una definición precisa del operador, y evaluaremos su aplicación tanto en el contexto de testing (mutation testing), como en el contexto de reparación de programas.

Palabras Clave: Ingeniería de Software, Testing, Reparación de Programas, Mutación, Operadores de Mutación, Expresiones de Navegación.

1

Introducción

En la actualidad, el software se encuentra en todos los aspectos de la vida diaria, y en general la expectativa del usuario es confiar en que el mismo simplemente funciona. Nadie espera que el automóvil no arranque por un error en el software embebido que utiliza, o que el cajero automático de un banco no retorne billetes al realizar una extracción válida, o que el reloj del teléfono celular marque un horario incorrecto, etc. Sin embargo, son innumerables los ejemplos de errores de software, que causan comportamientos indebidos del mismo. Algunos de estos errores pueden ser catastróficos. Algunos ejemplos son los siguientes. En 2015 se descubrió un bug en el modelo Boeing 787 Dreamliner que podía causar el apagado de todos los generadores eléctricos del avión, si las unidades de control de los generadores permanecían encendidas por más de 248 días [(FAA), 2015]. En los años 80, un error en el código del controlador para la máquina de terapia de radiación *Therac-25*, causó la muerte de varios pacientes al administrar cantidades excesivas de radiación beta [Leveson and Turner, 1993]. Otros ejemplos más recientes (y menos críticos) incluyen el caso de un bug de WhatsApp, descubierto en 2018, que causaba que la aplicación, y en muchos casos el dispositivo, dejara de responder, si se recibía un mensaje en Unicode, conteniendo una secuencia que repetía cierto caracter especial (el caracter especial que especifica la dirección de renderizado del texto).

Claramente, suponer que el software simplemente funciona, es un error. Los errores en un programa pueden tener consecuencias que van desde una molestia menor al usuario, hasta la pérdida de vidas. Esto muestra la enorme relevancia del problema de garantizar la corrección de un programa, es decir, la comprobación de que para todo escenario de uso, incluyendo las entradas directas e indirectas del software, el programa se comporta como se espera, o equivalentemente, que arroja el resultado esperado. Uno de los enfoques tradicionales y de mayor adopción en la práctica, para analizar la corrección de un programa, es el *testing*, que consiste en evaluar el comportamiento del programa en cuestión en un conjunto específico de escenarios de ejecución. Dado que la comprobación exhaustiva de todos los escenarios de ejecución es en general inviable salvo para programas muy simples, en la práctica sólo un subconjunto de estos escenarios puede ser evaluado. Cuántos y cuáles de todos los escenarios se seleccionan está directamente relacionado con la confianza que

brindará el proceso de testing de que el software funciona correctamente, cuando este proceso no encuentre fallas.

Un conjunto de tests (los escenarios de ejecución del software bajo análisis) debe ser necesariamente finito, lo cual implica que en general no es posible en la práctica evaluar exhaustivamente una pieza de software en todos los posibles escenarios de ejecución. Más aún, la ejecución del software en los escenarios elegidos debe insumir recursos razonables; esto por supuesto va a depender del contexto y los recursos disponibles, pero en líneas general es esperable que la ejecución de los casos de tests sea *eficiente*, especialmente aquellos que el desarrollador utiliza como apoyo constante a sus actividades de programación.

Dado que la finalidad del testing es evaluar un programa en un subconjunto de todos sus escenarios posibles, como forma aproximada de estimar (a través de una generalización) su correcto comportamiento en el universo de todos los escenarios de ejecución, es necesario contar con un criterio para evaluar la calidad del conjunto elegido, es decir, evaluar cuán bien un conjunto de escenarios representa el universo de éstos. Intuitivamente, un buen conjunto de casos de tests, o *test suite*, es aquel que tiene una alta capacidad de detectar fallas: si existe un bug en el programa, algún test en la suite es capaz de detectarlo. Sin embargo, esta intuición, como criterio para evaluar cuán buena es una suite, es viable sólo si uno conoce de antemano los bugs del programa bajo análisis. Resulta necesario entonces utilizar criterios indirectos, para medir la calidad de suites de tests. Estos criterios definen en general metas o requisitos a cubrir por los tests, a partir de los cuales se puede *medir* cuántos son efectivamente cubiertos. Estos criterios de evaluación se suelen dividir en dos categorías principales: *caja blanca* (white box), cuando las metas a cubrir se basan en la estructura del programa, y *caja negra* (black box), cuando las metas se basan en las especificaciones. Algunos ejemplos típicos de criterios de caja blanca son la *cobertura de sentencias*, que exige ejecutar a través de la suite todas las sentencias en un programa, y *cobertura de ramas*, que exige ejecutar todas las alternativas para las sentencias de control de flujo. Determinar clases de equivalencia para las entradas del programa basado en su especificación, y tener al menos un escenario por cada una de estas clases, es un ejemplo de un criterio de caja negra (denominado *particionado en clases de equivalencia*).

Un ejemplo de cómo los criterios de evaluación de test suites intentan dar una medida indirecta de la capacidad de las mismas en detectar fallas potenciales se puede apreciar en la cobertura de sentencias. Este criterio impone, como metas a cubrir, la ejecución de cada una de las sentencias del programa bajo evaluación. La intuición de este criterio es bastante simple: lo mínimo indispensable para descubrir una falla, es ejecutar la sentencia o sentencias donde se encuentra el defecto asociado a la misma. Es sin embargo fácil encontrar ejemplos de fallas

simples donde este criterio da una evaluación positiva a un conjunto de tests, y aún así este conjunto sea incapaz de detectar fallas. Varios de estos ejemplos pueden ser detectados por una test suite que tenga una buena evaluación con respecto al criterio de cobertura de ramas. Esto lleva a que evidentemente hay criterios que generan más confianza que otros, o equivalentemente, que imponen requisitos más fuertes a las test suites.

Retomando la intuición inicial de evaluar una test suite con respecto a su habilidad de detectar fallas, un criterio razonable y factible es el de utilizar fallas artificiales, es decir, fallas conocidas, inyectadas en el programa, para evaluar si las mismas son detectadas por el conjunto de tests bajo evaluación. Esta idea se basa en el razonamiento de que los programadores suelen crear programas, que cuando tienen fallas, no están lejos de una solución correcta correspondiente [DeMillo et al., 1978], y por lo tanto pequeños cambios sintácticos en un programa deberían emular los errores que se suelen cometer en su desarrollo. *Mutation testing* es un criterio de testing basado en esta idea. El mismo se basa en generar copias del programa original, donde cada una, denominada *mutante*, tiene inyectada una falla artificial en forma de un cambio sintáctico simple, denominado *mutación*. Por cada mutante se ejecutan los tests, y si al menos uno de éstos falla, entonces se marca al mutante como detectado. El valor asociado a este criterio, denominado *mutation score*, es la relación entre mutantes detectados y todos los generados. Las fallas artificiales generadas por mutation testing están basadas en distintos operadores de mutación, que definen familias de cambios sintácticos similares. Por ejemplo, dada una expresión relacional binaria, cambiar el operador de relación por cada uno de los existentes en el lenguaje en el cual el programa está desarrollado, es un operador de mutación (es decir, cambiar por ejemplo un operador como $<$ por todas las alternativas de comparación, $==$, $\leq$, $\geq$, etc., generando por cada una un mutante diferente). Numerosos estudios han intentado responder si existe una correlación (acoplamiento) entre la capacidad de una test suite en detectar fallas artificiales, utilizadas en mutation testing, y la capacidad de hacerlo para fallas reales [Offutt, 1989; Offutt, 1992; Andrews et al., 2005a; Just et al., 2014]. Estos estudios han encontrado que de hecho existe una correlación, aunque siempre acotada a casos de estudio particulares. Aún teniendo en cuenta estos resultados, el rendimiento del criterio está directamente relacionado a los operadores de mutación utilizados, y las fallas artificiales que éstos generan. Por un lado, la cantidad de mutantes impacta en los recursos necesarios para ejecutar el análisis, ya que en el peor caso es necesario ejecutar todos los tests para cada uno de los mutantes. Por otro lado, existen fallas artificiales que son poco deseables para los propósitos de mutation testing, o bien porque son trivialmente detectables (por ejemplo, en ciertos lenguajes, modificar el índice en el acceso a un arreglo a un valor negativo) o porque dan lugar a programas semánticamente equivalentes al programa original (por

ejemplo, realizar un incremento *a posteriori*, o post-incremento, en una variable local en la sentencia de retorno de un método). En el primer caso, ejecutar la sentencia va a causar un error, y cualquier test que lo haga va a detectar dicho mutante; este tipo de fallas triviales va a aumentar el mutation score sin realmente implicar un aumento en la calidad de la test suite. En el segundo caso, no existe ningún escenario para el cual el comportamiento del mutante difiera del comportamiento observable del programa original, causando un decremento en el valor del mutation score, sin significar un empeoramiento en la test suite. Finalmente, así como las fallas artificiales pueden estar acopladas a fallas reales, el mismo fenómeno puede ocurrir entre los mutantes. Más precisamente, detectar ciertos mutantes va a implicar que otros sean también detectados. Este tipo de acoplamiento entre mutantes lleva a un aumento del mutation score, que si bien puede estar asociado a fallas artificiales no triviales, al ser éstas similares, no implican que la test suite sea capaz de detectar una mayor variedad de fallas reales.

Existen varios estudios que intentan atacar los problemas mencionados anteriormente, incluyendo aquellos que lo hacen seleccionando un conjunto *suficiente* de operadores de mutación [Offutt et al., 1996; Namin et al., 2008]; otros que buscan métodos para detectar mutantes equivalentes [Grün et al., 2009; Schuler and Zeller, 2010; Just et al., 2013], y otros que utilizan una combinación de mutaciones para generar mutantes sutiles bajo el razonamiento de que éstos deberían representar fallas más difíciles de detectar [Jia and Harman, 2008; Jia and Harman, 2009a; Harman et al., 2011]. Un estudio particularmente interesante, y relevante para los objetivos de esta tesis, es el que se presenta en [Just et al., 2014], cuyas conclusiones incluyen el hecho de que existen fallas reales que requieren mejorar operadores de mutación existentes, o desarrollar operadores nuevos (además de mostrar que mutation testing es un criterio que mantiene una mayor correlación con la detección de bugs reales que los criterios tradicionales de caja blanca). Esto expone la necesidad de mejorar operadores de mutación o desarrollar nuevos operadores, para poder representar a estos tipos de fallas reales no representadas actualmente por operadores existentes.

Además de la aplicación tradicional de operadores de mutación para mutation testing, recientemente ha surgido un área dentro de la Ingeniería de Software que le da utilidad a estos operadores, invirtiendo su aplicación respecto a mutation testing: dado un programa incorrecto, es decir, que no cumple con la especificación asociada al mismo, se aplican mutaciones al programa con la finalidad de intentar *repararlo*. Esta área, denominada *reparación automática de programas* [Harman, 2010], encuentra una nueva aplicación a los operadores de mutación, y de hecho da lugar a nuevas formas de mutación, que podríamos describir como de “granularidad gruesa”, que alteran la estructura de programa

de manera más drástica, replicando bloques de código, eliminando e intercambiando bloques de código, etc [Le Goues et al., 2012]. De la misma forma que en el contexto de mutation testing los operadores de mutación especificamente utilizados impactan en la calidad de la evaluación que mutation testing brinda, en el contexto de reparación de programas los operadores utilizados afectan tanto la eficiencia del proceso de reparación, como la efectividad del mismo. A diferencia de mutation testing, en reparación de programas los operadores de mutación suelen requerir múltiples aplicaciones (reparar un programa suele requerir múltiples alteraciones al programa original), y por lo tanto contar con muchos operadores de mutación da lugar a una explosión en las reparaciones "candidatas"; por otra parte, contar con pocos operadores de mutación disminuye las chances de reparar programas [Le Goues et al., 2012]. Nuevamente, existen diversas formas de enfrentar estos problemas, desde los que consideran *sólo* mutaciones de granularidad gruesa [Le Goues et al., 2012], o producen mutaciones imitando reparaciones realizadas por desarrolladores [Kim et al., 2013], hasta los que sustituyen la exploración exhaustiva (acotada) de reparaciones candidatas por búsquedas guiadas [Le Goues et al., 2012; Staber et al., 2005; Arcuri and Yao, 2008]. Destacamos cómo, en el contexto de reparación de programas, surge nuevamente el problema del diseño y la construcción de operadores de mutación.

1.1 MOTIVACIÓN Y OBJETIVOS

La relación entre las fallas artificiales generadas por los operadores de mutación utilizados en mutation testing, y las fallas reales cuya detección representa el objetivo final de una test suite, ha sido estudiado con resultados positivos hacia la existencia de esta relación [Daran and Thévenod-Fosse, 1996; Andrews et al., 2005b; Namin and Kakarla, 2011] aunque es en el trabajo de Just et al [Just et al., 2014] en donde no solamente consideran casos de estudios de mayor tamaño (entre otras consideraciones que los trabajos previos no contemplaban) sinó también un análisis más profundo que resulta en grupos de fallas reales para los cuales finalmente reportan si éstos se encuentran acopladas a mutantes generados por operadores en el conjunto suficiente, o si es necesario mejorar o definir nuevos operadores que los representen. Este trabajo nos significa (como veremos más adelante) una de las motivaciones para mostrar la necesidad de un operador de mutación orientado hacia expresiones de navegación.

En la actualidad, los lenguajes orientados a objetos, o que contienen características de orientación a objetos, tienen una significativa relevancia dentro del espectro de los lenguajes de programación más utilizados. Un tipo de expresión comúnmente encontrado en este tipo de lenguajes son las *expresiones de navegación*. Éstas se forman al acceder a miembros de clases (campos y métodos)

```
public class Queue {

  private Node front;
  private Node last;

  ...
  public void dequeue() {
    this.front = this.front.next;
  }

  public int size() {
    // computes number of nodes in the
    // underlying list
    ...
  }
  ...
}
```

Figura 1: Una implementación de colas basada en referencias.

mediante un operador de acceso que suele seguir la notación punto. Desde la perspectiva de mutation testing, cabe resaltar que incluso con la popularidad de los lenguajes orientados a objetos, y la importancia de las expresiones de navegación en estos lenguajes, ningún operador de mutación actual, en particular ninguno de aquellos que pertenecen al grupo de *operadores suficientes*, generan mutaciones para este tipo de expresiones. Por supuesto esto podría deberse a la suficiencia de los operadores de mutación tradicionales, y la irrelevancia de operadores de mutación para expresiones de navegación, a pesar de la importancia de estas expresiones en programas orientados a objetos. Veremos sin embargo, a través de un simple ejemplo, la necesidad de agregar operadores que generen mutaciones para expresiones de navegación.

Consideremos una implementación de una cola sobre una lista simplemente encadenada acíclica, con una referencia al primer nodo, *front*, y otra al último, *last*, y un método *dequeue()*, tal como se muestra en la Figura 1. Notemos que esta implementación es defectuosa, dado que no trata adecuadamente los casos en los cuales la cola está vacía o consta de un único elemento. Desde el punto de vista de los criterios clásicos de caja blanca, y dado que la implementación de este método no contiene bifurcaciones en su flujo de control, cualquier test que haga una llamada a este método va a lograr una cobertura estructural (ramas, por ejemplo) del 100 %. En lo que respecta a mutation testing, cabe resaltar

que ninguna de las herramientas modernas de mutation testing, por ejemplo, *PITest*, *Major*, o *μJava*, generan mutaciones para el código en *dequeue()*. Luego, resulta trivial para cualquier test lograr una “cobertura” perfecta de mutación. Concretamente, tener un único test para este método, como el siguiente:

```
@Test
public void dequeueTest() {
  Queue q = new Queue();
  q.enqueue(1);
  int size = q.size();
  q.dequeue();
  assertEquals(size - 1, q.size());
}
```

resulta suficiente para lograr una cobertura (ramas) del 100 %, y cubrir de manera trivial todos los objetivos definidos por mutation testing.

Evidentemente, el problema en este caso se debe a que, en el contexto de mutation testing, ningún operador de mutación es capaz de mutar el cuerpo de *dequeue()*. Por lo tanto, este criterio falla en imponer cualquier requisito (metas de cobertura) a la test suite. Si las expresiones de este tipo fueran raramente encontradas en programas actuales, podríamos simplemente considerar a este problema poco importante, e ignorarlo. Pero las expresiones de navegación, del estilo de las que constituyen el cuerpo de *dequeue()*, son muy comunes en programas que utilizan lenguajes orientados a objetos. Es más, algunos patrones de diseño, populares en lenguajes orientados a objetos, como *Builder*, o *Fluent Interfaces*, hacen uso sustancial de expresiones de navegación.

Consideremos por ejemplo el patrón *Builder*. Este patrón es utilizado para construir instancias de objetos que suelen contener una gran cantidad de atributos opcionales, sin la necesidad de definir en la clase correspondiente un alto número de constructores distintos. Más aún, este patrón permite también que la construcción de un objeto complejo sea legible, en el código fuente. Un ejemplo de una biblioteca que utiliza este patrón es *Apache Commons CLI*, una biblioteca para manejar los argumentos de entrada de una aplicación con interfaz de línea de comandos. Un ejemplo del uso de ésta se muestra a continuación:

```
Option inputOption = Option.builder("V")
  .longOpt("Value")
  .desc("The input value")
  .hasArg(true)
  .numberOfArgs(1)
```

```
    .type(Integer.class)
    .required(true)
    .build();
```

Este ejemplo muestra la creación de un argumento para un programa que utiliza `-V` o `-Value` para hacer referencia al argumento; define una descripción del mismo (`The input value`), especifica que el argumento requiere un único valor asociado, de tipo entero, y que el argumento es obligatorio. Un ejemplo de uso de esa opción sería:

```
miPrograma --Value 42
```

Otro patrón que realiza una utilización extensiva de expresiones de navegación es *Fluent Interfaces*, o interfaces fluentes. El mismo permite organizar programas alrededor de la utilización de una jerarquía de clases (asociada a los lenguajes orientados a objetos), para definir una *gramática*, que permita caracterizar un lenguaje de dominio específico. Veamos aquí un ejemplo, con una biblioteca de SQL en Java:

```
dbconnector.from("Songs").select().where()
.attribute("year").ge().value(1980)
.and()
.attribute("year").lt().value(1990)
.execute();
```

En este ejemplo se muestra el uso de *method chaining* para escribir una consulta a la tabla *Songs*, de la cual se van a seleccionar aquellas entradas en las cuales el atributo *year* sea mayor o igual a 1980 y menor que 1990.

Un punto importante a destacar en relación al ejemplo anterior es que si bien existen operadores relacionales que pueden modificar un "mayor o igual a" por otro operador, el caso de `attribute("year").ge().value(1980)`, que intuitivamente representa `year >= 1980`, no puede ser mutado.

El conjunto de *operadores suficientes* de mutación ha permanecido prácticamente inalterado desde hace años, aún cuando el primer trabajo que ofrece un conjunto de operadores formalmente definido lo hace para el lenguaje Fortran [Offutt and King, 1987; King and Offutt, 1991]. Mutation testing ha sido aplicado y evaluado en varios lenguajes, incluyendo aquellos considerados orientados a objetos como Ada y Java, por dar un par de ejemplos. Sin embargo los operadores de mutación específicos para lenguajes orientados a objetos sólo afectan, en general, jerarquías de clases, visibilidad, y sobre-escritura.

Un problema similar, en relación a las expresiones de navegación, se observa en el contexto de reparación de programas. Consideremos nuevamente la implementación de colas sobre listas enlazadas acíclicas, y el código del método *enqueue()* que se muestra a continuación:

```
public void enqueue(int elem) {
        QueueNode newNode = new QueueNode(elem);
        if (front == null) {
            front = newNode;
            last = newNode;
        } else {
            front.next = newNode;
            last = newNode;
        }
        size++;
}
```

Esta implementación es errónea dado que, cuando la cola no está vacía, inserta el nuevo elemento como el siguiente del comienzo, en lugar de insertar al final, como exige la política de visita de elementos en una cola. El error y su reparación son simples: la expresión `front.next = newNode` debería en realidad ser `last.next = newNode`. Sin embargo, herramientas de reparación automática como GenProg son incapaces de reparar este código, pues sus mutaciones de granularidad gruesa no consiguen reproducir el comportamiento esperado para *enqueue()*.

Esto nos lleva a nuestros objetivos:

Analizar las características que un operador de mutación debe poseer Diseñar un operador de mutación no es una tarea trivial. Tal como mencionamos anteriormente, mutantes equivalentes al programa original son indeseados, así como aquellos que son trivialmente detectados; los mutantes generados deben representar (o estar acoplados a) fallas reales. La aplicación de operadores de mutación para reparación introduce características adicionales sobre los operadores de mutación, algunas contrapuestas a las de mutación para testing. De esta manera, nuestro primer objetivo es caracterizar las propiedades que un operador de mutación debe poseer, y definir éstas de manera precisa, acompañadas de mecanismos para su evaluación objetiva.

Definir un operador de mutación para expresiones de navegación Teniendo en cuenta las características surgidas en el punto anterior, y basándose en la motivación para diseñar nuevo[s] operadores de mutación para expresiones de navegación, nuestro segundo objetivo es dar una definición

de un nuevo operador que satisfaga esta necesidad, y que cumpla con las características identificadas. Más aún, en la medida de lo posible, esta definición debe ser agnóstica al lenguaje de programación utilizado para implementarlo.

Implementar el operador de mutación para expresiones de navegación
Los operadores existentes actualmente están definidos por reglas de transformación muy simples, incluso pudiendo ser implementados usando expresiones regulares. Modificar una expresión de navegación requiere información de tipos y análisis de alcanzabilidad para determinar las expresiones disponibles al modificar una existente. Nuestro tercer objetivo es entonces encontrar una herramienta de mutación que permita realizar el análisis necesario para generar estas mutaciones; elegir un lenguaje sobre el cual y para el cual implementar este operador; e implementar tanto el operador como herramientas u extensiones que permitan realizar los análisis apropiados para las evaluaciones necesarias a nuestro operador.

Aplicaciones e impacto del operador de mutación en testing
Incluso si es cierto que existe una necesidad por la generación de mutantes para expresiones de navegación, si el o los operadores que los generan no cumplen con las propiedades anteriormente descriptas, entonces su utilidad se ve muy reducida. Por eso, otro de nuestros objetivos es evaluar el desempeño de nuestro operador para analizar su utilidad e impacto de su incorporación como operador de mutación, en mutation testing.

Aplicaciones del operador de mutación en reparación
Si bien el énfasis principal en esta tesis estará alrededor de la introducción de operadores de mutación para testing, analizaremos también la aplicación de operadores de mutación de granularidad fina en el contexto de reparación automática de programas (en contraposición con operadores de mutación tradicionales en reparación, que optan por una granularidad gruesa). En particular, analizaremos errores comunes en programas orientados a objetos, que tienen que ver con el uso de expresiones de navegación, y cómo el operador de mutación a introducir contribuye a su reparabilidad. En este proceso, analizaremos y discutiremos el rol de las especificaciones en la reparación automática de programas, y las limitaciones del uso de tests para suplantarlas.

1.2 CONTRIBUCIONES

La definición e implementación de un operador de mutación, altamente configurable, para expresiones de navegación, dentro del contexto de mutation testing

pero también aplicable a reparación de programas, es la principal contribución de este trabajo. Dado que la mutación de expresiones de navegación generaría en principio una cantidad inmanejable de mutantes, es necesario encontrar y definir el tipo de fallas que se desean generar (o alternativamente, el tipo de reparaciones candidatas, para el contexto de reparación). Aún cuando la generación de mutantes para expresiones de navegación pudiera representar un conjunto de fallas no representadas por operadores de mutación existentes, no es necesariamente beneficioso generar tales mutaciones: en el caso particular de testing de mutación, debemos comprobar que la generación de tales mutaciones contribuye al análisis, más precisamente, no afecta negativamente el análisis de mutación produciendo mutantes débiles (fáciles de matar y consecuentemente incrementando artificialmente el mutation score), ni mutantes "acoplados" a otros mutantes tradicionales (es decir, que se vean representados por otras mutaciones), ni un número alto de mutantes equivalentes (imposibles de "matar", dado que equivalen al programa original, y que por lo tanto disminuyen artificialmente el mutation score). De la misma manera, deberemos comprobar, en el contexto de reparación automática de programas, que la incorporación de mutaciones para expresiones de navegación incrementa la "reparabilidad", es decir, ayuda a reparar programas que, sin estas mutaciones, no pueden ser reparados o sus reparaciones son más difíciles de construir o encontrar. Específicamente, en esta tesis presentaremos las siguientes contribuciones principales:

1. Introduciremos un operador de mutación para expresiones de navegación. Este operador estará definido de manera precisa, y será configurable en una variedad de aspectos. Más aún, si bien la implementación que se ofrece en esta tesis se provee para un lenguaje particular, la definición del operador permite su implementación para cualquier lenguaje con expresiones de navegación. Esto es particularmente relevante para el caso de lenguajes de modelado, como Alloy [Jackson, 2006], en cuyo contexto se ha identificado la necesidad de "mutar" expresiones, como en [Wang et al., 2018a], que propone una herramienta de reparación de modelos escritos en este lenguaje, y [Wang et al., 2018b], que expone la necesidad de la generación automática de expresiones en álgebra relacional para distintos propósitos, entre ellos reparación automática y ayudas al desarrollador (tooltips automáticos).

2. Implementaremos el operador para expresiones de navegación, enfocándonos en un lenguaje de programación particular, *Java*. Elegimos este lenguaje porque lo consideramos un buen representante de lenguajes orientados a objetos. La implementación estará basada en una versión modificada de μ*Java* [Ma et al., 2005], que denominamos μ*Java++*, y que nos permite realizar análisis de tipos y alcanzabilidad que otras herra-

mientas no permiten. Como veremos más adelante, generar mutaciones para expresiones de navegación requiere una cantidad significativa de "fine tuning", para maximizar las fallas reales representadas, minimizando características no deseadas, entre ellas la explosión en el número de mutantes generados. Por esto, nuestra implementación del operador de mutación funciona como un "meta-operador", el cual, mediante una amplia gama de configuraciones, puede verse como una familia de operadores de mutación (uno por cada configuración particular).

3. Una evaluación detallada del impacto de la incorporación de mutaciones para expresiones de navegación, en el contexto de mutation testing. Esta evaluación involucrará una serie de implementaciones orientadas a objetos, test suites variadas para las mismas, sobre las cuales se analizarán distintas métricas (que forman parte de nuestra implementación): *dureza* (toughness), cuyo propósito es medir cuán difícil de matar resulta un mutante, a través de la medición del número de tests a los cuales fue capaz de sobrevivir un mutante antes de ser detectado (este valor es relevante para realizar análisis sobre la dificultad de matar los mutantes generados por un operador); *subsumción* (subsumption) de mutantes, que permite poner en evidencia, para un programa particular y con un conjunto de tests específico, qué mutantes resultan "redundantes"; entre otras.

4. Una evaluación de la viabilidad de la incorporación de mutaciones para expresiones de navegación, en conjunto con otras mutaciones de "granularidad fina", en el contexto de reparación automática de programas. En particular, evaluaremos la reparabilidad de programas a través de estas mutaciones, y la viabilidad en términos de eficiencia, de la reparación con tales mutaciones. En relación a esta contribución, analizaremos también la importancia de las especificaciones en la reparación automática de programas, y las limitaciones del uso de tests para suplantarlas en este contexto.

1.3 ORGANIZACIÓN

El resto de esta tesis está organizada de la siguiente manera. En el Capítulo 2 daremos una introducción a *testing* como técnica para comprobar parcialmente el correcto funcionamiento de un programa, la automatización de esta técnica, y una breve introducción a criterios de cobertura. En el Capítulo 3 presentaremos *mutation testing*, como un criterio de testing. En este capítulo mostraremos cuáles son las propiedades que afectan a la calidad de este criterio para evaluar un conjunto de tests, y algunos mecanismos para evaluar estas propiedades. La

presentación y definición de *prvo* como un operador de mutación para expresiones de navegación se introducirá en el Capítulo 4, así como la descripción de las fallas reales que están relacionadas a este operador. En el Capítulo 5 se presenta una versión implementada de *prvo*, considerando restricciones a su definición general provista en el capítulo anterior, y teniendo en cuenta las propiedades que identificamos para el diseño de operadores de mutación. Describiremos también la plataforma sobre la que *prvo* es implementado, y las características principales de la misma, principalmente el análisis dinámico de subsumción de mutantes, el cual será parte de nuestra evaluación. La evaluación en el contexto de mutation testing, se realiza en el Capítulo 6. Introduciremos brevemente el área de reparación automática de programas en el Capítulo 7, en conjunto con una descripción de la herramienta que utilizaremos para poner a prueba a *prvo*, el operador de mutación introducido en esta tesis, en dicho contexto. Las conclusiones se presentan en el Capítulo 8, y trabajos futuros son descriptos en el Capítulo 9.

2

Testing

Garantizar que un programa realiza de manera correcta las tareas para las cuales fue desarrollado se encuentra dentro de los problemas más desafiantes y uno de los más importantes temas de investigación en el contexto de la Ingeniería de Software [Ghezzi et al., 1991; Pressman, 2001; Jalote, 2005]. Es un tema central en la calidad de software, que demanda una cantidad significativa de recursos [Jalote, 1997], y por lo tanto tiene impacto sustancial en el costo de producción de software.

La introducción de defectos puede ocurrir en cualquier etapa del desarrollo del software. El costo de resolver los problemas asociados a corregir estas deficiencias, sin embargo, cambia sutancialmente de acuerdo a la etapa en la cual fue introducido y cuándo fue detectado: detectar un error introducido en la etapa de requisitos (por ejemplo, comprensión errónea de algún aspecto del problema a resolver) en la etapa de testing puede costar hasta 100 veces más que hacerlo durante la etapa de requisitos, donde fue introducido [Jalote, 1997]. De la misma manera, las técnicas para la detección de defectos de software cambian, de acuerdo al tipo de defecto, y a la etapa del proceso de desarrollo en la que se aplican. El *testing*, que en esencia consiste en ejecutar un programa bajo un conjunto específico de escenarios, contrastando el comportamiento actual con el esperado [Ammann and Offutt, 2016], es el enfoque más comúnmente usado para la detección de defectos de software, que dada su naturaleza, se aplica luego de la implementación de funcionalidad a evaluar. A pesar de sus conocidas limitaciones, que E. W. Dijkstra resumió notablemente en su conocida frase *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [Dijkstra, 1972], testing es la técnica de mayor aplicación, en la práctica, para brindar garantías (parciales) de calidad de software. La limitación central del testing está asociada al hecho de que en general es inviable, o imposible, ejecutar de manera exhaustiva un programa bajo todos sus posibles escenarios de ejecución (es decir, considerando todas sus entradas, directas e indirectas). Por esta razón, es necesario seleccionar una muestra, un subconjunto de todos los escenarios posibles, sobre los cuales se realizará la evaluación del comportamiento del software. Claramente, el testing, como técnica de verificación, es una técnica necesariamente incompleta, aunque goza de una simplicidad y escalabilidad

```
1   la entrada a evaluar es "(1 + 3)^2"
2   evaluar Calculator#evaluate con la entrada anterior
3   el resultado obtenido debe ser 16
```

Figura 2: Ejemplo de un test

que otros enfoques de verificación, especialmente aquellos basados en análisis estático, tienen dificultades de alcanzar.

Un *test* consta básicamente de los siguientes pasos:

1. Preparación (*Arrange*), define los datos necesarios para el escenario bajo el cual se va a ejecutar el test. A modo de ejemplo, consideremos el test de la Figura 2. La primera línea de este test, en la cual se define el escenario sobre el cual se va a evaluar la implementación de una calculadora, corresponde al *arrange*.

2. Ejecución (*Act*), es la ejecución del programa a evaluar. Generalmente incluye la obtención del resultado de dicha ejecución, para su posterior contrastación con el resultado esperado. La línea 2 de la Figura 2 ejecuta una funcionalidad de la calculadora, a través del método `Calculator#evaluate(String)`, para la expresión definida en el arrange; esta línea corresponde al *act* del test.

3. Evaluación (*Assert*), consiste en la verificación de que el resultado obtenido, ya sea un valor de salida o un estado del programa, se corresponde con el esperado. La última línea de la Figura 2 evalúa que la ejecución del método `Calculator#evaluate(String)`, para la expresión definida en el arrange, da como resultado `16`; esta línea corresponde al *assert* del test.

Evidentemente, cómo se elige la muestra de entradas que será utilizada para testing afecta significativamente la habilidad del proceso de testing en la detección de fallas en el software. Recordemos que el programa bajo análisis se evaluará sobre un conjunto de entradas, con la intención de verificar que el comportamiento real del software coincide con el esperado en estos casos, y a partir de este resultado “generalizar” el resultado a todos los posibles escenarios de ejecución. Dado que la selección de los escenarios para testing es crucial para este proceso, se necesita algún tipo de criterio, denominado criterio de testing, que asista en la selección de los escenarios. Los criterios de testing ayudan a seleccionar escenarios, y al mismo tiempo sirven como *métricas* de la calidad de conjuntos de tests.

Los criterios de testing típicamente se clasifican en *caja blanca* (white box), si tienen en cuenta la estructura del código bajo análisis, o *caja negra* (black box), si en cambio se concentran en la especificación del programa bajo prueba. Los criterios de testing, independientemente de la clase a la que pertenezcan, definen, directa o indirectamente, una métrica, que permite evaluar la calidad de una test suite. Esta medida se puede entender como una métrica de cuán exhaustivamente evalúa una test suite el comportamiento del software, o equivalentemente, cuán probable es que existan defectos en el software, que pasen desapercibidos a la suite. Las medidas son, por supuesto, indirectas.

2.1 AUTOMATIZACIÓN DE LA EJECUCIÓN DE TESTS

Si bien la definición de test dada anteriormente no menciona ejecutabilidad, sino que simplemente se limita a describir los pasos o etapas que constituyen un test, es directo suponer que estos pasos son, con excepción quizás del *assert*, evidentemente implementables. Decimos que la implementación de la etapa de *assert* es menos evidente porque este paso es en muchos casos, cuando se llevan adelante procesos ad hoc de testing, realizado manualmente: es el desarrollador o usuario quien contrasta el resultado real con el resultado esperado. Sin embargo, muchas características del resultado esperado de un programa son, en muchos casos, verificables de manera directa, y por lo tanto también lo es su automatización.

Existen numerosas bibliotecas disponibles para automatizar el proceso de ejecución de tests. Una de estas, que goza de enorme popularidad, es la biblioteca *JUnit*, para tests de unidad en Java. En la Figura 3 se muestra el ejemplo anterior implementado en *Java* utilizando la biblioteca *JUnit*.

```
1   String expression = "(1 + 3)^2";
2   Integer result = calculator.evaluate(expression);
3   assertEquals(new Integer(16),result);
```

Figura 3: Ejemplo de un test JUnit

2.1.1 *Generación automática de escenarios de testing*

La automatización de la ejecución de los tests es muy útil, por diversas razones, pero esencialmente porque permite ejecutar y re-ejecutar los tests asociados a una aplicación de manera muy conveniente y eficientemente, especialmente

cuando la comprobación de que los resultados esperados coinciden con los obtenidos (los *asserts*) también se chequean automáticamente. Sin embargo, la construcción de tests es un proceso cuya automatización es sustancialmente más difícil, entre otras razones porque requiere que alguien provea los escenarios en los cuales evaluar un programa particular. La generación de escenarios de prueba de manera automática tiene gran valor práctico, puesto que permite automatizar parte del proceso de *generación* de tests, y reducir los costos asociados a este proceso.

Actualmente existen numerosas aplicaciones cuyo objetivo principal es la generación automática de entradas para testing. Algunos de los enfoques utilizados para la generación automática de entradas incluyen:

- *Generación aleatoria de entradas*, que consiste en elegir, para algún conjunto de valores predefinidos y de manera aleatoria, elementos del mismo para usar como entradas del programa a evaluar. Esta metodología es muy sencilla de implementar para valores de tipos no estructurados, como por ejemplo enteros y valores numéricos. Para la generación de valores de tipos estructurados, como por ejemplo una lista simplemente encadenada, es necesario combinar la generación aleatoria para cada componente que conforma el tipo estructurado. Sin embargo, estos tipos estructurados tienen invariantes que deben ser satisfechos para que una instancia (un valor de este tipo) sea considerada válida. En las Figuras 4 y 5 se pueden ver ejemplos de generación aleatoria para tipos primitivos y estructurados respectivamente.

- *Generación exhaustiva (acotada) de entradas*, que consiste en utilizar todos los valores disponibles dentro de un espacio acotado de entradas del programa a evaluar. Para valores primitivos esto consistiría en utilizar todos los valores entre un valor mínimo y un máximo (que definen un dominio acotado). Evidentemente, una cota debe definir un sub-dominio finito, para que la generación exhaustiva acotada termine. Para valores que corresponden a tipos estructurados, la metodología es similar a generación aleatoria. Esta metodología parecería en primera instancia generar una mayor confianza ya que permite evaluar un programa bajo todas las posibles entradas dentro de una determinada cota. La cantidad de entradas que se obtienen por generación exhaustiva llegan muy rápido a cantidades que hacen al proceso de generación o al de testing inviable. En [Boyapati et al., 2002] se presenta una herramienta de generación exhaustiva acotada de instancias para tipos estructurados, y se muestra cómo la cantidad de valores generados aumenta exponencialmente respecto a las cotas utilizadas.

```
Set intInputs = new Set();
for (int i = 0; i<10; i++) {
  int rndValue = 10 + random.nextInt(11);
  intInputs.add(rndValue);
}
```

Figura 4: Ejemplo de generación aleatoria de 10 valores enteros entre 10 y 20

```
Set structuredInputs = new Set();
for (int i = 0; i<10; i++) {
  int rndSize = random.nextInt(10);
  List rndList = new List();
  for (int elemIdx = 0; elemIdx < rndSize; elemIdx++) {
    int rndValue = random.nextInt(50);
    rndList.add(rndValue);
  }
  structuredInputs.add(rndList);
}
```

Figura 5: Ejemplo de generación aleatoria de 10 listas de enteros con tamaño entre 0 a 10, y con valores entre 0 a 50

2.1.2 *El problema del oráculo*

Teniendo la capacidad de generar escenarios de testing de manera automática, y dado que la ejecución del programa a evaluar, bajo una entrada particular, es trivial de automatizar, conseguimos automatizar una parte importante de la generación de tests. Sin embargo, el resultado esperado de la ejecución de un programa es algo que escapa a la generación automática directa, pues depende de la *especificación* del programa desarrollado. A diferencia de las especificaciones formales al estilo de las provistas en lógica de Hoare o formalismos similares, las especificaciones en los tests suelen ser particulares a cada escenario. Por ejemplo, en el caso del test en la Figura 3, el valor esperado *16* es la *especificación* para ese escenario, y obviamente fue provisto por el desarrollador del test, de manera manual.

El ejemplo anterior corresponde a un test desarrollado manualmente. En el contexto de la generación automática de tests, y con el propósito de automatizar de manera completa la generación de tests, se necesita entonces alguna fuente de la cual tomar la *especificación* del comportamiento esperado del software. Este problema, el de dado un programa a evaluar y una entrada específica para el mismo poder determinar el resultado esperado de la ejecución del programa en esa entrada, se conoce como el *problema del oráculo*. Existen diferentes enfoques para atacar este problema, entre ellas las siguientes:

1. Describir manualmente la salida esperada. Esta solución es claramente no automática, es tediosa e insume tiempo, dado que los resultados esperados se describen por cada test.

2. Provisión de especificaciones del comportamiento esperado. Si se cuenta con una especificación al estilo de una post-condición, ésta puede actuar como oráculo del comportamiento esperado. Ahora bien, si el objetivo es la automatización del proceso de generación automática de tests, necesitamos que estas especificaciones sean *ejecutables*, es decir, que puede comprobarse automáticamente, dado un estado específico de programa, si el mismo satisface la especificación o no. A modo de ejemplo, si necesitamos producir tests para un método que ordena listas de enteros, el resultado deben ser listas que sean permutaciones de la original, y que estén ordenadas. Esto se puede comprobar mecánicamente, y actuaría como oráculo del comportamiento esperado.

3. Testing diferencial, el cual consiste, dado un programa bajo evaluación `P` y una entrada `E` sobre la cual se quiere evaluar el comportamiento de `P`, en la utilización de un programa alternativo (ya implementado) `P'`

con el *mismo* comportamiento que el esperado de `P`. El oráculo constará de un chequeo del estilo de `P/(E) == P(E)`. Evidentemente, este enfoque requiere una confianza suficiente en el correcto comportamiento de `P/` (que puede ser, por ejemplo, una versión del programa desarrollado más simple y posiblemente menos eficiente, pero cuya corrección es más fácil de garantizar).

4. Testing de regresión, el cual se puede entender como un caso particular de *testing diferencial*, donde una versión anterior del programa a evaluar se utiliza como alternativa "confiable". Esta técnica es utilizada principalmente para evaluar que el comportamiento previo de un programa no fue modificado involuntariamente al, por ejemplo, agregar nuevas características.

2.1.3 *Modelo RIP*

La existencia de fallas en el código no es, en muchos casos, fácil de detectar. El modelo RIP describe, precisamente, las condiciones que deben darse para la detección de una falla:

- *Reachability*, el defecto en el código debe ser alcanzable por la ejecución del programa bajo algún escenario particular;
- *Infection*, el defecto debe "infectar" el estado del programa, es decir, al ejecutar el código que contiene el defecto, debe ocurrir un cambio en el estado del programa que difiere del estado del programa si el mismo no tuviera el defecto (o difiere respecto al estado esperado);
- *Propagation*, el cambio del estado debe propagarse hasta algún punto observable, para permitir diferenciar una salida incorrecta (error) de la correcta.

La Figura 6 contiene una implementación de un programa que calcula el máximo entre dos números. El defecto se encuentra en la segunda condición, `b < a`. Cualquier entrada en la cual no se cumpla que `a > b`, va alcanzar la sentencia con el defecto; la infección se da cuando una entrada que cumpla con `b > a`, no cambia el valor de `result` para contener el valor de `b`; la propagación se da porque el valor incorrecto en `result` se retorna en la última sentencia del programa. Incluso al tener un escenario que cumpla con el modelo *RIP* para un defecto, éste no va a ser detectado si no se cuenta con un oráculo apropiado, en este caso, validar que por ejemplo para los valores *5* y *3*, el resultado esperado es *5*.

```
1     public int max(int a, int b) {
2       int result = a;
3       if (a > b) {
4         result = a;
5       } else if (b < a) {
6         result = b;
7       }
8       return result;
9     }
```

Figura 6: Una implementación incorrecta de un método *max*

2.1.4 *Criterios de cobertura*

Como se mencionó anteriormente, al utilizar testing como técnica para evaluar a un programa con respecto a las tareas que éste realiza, es necesario seleccionar un conjunto finito de escenarios sobre el cual ejecutar y evaluar el programa. La selección debe hacerse de forma tal que si el comportamiento del programa es el correcto dentro de los escenarios seleccionados, sea posible generalizar el correcto comportamiento a todos los escenarios posibles. Intuitivamente lo que se busca es poder seleccionar el conjunto de escenarios con la mejor capacidad de detectar potenciales fallas. Analizar la capacidad de un conjunto de tests en detectar fallas potenciales no es posible, al menos de manera directa, por la razón de que éstas no son conocidas. En la práctica, la evaluación de la capacidad de un conjunto de tests en detectar fallas potenciales se realiza de manera indirecta, donde el objetivo es definir una métrica tal que mientras más alto sea el valor asociado a la misma, mayor sea la confianza sobre los tests en detectar fallas potenciales.

Los criterios de cobertura definen métricas de evaluación generando metas, o requisitos, para la test suite bajo evaluación, y evalúan cuántas de estas metas u objetivos son satisfechas por la suite. Un test específico pueden alcanzar o "cubrir" varias metas simultáneamente. Dado que un criterio de evaluación para una test suite no puede directamente analizar la capacidad del mismo en detectar fallas (salvo para aquellas conocidas), en el diseño de criterios de testing se intenta construir metas evaluables, y que indirectamente impliquen, o estén relacionadas con, la posibilidad de detectar fallas. El modelo *RIP* ayuda al diseño de criterios, como veremos a continuación.

Como criterios de cobertura, aquellos denominados de *caja blanca*, se basan en la estructura del programa para generar metas a cubrir. Por ejemplo, *cobertura*

de sentencias genera como metas la ejecución de las sentencias del programa. Este criterio se enfoca principalmente en *alcanzabilidad* (reachability), claramente influenciado por el modelo RIP: para que una suite sea capaz de detectar defectos, debe en principio alcanzarlos.

Los criterios de *caja negra* se enfocan en la especificación del programa para generar metas. Por ejemplo, el criterio de *partición de clases de equivalencia* se enfoca en dividir las entradas en conjuntos para los cuales el comportamiento del programa debería ser "similar". En el caso de *max*, se podrían definir diferentes clases, por ejemplo que el primer valor sea menor que el segundo, que sean iguales, o que el primer valor sea mayor que el segundo.

Un ejemplo de criterios de cobertura de caja blanca se muestra en la Figura 7. En esta Figura, el método *countEvenIn(int[])* toma como entrada un arreglo de enteros y retorna la cantidad de números en el arreglo que son pares. La lógica del programa es bastante simple: se recorre uno a uno los elementos del arreglo y se verifica si el resto de dividir un elemento por 2 es cero; si lo es entonces el elemento es par y se incrementa una variable donde efectivamente se cuenta cuántos pares se encontraron. El criterio de cobertura de sentencias genera como objetivos a cubrir la ejecución de cada sentencia en el programa bajo evaluación. La *cobertura de ramas* añade como requisitos que las sentencias de bifurcación (sentencias *if*, *while*, *for*, entre otras) se ejecuten una vez con su condición siendo verdadera y otra con ésta siendo falsa. Solamente con un test que evalúe el método con un arreglo de tamaño 1 conteniendo un número par, se obtiene una cobertura del 100 % para sentencias (Figura 10). Sin embargo, si evaluamos cobertura de ramas podemos observar que con este test no logramos una cobertura del 100 % (Figura 11), en particular porque no se evalúa el caso en el cual un elemento es impar. Al generar tests para este criterio podemos simplemente cambiar la entrada a un arreglo de tamaño 4 con los elementos 1, 2, 3, y 4, obteniendo una cobertura de ramas del 100 % (Figura 10).

En el caso de criterios de caja negra, varios criterios comúnmente utilizados se basan en analizar la especificación de las entradas esperadas por el programa a analizar. En general se comienza partiendo el dominio de cada entrada de acuerdo a ciertas características provistas. Éstas deben dividir el conjunto de valores de cada entrada en subconjuntos disjuntos, y la unión de todos los subconjuntos debe resultar en el dominio completo de cada entrada. Por ejemplo, si una entrada es un entero, es posible dar como característica a *ser par*, lo que divide todo el conjunto de los enteros en dos subconjuntos disjuntos (uno conteniendo todos los enteros pares y otro conteniendo a todos los impares), tal que al unir estos conjuntos se obtiene nuevamente a todos los enteros. Cada criterio basado en las entradas del programa a evaluar define de qué forma combinar valores de cada subconjunto en los casos donde el programa tiene

```
public static int countEvenIn(int[] values) {
    int count = 0;
    int idx = 0;
    while (idx < values.length) {
        if (values[idx] % 2 == 0)
            count++;
        idx++;
    }
    return count;
}
```

Figura 7: Método para contar números pares en un arreglo de enteros.

```
@Test
public void testCountEvenInNonEmpty() {
    int[] input = new int[] {2};
    int expected = 1;
    int obtained = countEvenIn(input);
    assertEquals(expected, obtained);
}
```

Figura 8: Tests orientados a satisfacer cobertura de sentencias para el código de *countEvenIn(int[])* 7.

```
@Test
public void testCountEvenInNonEmpty() {
    int[] input = new int[] {1,2,3,4};
    int expected = 2;
    int obtained = countEvenIn(input);
    assertEquals(expected, obtained);
}
```

Figura 9: Tests orientados a satisfacer cobertura de ramas para el código de *countEvenIn(int[])* 7.

```java
public static int countEvenIn(int[] values) {
    int count = 0;
    int idx = 0;
    while (idx < values.length) {
        if (values[idx] % 2 == 0)
            count++;
        idx++;
    }
    return count;
}
```

Figura 10: Medición de cobertura de *countEvenIn(int[])* 7 con una cobertura del 100 % tanto de sentencias como de ramas.

```java
public static int countEvenIn(int[] values) {
    int count = 0;
    int idx = 0;
    while (idx < values.length) {
        if (values[idx] % 2 == 0)
            count++;
        idx++;
    }
    return count;
}
```

Figura 11: Medición de cobertura de *countEvenIn(int[])* 7 con una cobertura del 100 % de sentencias pero no de ramas.

Bloque	array vacÃŋo	what pertenece	with igual a what
1	T	T	T
2	F	F	F

Tabla 1: Características con las que se puede dividir los conjuntos de las entradas del programa *replace(int[], int, int)* 12.

varias entradas. Por ejemplo, combinar todos con todos, o cada subconjunto debe estar representado por lo menos una vez sin importar con qué otro subconjunto es combinado. Para el ejemplo de la Figura 12, que muestra un método que dado un arreglo de enteros, un valor entero a buscar en el arreglo, y un valor con el cual reemplazar el valor anterior, realiza los reemplazos y retorna cuantos fueron realizados, en la Tabla 1 se muestran posibles características para dividir cada entrada: que el arreglo sea o no vacío; que el valor a buscar esté o no en el arreglo; y finalmente que el valor con el cual reemplazar al anterior sea igual o distinto al reemplazado. Vale aclarar que si bien todas las características dividen las entradas en dos subconjuntos, esto no debe ser necesariamente así. Una característica que divida en más subconjuntos es perfectamente válida, siempre que cumpla con las propiedades mencionadas anteriormente. Como ejemplo de combinación de valores de cada subconjunto definidos por las características en la Tabla 1, podemos utilizar el criterio *Each Choice Value*, el cual determina que cada subconjunto debe estar representado al menos una vez en los tests. Esto lleva a los tests que se muestran en la Figura 13, donde tenemos dos tests, uno donde se utiliza un arreglo no vacío, con el valor a reemplazar perteneciendo al arreglo (2) y el valor con el cual reemplazar siendo igual al anterior; el segundo test utiliza un arreglo vacío, obviamente el elemento a buscar no pertenece al mismo y el valor con el cual se realiza el reemplazo es distinto al anterior. El hecho de que como se puede apreciar en la Figura 14, los tests anteriores logran una cobertura de ramas del 100 % al tiempo que éstos son evidentemente un conjunto muy pobre de escenarios, sirve para remarcar cómo un criterio puede ser satisfecho al mismo tiempo que los tests que lo satisfacen son claramente de muy mala calidad.

```
/**
 * Reemplaza todos los valores que son iguales a
 * un valor especificado con otro valor
 * especificado y retorna cuantos valores fueron
 * reemplazados.
 *
 * @param array donde se realizan los reemplazos
 * @param what el valor a buscar
 * @param with el valor con el cual reemplazar
 * @return la cantidad de reemplazados realizados
 * ---------------------------------------------------
 * Pre:
 *     array != null
 * Post:
 *     foreach v in 0..(array.length-1) |
 *         old(array[v]) == what => array[v] == with)
 *     count v in 0..(array.length-1) |
 *         old(array[idx]) == what)
 */
public static int replace(int[] array, int what, int with) {
    int idx = 0;
    int changes = 0;
    while (idx < array.length) {
        if (array[idx] == what) {
            array[idx] = with;
            changes++;
        }
        idx++;
    }
    return changes;
}
```

Figura 12: Método para reemplazar todos los valores en un arreglo *array* que son iguales a un valor especificado (*what*) por otro valor especificado (*with*) y retornar cuantos cambios se realizaron.

```
@Test
public void testReplaceNonEmptyPresentEqual() {
    int[] input = new int[] {1,2,3};
    int what = 2;
    int with = 2;
    int changesExpected = 1;
    int[] arrayExpected = new int[] {1,2,3};
    int changedObtained = replace(input, what, with);
    assertArrayEquals(arrayExpected, input);
    assertEquals(changesExpected, changedObtained);
}

@Test
public void testReplaceEmptyNotPresentDifferent() {
    int[] input = new int[] {};
    int what = 2;
    int with = 2;
    int changesExpected = 0;
    int[] arrayExpected = new int[] {};
    int changedObtained = replace(input, what, with);
    assertArrayEquals(arrayExpected, input);
    assertEquals(changesExpected, changedObtained);
}
```

Figura 13: Tests para satisfacer criterio de cobertura *Each Choice Coverage* para 12 basado en las características definidas en 1.

```
public static int replace(int[] array, int what, int with) {
    int idx = 0;
    int changes = 0;
    while (idx < array.length) {
        if (array[idx] == what) {
            array[idx] = with;
            changes++;
        }
        idx++;
    }
    return changes;
}
```

Figura 14: Cobertura de ramas lograda por los tests definidos en 13.

3 Mutation testing

Mutation testing es un criterio de cobertura que puede ser considerado de *caja blanca*, donde las metas a cubrir por la test suite están representadas por fallas artificiales, las cuales deben ser detectadas por la test suite bajo evaluación. Las fallas artificiales se representan por variantes del programa original, en donde cada una tiene un cambio sintáctico simple, representando un defecto. Cada variante se denomina *mutante*, mientras que la falla artificial asociada se denomina *mutación*. Cuántos de los mutantes generados son detectados por la test suite, es el valor asociado a este criterio, y se denomina *mutation score*.

Las fallas artificiales involucradas en mutation testing se producen en base a *operadores de mutación*. Un operador de mutación define los cambios sintácticos que se van a realizar sobre un programa, típicamente eligiendo tipos particulares de expresiones en el mismo, y dando lugar a familias de mutantes. Dos ejemplos tradicionales de operadores de mutación son *reemplazo de operadores relacionales*, el cual reemplaza un operador relacional por todos los otros soportados por el lenguaje de programación utilizado, y *reemplazo de operadores aritméticos*, que realiza cambios similares pero sobre operadores aritméticos. La Figura 15 muestra un programa con varias mutaciones, en un mismo bloque de código. Cada mutación está indicada con Δ, y sustituye la línea (no mutada, es decir, sin Δ) inmediata superior. Las mutaciones 1Δ, 2Δ and 3Δ en esta Figura corresponden a reemplazos de operadores relacionales, mientras que 4Δ corresponde a reemplazo de operadores aritméticos.

Al igual que en cualquier otro criterio de testing, mientras más cercano al 100 % sea el valor asociado, más confianza se puede tener en la calidad de la test suite evaluada. En el caso de mutation testing, este valor aumenta mientras más mutantes sean detectados y disminuye en caso contrario. Sin embargo, existen situaciones que afectan de manera negativa a la confianza sobre este valor. Ciertas fallas son triviales de detectar, ya sea por que causan que la compilación o la ejecución (bajo cualquier entrada) falle, o sólo requieren alcanzabilidad para hacerlo. Por ejemplo, la sentencia

$\text{if (c) } x = \text{array}[-1];,$

```
int countEven( int[] input ) {
  int count = 0;
  for (int i = 0; i < input.length; i++) {
  1Δfor (int i = 0; i <= input.length; i++) {
  2Δfor (int i = 0; i != input.length; i++) {
    int value = input[i];
    if (value %2 == 0) {
    3Δ if (value %2 != 0) {
      count = count + 1;
      4Δ count = count / 1;
    }
  }
  return count;
}
```

Figura 15: Ejemplos de un programa y posibles mutaciones

sólo requiere una entrada que satisfaga c para detectar la falla. Ni siquiera es necesario un oráculo para evaluar el resultado: la ejecución genera una terminación abrupta, que manifiesta un bug. Las fallas triviales aumentan el mutation score sin implicar una mejora en la calidad de la test suite. En el espectro opuesto, puede haber fallas artificiales que tengan el mismo comportamiento semántico que el código original. Por ejemplo, la sentencia

for (int i = 0; i < 10; i++)

es equivalente a

for (int i = 0; i < 10; ++i).

Estos mutantes son llamados equivalentes y por lo tanto indistinguibles del programa original, lo que genera metas que no pueden ser cubiertas, que a su vez disminuyen el valor de mutation score, aunque la test suite no tiene una peor calidad por no ser capaz de detectar estos mutantes. Finalmente, un valor alto de mutation score no significa nada si no existe una relación entre detectar fallas artificiales y reales. Discutiremos en mayor detalle esta última situación a continuación.

3.1 OPERADORES DE MUTACIÓN SUFICIENTES

El estudio llevado a cabo en [Offutt et al., 1996] evaluó el conjunto de operadores de mutación utilizados por *Mothra*, una herramienta de mutation

testing para el lenguaje Fortran-77. En este estudio, se define el siguiente conjunto de operadores, considerados *suficientes*:

ABS Modifica cada expresión aritmética por 0, un valor positivo, y un valor negativo.

AOR Reemplaza cada operador aritmético por todos los operadores legales.

LCR Reemplaza operadores lógicos o condicionales por otros válidos.

ROR Reemplaza operadores relacionales.

UOI Inserta operadores unarios en expresiones compatibles.

Intuitivamente, un conjunto de operadores se considera suficiente si la adición de otros operadores de mutación no mejora la precisión del mutation score como métrica de la calidad de una suite. Si bien otros estudios sobre conjuntos suficientes de operadores se han realizado, entre ellos [Namin et al., 2008], todos arriban esencialmente a conclusiones similares. Por ejemplo, en [Namin et al., 2008] se define un conjunto de operadores suficientes muy similar al previamente descripto, con la diferencia de que la herramienta de mutación utilizada en este último ofrece una mayor cantidad de operadores de mutación, que en muchos casos están incluídos en los anteriores.

Cabe destacar que estos estudios siempre se ven afectados por la misma amenaza de validez: realizar el estudio bajo un conjunto de programas determinados con un conjunto de tests particulares no da necesariamente resultados generalizables a cualquier contexto de testing. Sin embargo, todos llegan al mismo conjunto de cambios, ya sea realizados por los mismos operadores o por operadores similares.

3.2 PROPIEDADES DE OPERADORES DE MUTACIÓN

En principio, un operador de mutación es una función que se aplica a elementos particulares de un programa, ya sea código fuente o binarios, y genera versiones modificadas de éstos. Desde el punto de vista de mutation testing, un operador de mutación a su vez debe tener un objetivo claro, como emular cierto tipo de fallas. Para que mutation testing sea un criterio efectivo, algunas propiedades deben ser tenidas en cuenta para el diseño de los operadores de mutación involucrados. La *eficiencia temporal y relativa a otros recursos* obligan

a que el conjunto de mutantes generados se mantenga acotado, de la misma forma en que, en el contexto general de testing, es necesario mantener al conjunto de tests a utilizar para evaluar un programa también acotado. La *efectividad* de mutation testing como forma de evaluar la calidad de una test suite, es decir, la habilidad de la suite para detectar potenciales fallas en un programa bajo análisis, es otra característica importante asociada a los operadores de mutación utilizados. Finalmente, los operadores de mutación deben ser buenos *representantes de fallas reales*, dado que aunque un conjunto de mutantes sea muy efectivo para evaluar un conjunto de tests, esto sólo implica que es capaz de detectar cambios semánticos, pero no implica pero que sea efectivo en detectar cambios semánticos provenientes de fallas reales. Esto es así principalmente porque mutation testing intenta emular fallas complejas mediante fallas simples. Estas caracterśticas están vinculadas a cuán significativo es el valor de mutation score que resulta de un conjunto particular de mutantes, es decir, a la *confianza* en el mutation score como medida de calidad. A continuación vamos a mencionar qué características afectan o están relacionadas con las propiedades anteriores, y cómo es posible medirlas.

3.2.1 *Equivalencia*

La equivalencia entre dos programas es una propiedad definida por la relación `Eq(P, P')` $\equiv \nexists$ `E : P(E) != P'(E)`, es decir, se establece que dos programas `P` y `P'` son equivalentes si y sólo si no existe un escenario `E` tal que el comportamiento de los programas en el mismo sea diferente. Determinar la equivalencia entre programas es indecidible, por lo cual los enfoques automáticos que intentan atacar este problema son incompletos. En el contexto de mutation testing, la equivalencia entre programas es relevante, y la generación de mutantes equivalentes al programa original (aquel sobre el cual se realiza testing) resulta en un conjunto de mutantes que no puede ser detectado (ni nunca podría serlo) pero que disminuye el mutation score, generando la falsa idea de que es necesario extender el conjunto de tests para detectar a éstos. Como hemos mencionado, la generación de mutantes equivalentes es indeseable, dado que implica una disminución en el valor del mutation score sin significar una deficiencia de parte de la test suite en detectar ciertas fallas artificiales. Otro caso de equivalencia se puede dar entre mutantes; si dos mutantes diferentes son equivalentes, uno es detectado por la suite si y sólo si el otro también lo es, lo cual puede traer aparejado un incremento el valor del mutation score sin significar una mejora de parte de la test suite en detectar más fallas artificiales. Es importante destacar que es imposible evitar por completo la generación de mutantes equivalentes, solo siendo posible disminuir la generación de los mismos.

Dentro de la investigación sobre la detección (evaluación) de esta característica y su impacto en el análisis de test suites usando mutation testing, el trabajo que se presenta en [Schuler and Zeller, 2010] propone la utilización de diferencia en cobertura de código y análisis de flujo de datos para determinar potencial equivalencia. Por otra parte, [Just et al., 2013] utiliza detección de restricciones condicionales para alcanzar el código mutado y *SAT Solving* para determinar si es posible satisfacer dichas restricciones al tiempo que se obtiene un valor distinto al del programa original en ese punto. Esto es similar en principio a *weak mutation*, una variante de mutación en la cual se considera que un mutante es detectado si en el estado siguiente a la mutación se detecta una diferencia en el estado de programa respecto a aquel del programa original, pero añadiendo control de alcanzabilidad y una verificación exhaustiva acotada para detectar si es posible que exista una diferencia. En [Grün et al., 2009] observan que manualmente, para los casos de estudios utilizados, un programador avanzado tarda aproximadamente 15 minutos en promedio para analizar mutantes equivalentes. Y claramente la existencia de mutantes equivalentes disminuye artificialmente el mutation score dando la falsa impresión de que es necesario agregar más tests, como ya mencionamos anteriormente. Si analizamos el impacto de la equivalencia entre mutantes, podemos diferenciar dos casos particulares:

- equivalencia entre mutantes detectados: dos mutantes diferentes m_1 y m_2, pero equivalentes, son detectados por la test suite bajo evaluación. Esta situación lleva a un incremento del mutation score, básicamente al detectar más de una vez el mismo mutante;
- equivalencia entre mutantes no detectados: dos mutantes diferentes m_1 y m_2, pero equivalentes, son sobrevivientes. Esto lleva a un decremento del mutation score por básicamente fallar en detectar el mismo mutante dos veces.

3.2.2 *Dificultad de detección*

Así como los mutantes equivalentes son indeseables por ser imposibles de detectar, los mutantes que son sólo detectables por un conjunto pequeño de tests, son altamente deseables. Éstos son denominados *stubborn* [Hierons et al., 1999]. La detección de estos mutantes requiere tests de “mejor calidad”, y si bien existen estudios que evalúan la generación de stubborns por operador [Yao et al., 2014], esto depende del conjunto de programas utilizados y los tests asociados. Con respecto a este obstáculo, en [Visser, 2016], proponen el uso de *model counting* sobre programas más simples pero utilizando un estudio más exhaustivo. Algunas medidas preventivas para evitar generar mutantes

Tests	$M1$	$M2$	$M3$
1	•	•	
2		•	•
3			•

Figura 16: Ejemplo de subsuma de mutantes

triviales de detectar incluyen controles más estrictos en la generación, para evitar mutantes que no compilen. Por ejemplo, AOIS, un operador que inserta $++$ y $--$ en variables aritméticas, puede generar mutantes inválidos si no comprueba que la variable a mutar no ha sido definida como inmutable (una constante). La razón por la cual resulta complicado detectar o prevenir una baja dificultad de detección para un conjunto de mutantes, es que es difícil definir precisamente a esta dificultad. Mientras que equivalencia es una propiedad binaria, en el sentido de que, dos programas son o no equivalentes, la dificultad de detección es una propiedad cuantitativa, muy difícil de definir.

3.2.3 *Subsumption*

Subsumption se define como una relación entre mutantes respecto a los tests que los detectan, e intenta capturar redundancia entre mutantes. Más precisamente, se dice que un mutante m_1 subsume a otro m_2 si los tests que detectan a m_2 incluyen a todos aquellos que detectan a m_1. Un mutante subsumido es considerado redundante, no aportan a la evaluación de los test e "infla" el mutation score obtenido. Además de la noción de mutantes redundantes, esta relación representa una forma indirecta de evaluar la dificultad de detección de mutantes: los mutantes que subsumen a otros pero no son a su vez subsumidos, son detectados por pocos tests. Presentado inicialmente en [Offutt et al., 1996], mutant subsumption es utilizado por [Ammann et al., 2014] y [Just et al., 2017] para evaluar la utilidad de mutantes dentro de mutation analysis. Partiendo de que los mutantes utilizados en mutation testing tienen como objetivo ejercitar a un conjunto de tests para evaluar de manera indirecta la capacidad de los mismos en detectar potenciales fallas reales, aquellos que ejerciten de manera más específica a los tests, son considerados entonces mejores. Un ejemplo de subsumption de mutantes se puede ver en las Figuras 16 y 17.

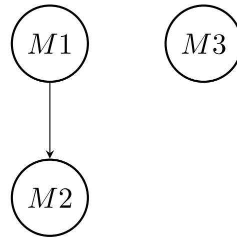


Figura 17: Grafo de subsuma basado en la Tabla 16

3.2.4 *Acoplamiento*

Coupling se define como el acoplamiento entre fallas reales y mutantes, y es una propiedad altamente deseable. Sin la misma, mutation testing perdería su utilidad, ya que no existiría una correlación entre un mutation score alto y una buena capacidad de parte de la test suite para detectar fallas reales. El trabajo más importante sobre este tema, y uno que nos representa una motivación importante para el desarrollo de nuestro operador de mutación presentado en esta tesis, es [Just et al., 2014]. El acoplamiento entre mutantes y fallas reales es una relación que indica que existe una correlación positiva entre la capacidad de un conjunto de tests de detectar un conjunto de mutantes, y la capacidad del conjunto de tests en detectar ciertas fallas reales. Por otra parte, el acoplamiento entre diferentes mutaciones (diferentes fallas artificiales) representa una situación indeseable, similar al caso de equivalencia entre mutantes. Una manera de medir acoplamiento entre un programa, con una falla real conocida, y un conjunto de mutantes, es evaluar si los tests que detectan la falla, son los mismos que detectan al conjunto de mutantes. En el trabajo previamente citado se utiliza esta métrica, aunque con la exigencia adicional de que sólo se muta el código asociado a la reparación de la falla real.

4

PRVO: UN META-OPERADOR PARA EXPRESIONES DE NAVEGACIÓN

4.1 HISTORIA DE PRVO

El operador de mutación de expresiones de navegación que definiremos en esta tesis surge a partir de un operador sustancialmente más simple (qu de hecho, no muta navegaciones), proveniente de una herramienta específica de testing de mutación, *µJava*. En *µJava* existe un operador de mutación denominado *PRV*, y cuyas mutaciones se basan en **P**olimorfismo, y sustituyen asignaciones de **R**eferencias a **V**ariables, por otras de un tipo compatible [Ma and Offutt, 2018; Ma et al., 2002]. Este operador modifica una variable que está siendo asignada a una referencia (un objeto), por otra que sea compatible con la referencia. En la Figura 18 vemos un ejemplo de una mutación producida por este operador. Una clara limitación de este operador es que sólo se aplica a asignaciones, y muta la parte derecha de la asignación. Su aplicación a otro tipo de expresiones o sentencias sería deseable. La segunda limitación está precisamente en sólo considerar variables, y está relacionada al objetivo que tiene el operador: en lenguajes orientados a objetos, el polimorfismo permite definir métodos o expresiones utilizando variables que pueden ser instanciadas por un conjunto de tipos distintos. La Figura 19 muestra ejemplos de polimorfismo por sobrecarga y herencia. En el primer caso, para cubrir los tres métodos definidos es necesario que la expresión con la que se llama al método *foo* sea de tres tipos diferentes; esto sólo es posible con *PRV* cuando se utiliza la verificación del tipo de una instancia para cambiar el comportamiento del código. En la Figura 20 se muestra un ejemplo del tipo de código para el que está diseñado este operador. Para los casos de sobrecarga mostrados en la Figura 19, si antes de la llamada al método se tiene una asignación **Object o = v;**, cambiar la variable por otra, como lo haría este operador, no causaría que se llame a otro método más que a *foo*, que toma un *Object*. Sin embargo modificando la variable con la que se está llamando al método si causaría, dependiendo del tipo de la variable, la llamada a otra definición del método.

```
List l = null;
LinkedList ll = new LinkedList();
ArrayList al = new ArrayList();
l = ll;
δ l = al;
```

Figura 18: Un ejemplo de una mutación de *PRV* donde se reemplaza *ll* por *al* en la asignación original a *l*

```
//(i) : polimorfismo por sobrecarga
public void foo(Integer i) {...}
public void foo(String s) {...}
public void foo(Object o) {...}
...
foo("Test");
foo(1);
foo(new LinkedList());

//(ii) : polimorfismo por herencia
public void append(List a, List b) {
  for (Object o : b) {
    a.add(o);
  }
}
```

Figura 19: Ejemplos de polimorfismo por sobrecarga (*i*) y por herencia (*ii*)

```
public Number abs(Number a) {
    if (a instanceof Integer) {
        Integer res = (Integer) a;
        return -res;
    } else if (a instanceof Float) {
        Float res = (Float) a;
        return -res;
    }
    ...
}
```

Figura 20: Ejemplo de polimorfismo al cual *PRV* apunta

```
current //expresión de tamaño 0
null    //expresión de tamaño 0
current.next //expresión de navegación de tamaño 1
current.next.value //expresión de navegación de tamaño 2
```

Figura 21: Ejemplos de expresiones de navegación

El operador que definimos en este capítulo, *PRVO*, surge como una extensión a *PRV*. Esta extensión no sólo incluye las mutaciones generadas por el operador original, ahora aplicadas en cualquier contexto y sin estar limitadas a la parte derecha de una asignación, sino también admite la mutación de expresiones de navegación.

4.2 EXPRESIONES DE NAVEGACIÓN

Para poder definir el operador *PRVO*, primero necesitamos definir qué consideramos una *expresión de navegación*. En lenguajes orientados a objetos, el acceso a miembros de una clase es de uso común, y se realiza mediante un operador de navegación que puede tomar distintas formas sintácticas, con la notación *punto* siendo la más común. Esta notación permite encadenar expresiones, donde la expresión encadenada comienza por un miembro de la última parte de la expresión anterior. Nos referiremos a una expresión de navegación, a una que cuenta con uno o más de estos operadores. El tamaño de una expresión de navegación se puede definir de distintas formas, nosotros la definiremos por el número de operadores de navegación involucrados. Una expresión de navegación de tamaño 1 entonces tendría la forma $a.b$. Ejemplos de distintas expresiones de navegación se pueden ver en la Figura 21, donde vale aclarar que una vez que una expresión de navegación se reduce a menos de una utilización del operador de navegación, ésta se considera de tamaño 0, y podría considerarse un caso especial de una navegación.

En una expresión de navegación, el tipo de la misma está dado por el de la última expresión en la cadena. Por ejemplo, $a.b.c$ es una expresión encadenada de tamaño dos, cuyo tipo está dado por el de c. Una expresión de navegación bien tipada se puede definir de dos formas. Por pertenencia: para toda cadena `a.b`, `b` debe ser un miembro de la clase de `a`. O mediante caminos en un grafo: si cada tipo se representa mediante un nodo, y los miembros de una clase se representan como arista sque une los tipos correspondientes, entonces todas las posibles expresiones de navegación se corresponden con caminos en este grafo. Por ejemplo, la expresión $header.next.value.toString()$, es una expresión de

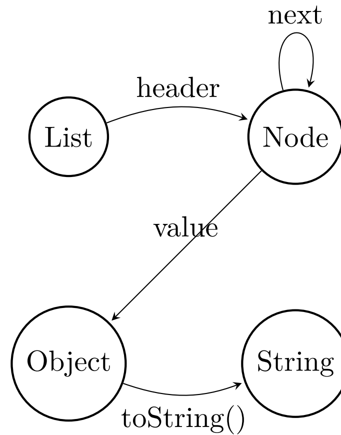


Figura 22: Una expresión de navegación modelada mediante un grafo

navegación de tamaño 3, que se considera válida respecto al grafo de la Figura 22.

4.3 PRVO

Como mencionamos anteriormente, nuestro operador extiende a *PRV*, esto lo hace de dos formas: por un lado extiende el reemplazo de variables en asignaciones a expresiones en general y extendiendo en que tipo de contextos éstas son reemplazadas. Por el otro lado, añade la modificación de expresiones de navegación. Para poder unir ambas extensiones necesitamos considerar el tamaño de una expresión como la cantidad de operadores de navegación que posee, destacando que una expresión de tamaño 0 no es considerada de navegación pero es posible convertirla en una si tiene miembros accesibles. Comenzaremos dando una descripción general de *prvo*:

> Dada una expresión e, *prvo* va a generar mutaciones al reemplazar sub-expresiones en e (incluyendo a la misma e), respetando el tipo de la expresión, y manteniendo, incrementando, o decrementando su tamaño.

La Figura 23 da una definición libre, es decir, sin ningún tipo de restricciones, de *prvo* como una gramática. Como un ejemplo simple de mutaciones de *prvo*, consideremos la expresión `front`, las mutaciones generadas incluyen `front.next`, `null`, `front.next.next`, `front.next.elem`, `front.next.next.next`, entre otras.

Un claro problema con la definición anterior es que lleva a un número potencialmente infinito de mutaciones, al no haber restricciones, como un límite sobre cuanto puede incrementarse o decrementarse el tamaño de una expresión.

Por que damos una definición de nuestro operador que no puede usarse en la práctica?

Inicialmente parece innecesario tener una definición de un operador que resulta imposible de aplicar en la práctica. Pero la razón detrás de esto es *flexibilidad*. Al contrario de lo que pasa con otros operadores de mutación que suelen tener una definición muy simple, como cambiar un operador por otros en un conjunto finíto y predefinido, mutar expresiones tal como lo hace *PRV* y como lo hace nuestra extensión (que incluye aún más expresiones a mutar) es altamente afectado por el contexto. Solo considerando *PRV*, un código donde hay una asignación a una referencia con una variable va a generar una cantidad de mutantes directamente proporcional a cuantas variables alcanzables existan y sean compatibles con la referencia en la asignación. Dado que nuestro operador extiende a *PRV* tanto en que tipo de expresiones modifica, en donde y como, el contexto afecta aún más a cuantos mutantes son generados. Determinar que expresiones utilizar al generar mutantes es algo que si bien puede automatizarse hasta cierto punto (considerando por ejemplo la clase que se está mutando) sigue siendo un problema que va a ser mejor resuelto por la persona que está realizando el análisis y tiene conocimiento de cuales son las expresiones más significativas, de la misma forma que el elegir que operadores se utilizan queda a cargo de quien hace el análisis. Además, el objetivo a lograr con nuestro operador puede cambiar, con respecto a que tipo de fallas se quiere representar. De ésta forma, dar una definición inicialmente acotada nos restringe demasiado que fallas podemos representar y como.

Por qué decimos que prvo es un "meta-operador"?

En general, un operador de mutación se define como una serie de reglas del estilo "a partir de X genera Y", y su uso se decide en base a sus pros y cons. En el caso de *prvo*, contamos con una definición que en principio no es práctica, generar un número potencialmente infinito de mutaciones evidentemente nos impide su uso, limitar el cambio de tamaño en las expresiones generadas es totalmente posible, pero cada límite podría ser correctamente argumentado. A su vez, incluso limitando el tamaño, existen muchos factores a considerar, como propiedades de las expresiones a mutar, por ejemplo solo mutar expresiones de navegación y que se encuentren en la parte derecha de sentencias de asignación.

$$
\begin{aligned}
PRVO(x) &:= expression \\
&:= PRVO(x).expression \\
&:= expression.PRVO(x) \\
\\
PRVO(x.y) &:= expression^{*} \\
&:= PRVO(x).expression \\
&:= expression.PRVO(y) \\
&:= PRVO(x).expression.PRVO(y) \\
&:= expression.PRVO(x).PRVO(y) \\
&:= PRVO(x).PRVO(y).expression
\end{aligned}
$$

* : *puede incluir x or y*
expresión : *llamada a método* , *acceso a atributo* , *variable* ó *literal*

Figura 23: Definición abstracta de *prvo*

Claramente, al usar *prvo*, va a ser necesario contar con estas limitaciones, pero cuales van a ser dependerán del caso particular. Es por esto que consideramos *prvo* como un meta-operador, en el sentido que define una potencial familia de operadores donde cada uno es una configuración particular.

Teniendo en cuenta los criterios para el diseño de operadores de mutación, discutidos en 3.2, en particular con respecto al número de mutaciones que un operador produce, es necesario proveer algunas cotas razonables para la aplicación de *prvo*. El número de mutantes generados por *prvo* puede ser limitado al limitar tres características: las expresiones *objetivo* (donde se va a aplicar *prvo*); el *tamaño de las expresiones*, por cuanto se permite cambiar, incrementar o reducir, el tamaño de las expresiones resultantes (las mutaciones); y los *reemplazos*, es decir, las expresiones que se van a usar para el intercalado/substitución en *prvo*. En cuanto a los objetivos, *prvo* solo va a ser aplicado a expresiones de navegación, es decir, expresiones que involucran al menos una navegación. Respecto al tamaño, vamos a limitar *prvo* a producir expresiones del *mismo* tamaño que la expresión original (el número de navegaciones se mantiene). En cuanto a los reemplazos, solo vamos a reemplazar expresiones con otras de exactamente el mismo tipo (al contrario de considerar definiciones menos estrictas, que permitirían utilizar tipos compatibles más generales), que pertenezcan a la misma clase en donde se encuentra la expresión original, o en clases directamente alcanzables desde esta clase.

Hemos definido un operador de *prvo* como una serie de restricciones a la definición abstracta/general. Esto nos permite centrarnos en cierto tipo de fallas

particulares sin perder la habilidad de eventualmente generar otra configuración que se centra en otras.

4.4 FALLAS ASOCIADAS A PRVO

En [Just et al., 2014] se concluye que cierto tipo de fallas reales requieren mejorar operadores existentes o definir nuevos operadores de mutación para representarlas. Mientras que cierto tipo de fallas reales se consideran como imposibles de acoplar a mutantes. En esta tesis estamos interesados en aquellas fallas que pueden ser acopladas a *prvo*, aun cuando algunas de estas solo puedan ser parcialmente acopladas, en referencia a que solo un subconjunto de las mismas pueden representarse mediante *prvo*. En cuanto a las fallas que nos interesan, es necesario especificarlas de la forma más completa posible, y analizar como deberíamos configurar un “operador” en *prvo*, como mencionamos anteriormente, cada *operador* no se asocia directamente con un conjunto de funciones como ocurre en la práctica, sino que se refiere a una serie de restricciones particulares sobre funciones fijas ya definidas.

4.4.1 *Fallas que requieren mejorar operadores existentes*

Eliminación de sentencias

Una sentencia olvidada, donde un ejemplo simple es la implementación de un ciclo infinito con una sentencia de retorno bajo cierta condición (Figura 24); una sentencia *switch* en donde para algún caso no se escribió una sentencia de retorno o frenado (*break*); inicializaciones faltantes como en la Figura 25 donde una variable local utiliza el mismo nombre que un campo de clase, la falta de la sentencia que define esta variable resulta en un programa que compila pero incorrecto. Éstos, entre otros casos, son todos ejemplos de fallas reales que requieren un operador que elimine sentencias para poder generar el mismo tipo de falla.

En principio, ya existen operadores que realizan este tipo de mutaciones. No solo eso, sino que además no parece ser un tipo de fallas asociadas a mutaciones de expresiones de navegación.

Sin embargo, *Fluent interfaces*, un diseño muy usado en conjunto con el *patrón Builder*, permite escribir algoritmos semánticamente equivalentes a un programa imperativo, ciclos y sentencias condicionales incluídas. Un código como el mostrado en la Figura 26 no es extraño en programación orientada

```
while(true) {
  String input = getUserInput();
  if (isExitCommand(input)) {
    break;
  }
  ...
}
```

Figura 24: Ejemplo de un ciclo infinito con una sentencia *break* asociada a una condición particular

```
1   private Node cursor ...
2
3   public boolean find(int elem) {
4     Node cursor = header;
5     while(cursor != null) {
6       if (cursor.elem == elem) return true;
7       cursor = cursor.next;
8     }
9     return false;
10  }
```

Figura 25: Ejemplo en donde una inicialización faltante (linea 4), produciría un programa que compila pero incorrecto

```
list.foreach().filter(c1).if(c2).then(p1).else(p2)
```

Figura 26: Versión utilizando *interfaces fluentes* del recorrido de una lista filtrando y ejecutando condicionalmente un procedimiento.

```
List<Elem> list = ...;
for (Elem e : list) {
  if (c1(e)) {
    if (c2(e)) {
      p1(e);
    } else {
      p2(e);
    }
  }
}
```

Figura 27: Versión en lenguaje imperativo del recorrido de una lista filtrando y ejecutando condicionalmente un procedimiento.

a objetos, en la Figura 27 se muestra la implementación más común de esta expresión en lenguaje imperativo.

Esto muestra que por un lado es interesante poder mutar expresiones de navegación de este tipo. Además, uno de los principales problemas con operadores que eliminan o insertan sentencias, es que suelen generar numerosos mutantes triviales que son detectados por no compilar o por que se eliminó directamente una gran parte de código por una sentencia de control faltante. En *interfaces fluentes*, el uso de jerarquía de tipos para garantizar la correctitud de una expresión garantiza que nunca va a ser posible eliminar una sentencia cuando ésta es necesaria.

Intercambio de argumentos

Este tipo de fallas involucra utilizar argumentos con tipos correctos, pero en el orden incorrecto en la llamada a un método. En la Figura 28 se ve un método que toma como entrada dos listas y agrega la primera al final de la segunda. Los nombres de los argumentos son quizás ambígüos, y sería incluso esperable que sin una documentación apropiada, haya programadores que asuman que se hace un *append* de `thiz` *en* `that`, aunque otros podrían entender de que el *append* se hace con el orden de los argumentos resultando en `that` *en* `thiz`.

Este tipo de fallas es equivalente a aplicar dos mutaciones de cambio de referencias, append(thiz,that) −> append(that,that) −> append(that,thiz). Un tipo de mutación de expresiones como las heredadas de *PRV*, salvo que requiere dos cambios en una misma mutación. Este tipo de mutaciones no puede definirse como una serie de restricciones a la definición de *prvo* ya que incluye definiciones

```
public void append(List<E> thiz, List<E> that) {
  ...
  for (E e : thiz) {
    that.append(e);
  }
}
```

Figura 28: Método que agrega una lista al final de otra.

extra para incluir múltiples cambios. Sin embargo vale la pena notar que con dos mutaciones de *prvo* podría lograrse.

Llamada a un método similar de la misma librería

Existen ejemplos de métodos que si bien distintos, tienen una semántica relacionada, `indexOf` y `lastIndexOf` son dos métodos de la clase *java.lang.String* que permiten obtener el primer o el último índice de ocurrencia de una subcadena, respectivamente. Un operador que modifique una llamada a un método por todas las posibles, generaría una cantidad demasiado grande de mutantes para ser útil. Sin embargo cometer una equivocación al usar un método por otro con una semántica similar y provisto por la misma clase o librería, es común. Este es un caso de un tipo de fallas que se pueden generar utilizando restricciones sobre *prvo*.

4.4.2 *Fallas que requieren nuevos operadores*

Omitir la llamada a un método

Retomemos el ejemplo de *interfaces fluentes* de la Figura 26. Si en este código nos olvidáramos de llamar `filter(c1)`, la expresión seguiría siendo correcta, solamente que aplicaríamos el tratamiento condicional a todos los elementos de la lista en lugar de solo a aquellos que filtraría `c1`. Un ejemplo más sencillo, una aplicación que toma un valor dado por el usuario y realiza una consulta en una base de datos. Es un caso típico de uso de datos no confiables, lo normal es limpiar o validar los datos ingresados antes de hacer la consulta. Una posible implementación sería:

```
...
userQuery = getFromUser();
results = database.execute(cleanQuery(query));
```

```
1     MyStructure s = ...;
2     for (int elem : s.getIntValues()) ...
3    Δfor (int elem : s.intValues) ...
```

Figura 29: Acceso a campo mediante método *getter* vs acceso directo (Δ).

...

Un error clásico sería que el desarrollador olvide de usar `cleanQuery(query)` y directamente use `query`. Ambos casos responden a configuraciones particulares de *prvo*. El primer caso, restringir las expresiones objetivo a llamadas a métodos, involucradas en una expresión de navegación. El segundo caso responde a restringir las expresiones objetivo a llamadas a métodos en casos donde ésta ocurra en una condición, argumento u asignación, siempre que el tipo de retorno del método sea igual o compatible con el del argumento usado en la llamada. Para los ejemplos anterior se obtendrían los mutantes list .foreach(). if(c2).then(p1).else(p2) y:

```
...
userQuery = getFromUser();
results = database.execute(query);
...
```

Acceso directo a un campo

Un caso particular del tipo de fallas por omitir la llamada a un método, es el acceso a campos de clase de manera directa, en lugar de mediante el método *getter* asociado. Usualmente para cualquier atributo x de una clase al cual se desea dar acceso, se genera un método *getX()* del mismo tipo y conteniendo solo una sentencia de retorno *return x*. Sin embargo, existen casos donde el método *getter* realiza mas tareas que solo retornar. Un ejemplo muy simple es un método que retorna una colleción, cuando el campo asociado es nulo, el método se aseguraría de retornar una lista vacía, en este caso utilizar el método *getter* asegura que siempre vamos a obtener una colleción, mientras que acceder directamente al campo, nos puede retornar un valor nulo y causar errores como *NullPointerException* en Java. Un ejemplo de este tipo de fallas se muestra en la Figura 29, cuando *MyStructure#intValues* es *null*, el acceso directo generaría una *NullPointerException*.

Conversión de tipos

Muchas veces existen conversiones no visibles de tipos numéricos. Una división `2/3` no da lo mismo que `2/3.0`, aunque es difícil darse cuenta durante el desarrollo de un programa. Estas situaciones involucran en general *casteos* explícitos de tipos, `2/(float)3`, o uso de valores que especifícan claramente el tipo particular, `3.0f`. Fallas de este tipo no están representadas por operadores actuales, ya que en muchos casos el error, finalmente se produce por el acarreo de numerosas pérdidas de precisión. Aunque no relacionado con expresiones de navegación, este tipo de fallas artificiales se puede definir en base a restricciones sobre *prvo*, dada una expresión numérica, las posibles mutaciones son nuevas expresiones con el mismo valor pero distinto tipo.

4.4.3 *Fallas no asociadas a mutantes*

Las fallas en esta categoría, son aquellas que no pueden ser acopladas a operadores de mutación. Las razones que evitan esto se pueden dividir en casos donde no es posible definir un operador dado que sería necesario dar una definición por cada falla particular que se quiere representar. Y aquellos casos donde si bien es posible dar una definición general, como *reemplazar una llamada a un método por todas las posibles*, lleva a una cantidad intratable de mutantes. Incluso cuando estas fallas se definen como incapaces de ser acopladas a operadores de mutación, es decir, no se puede definir un operador que las represente de manera eficiente. Creemos que es posible mediante *prvo* poder representar algunos subconjuntos de las mismas.

Modificación o simplificación del algoritmo

Las fallas en este conjunto son aquellas que se dan por un algoritmo incorrecto en lugar de un algoritmo defectuoso. Mutation testing parte de asumir que el programador es competente y escribe programas cercanos a la solución correcta, si esto no se cumple, y el programa difiere significativamente, tal que se dificulta encontrar una versión "cercana" y correcta del programa, mutation testing se vuelve inaplicable. Esto no significa que no es posible construir una falla que represente estos casos, pero no es posible hacerlo de manera general, lo que imposibilita diseñar un operador que represente este conjunto de fallas. Pero si nos quedamos con el subconjunto de algoritmos implementados mediante *interfaces fluentes*, ahora llevamos el problema a realizar cambios en una expresión de navegación, y esto es precisamente lo que hace *prvo*. Como mencionamos, cualquier "operador" *prvo* es una configuración, que define restricciones sobre

que expresiones se van a modificar y como, siendo las expresiones generadas, válidas respecto a tipos. Si restringimos a utilizar solo expresiones pertenecientes a los tipos utilizados por una *interfaz fluente*, mientras el tipo del primer elemento de la expresión original, y el del último, se mantienen fijos, entonces podemos generar "programas" que trabajan sobre el mismo tipo de entradas que el original, y que retornan el mismo tipo que el original, y cuan diferentes estos "programas" pueden ser del original, depende en gran medida de los límites a cuanto se puede modificar el tamaño de la expresión original. En estos casos, modificar un algoritmo es totalmente posible, aunque es necesario controlar las restricciones impuestas para no caer en una explosión de mutantes a analizar.

Código no requerido

Existen fallas causadas por código que no debería estar, es decir, cuando la reparación asociada es eliminar código, ya sea una línea o varias (no necesariamente secuenciales). Por ejemplo:

```
...
if (c) breakProgram();
...
```

En donde la reparación no es una condición mal escrita sino directamente la eliminación de ciertas líneas de código. Si se quisiera representar este tipo de fallas sería necesario generar código e insertarlo, la complejidad de generar código válido y la cantidad que se puede generar, sumado a la cantidad de lugares en donde se puede insertar, hacen que este tipo de fallas no puedan ser representadas mediante mutación. Sin embargo, éste caso, es muy similar al anterior cuando se toman en cuenta como programas, a expresiones que utilizan *interfaces fluentes*.

Llamada a métodos similares

Si bien el tipo de fallas en donde se utiliza un método similar de una librería fue mencionado como acoplable a mutantes. Eliminar la restricción de que los métodos pertenezcan a la misma librería, causa que el espacio posible de mutantes se convierta en completamente inmanejable. Lo único que vale la pena destacar de este tipo de fallas en relación a *prvo*, es que es posible, mediante configuraciones, considerar un conjunto acotado de métodos que no necesariamente pertenezcan a la misma librería.

Violación de especificaciones

En mutación, el conocimiento que se tiene del código a mutar es, a lo sumo, contexto: que elementos son alcanzables en un determinado punto del código; tipos: cual es el tipo de una variable, el retorno de un método, los parámetros del mismo, etc. Pero no existe un conocimiento de que hace un determinado código, que precondiciones requiere y que postcondiciones asegura. Representar fallas causadas por el mal uso con respecto a invariantes, precondiciones y postcondiciones, es entonces imposible de conseguir. Sin embargo, un caso particular que creemos que *prvo* puede ser de gran utilidad, es en diseños basados en encadenar métodos (interfaces fluentes, builder pattern, entre otros). En estas situaciones el programador debe utilizar jerarquía de clases para definir una gramática, donde todo método involucrado tiene una precondición y postcondición basada en tipos. Con *prvo* es posible generar mutantes que permitan analizar si éstas son incumplidas.

5

PRVO y μJava++

5.1 μJAVA++

La herramienta *μJava* [Ma et al., 2005] funciona traduciendo código fuente *Java* a un *AST*, una estructura de árbol que representa el código original como nodos, y recorriendo el mismo mediante un patrón *visitor*. Cada operador sobreescribe métodos particulares de visita para los nodos objetivo, el operador *ROR*, por ejemplo, solo tiene por objetivo a expresiones binarias (en cuanto al tipo de nodos) y particularmente a aquellas que utilicen un operador relacional. En la Figura 30 se ve una parte de un *AST* correspondiente a la sentencia $x = y < 0$, *ROR* al visitar el nodo *BinExpr* con el operador $<$, va a generar varias mutaciones, entre otras, el nodo mutado *BinExpr[>]* que corresponde al cambio de el operador $<$ por $>$, mientras mantiene los subnodos de *BinExpr* sin modificaciones.

Una de las primeras tareas para poder desarrollar *prvo*, fue la reimplementación de esta herramienta. Hay dos herramientas involucradas, una es *OJ*, anteriormente conocida como *OpenJava*, que permite la transformación de código fuente *Java* en un *AST*, así como ofrecer clases para el recorrido y modificación de el mismo. Ésta es la base que utiliza *μJava*, y actualmente *μJava++*. Ambas herramientas fueron altamente modificadas, con *μJava* siendo

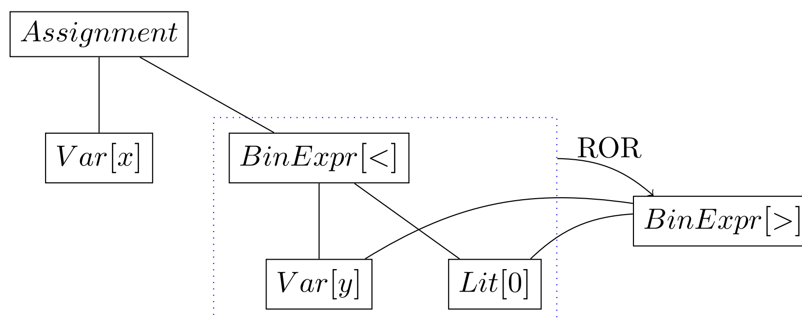


Figura 30: Ejemplo del operador *ROR* en funcionamiento

prácticamente reescrita por completo. Las razones para realizar dicha tarea tiene que ver con una serie de requerimientos que definimos a continuación:

5.1.1 *Representación*

Originalmente, las mutaciones no estaban representadas en *muJava*, los mutantes se generaban directamente, muchas veces como una responsabilidad del operador. Al diseñar *prvo*, observamos que era muy fácil generar mutaciones repetidas, tomemos por ejemplo la expresión a.b.c, es posible generar a.c eliminando la subexpresión del medio, pero también es posible al reemplazar las dos primeras por a. Al mismo tiempo, la variable especial this, permite que para todo campo de una clase, haya dos formas de referirse al mismo, una mediante el uso de this, y otra obviando el mismo. Por otro lado, *prvo* estaba siendo muy utilizado en el desarrollo de *Stryker*, herramienta de reparación automática mediante mutación y *SAT solving*, la cual se describe en la sección 7.1.3. La necesidad de contar con cierta información sobre las mutaciones para permitir ordenar e incluso ignorar algunas durante la búsqueda de un potencial arreglo, nos llevó a definir una representación para las mutaciones generadas mediante ternas `(original, mutante, operador)` en donde los dos primeros elementos representan, respectivamente, el nodo original y mutado del *AST* y el tercero denota el operador utilizado. Éstas se almacenan durante la generación de mutaciones hasta que se decida generar los mutantes, esto permite, entre otras cosas, analizar y evitar mutaciones repetidas.

5.1.2 *Anotaciones y control de generaciones*

Las herramientas de mutación, suelen no contar con un control sobre que partes de código se van a mutar, o éste es muy grueso, por ejemplo al permitir definir clases y métodos a mutar. Debido nuevamente a los comienzos orientados a reparación de *prvo*, necesitábamos un control más fino sobre donde se aplicarían las mutaciones. Lo que nos llevó, además de permitir definir que métodos se permiten mutar, a utilizar comentarios //mutGenLimit K luego de una sentencia para permitir que solo aquellas que tengan estas anotaciones y el K sea mayor a 0, sean mutadas.

A su vez, las anotaciones utilizadas son escritas en el mutante con el K asociado disminuído en 1. Esto permite, en casos como reparación o generación de mutantes compuestos (aquellos con más de una mutación aplicada), controlar las generaciones máximas por sentencia.

5.1.3 *Estandarización de componentes*

La versión original de la herramienta tenía implementaciónes muy diferentes por operador, algunos escribían directamente cada mutante (sin tener una representación para las mutaciones aplicadas), esto hacía que desarrollar nuevos operadores fuera una tarea compleja, nuestra versión permite desarrollar nuevos operadores con relativa facilidad. Durante el desarrollo de *prvo*, tuvimos en cuenta que en el futuro podríamos querer implementar nuevos operadores, y funcionalidades, lo que nos llevó a re-diseñar grandes partes de la herramienta. Gracias a esto, pudimos utilizar a *muJava++* para desarrollar otro operador de mutación, *BEE* [Gutiérrez Brida and Scilingo, 2017], centrado en generar mutaciones para fortalecer y debilitar expresiones condicionales de una manera más fina que operadores existentes, a su vez, pudimos implementar funcionalidades de análisis descriptas a continuación.

5.1.4 *Evaluación*

Mutation testing consiste en generar mutantes y evaluar cuantos son detectados por el conjunto de tests en evaluación. En la sección 3.2 se mencionan varias características que afectan a este criterio, en particular dificultad de detección y redundancia de mutantes. Nuestra versión de *μJava* provee ciertas mejoras en el análisis con respecto a estas propiedades, como ofrecer dos valores de *mutation score*, uno en donde se consideran los mutantes que no compilaron y otro donde no se los considera. Si bien, una evaluación inicial y muy básica de dificultad de detección, *muJava++* ofrece un análisis de "dureza" de los mutantes, al analizar, cuantos tests fue capaz de sobrevivir cada mutante. Con respecto a redundancia, un análisis más complejo que ofrece nuestra herramienta, es *dynamic mutant subsumption*, en donde se genera un grafo de subsuma de mutantes para la test suite particular, permitiendo evaluar que mutantes fueron redundantes y cuales son indispensables. Finalmente, para lidiar con la explosión de mutantes que se puede dar, y cómo el rendimiento es afectado por eso, se implementaron la capacidad de frenar la ejecución de los tests para un mutante cuando éste es detectado (perdiendo ciertos análisis que requieren datos completos de la ejecución de los tests), y el análisis en paralelo de mutantes para acelerar el proceso.

$$
\begin{aligned}
PRVO(x) &:= expression^{|0-1|} \\
&:= x.expression^{|0|} \\
&:= expression^{|0|}.x \\
&:= null \\
\\
PRVO(x.y) &:= x \\
&:= y \\
&:= expression^{|0-1|}.y \\
&:= y.expression^{|0-1|} \\
&:= expression^{|0|} \\
&:= x.y.expression^{|0|} \\
&:= x.expression^{|0|}.y \\
&:= expression^{|0|}.x.y
\end{aligned}
$$

Figura 31: Definición base de la implementación de *prvo*

5.2 PRVO

La definición general de *prvo*, dada en 4.3, no es efectiva en la práctica, desde ya por que permite entre otras cosas, infinitas mutaciones. Si bien mencionamos, e incluso presentamos a *prvo* como un meta-operador altamente configurable, es necesario establecer algunas restricciones básicas, ya que mientras más flexible es un operador, más compleja es su implementación. Por esta razón, es que decidimos fijar ciertas restricciones iniciales, basadas en la evolución de *prvo* durante su desarrollo, tanto en su uso en reparación, como en mutation testing.

Inicialmente *prvo* comienza como una necesidad en reparación automática de programas mediante mutación y *SAT solving*. Utilizando la herramienta *Stryker*, *prvo* provee el potencial de reparar ciertos defectos no reparables anteriormente mediante mutantes. Como un problema que se menciona en 7, la explosión de candidatos en el espacio de búsqueda es uno de los principales obstáculos en reparación automática basada en mutación. Por esto, *prvo* comienza con una serie de restricciones sobre el tamaño de las expresiones mutadas, la diferencia entre el tamaño original y final de una expresión no puede superar a `1`. Estas restricciones llevan a una gramática ligeramente distinta a la original.

En la Figura 31 se restringe la definición abstracta de *prvo* [23] en base a cuanto se puede cambiar la longitud de la expresión original. La notación $expression^{|R|}$ denota una expresión de un tamaño particular o un rango de tamaños, por ejemplo, de 0 a 1. El valor *null* es un caso especial, no es po-

sible incrementar el tamaño de una expresión encadenando nuevos elementos ni tampoco reemplazar *null* por una expresión de tamaño 1, solo es posible reemplazarlo por otra expresión de tamaño 0.

5.2.1 *Configuración de PRVO*

Con el tiempo la cantidad de comportamientos o restricciones configurables en *prvo* fue aumentando. En esta sección solo mencionaremos un conjunto de éstas, ya que varias fueron implementadas para casos muy particulares.

Restricciones de tamaño

A partir de la definición en la Figura 31, las restricciones sobre cuanto puede decrementar o incrementar una expresión respecto al uso de navegación se limitan a:

Reemplazar un elemento (R) Ciertas mutaciones de *prvo* solo modifican elementos unitarios (expresiones de tamaño 0) en una expresión, reemplazándolos por otros de tamaño 0.

Añadir un elemento (A) Una expresión puede incrementarse por 1 su tamaño, al insertar un elemento tanto al principio, al final, o intercalarlo en su interior.

Eliminar un elemento (D) En una expresión de navegación, se puede eliminar un elemento, ya sea al principio, final, o en el resto de la expresión. No es posible eliminar un elemento en una expresión que no hace uso del operador de navegación (tiene tamaño 0), ya que no hay ninguna subexpresión para remover.

Intercambiar dos elementos por uno (2x1) Dada una expresión de navegación, dos elementos contiguos, es decir, una subexpresión de navegación de tamaño 1, puede ser reemplazada por un único elemento, es decir, una expresión de tamaño 0.

Intercambiar un elemento por dos (1x2) En una expresión, un elemento de la misma puede ser reemplazado por dos elementos contiguos, es decir, una expresión de navegación de tamaño 1.

En el ejemplo siguiente pueden verse algunas de las distintas mutaciones que corresponden a las restricciones anteriores, en algunos casos el mismo mutante puede ser logrado por distintas mutaciones:

```
current = current.next; //original
δcurrent = header; //2x1
δcurrent = current; //2x1 o D
δcurrent = current.next.next; //1x2 o A
δcurrent = header.previous.next; //1x2 o A
previous = current; //original
δprevious = header; //R
δprevious = null; //R
δprevious = current.next; //1x2 o A
δprevious = header.next; //1x2
```

En las Figuras 32, 33, 34, 35, y 36, se muestran versiones muy simplificadas de los métodos de *prvo* para generar mutaciones (reemplazo por literales por ejemplo no se muestra). Métodos auxiliares utilizados por éstos incluyen a:

generateMutant(Expression, Expression)
Este método toma la expresión original (primer parámetro) y la reemplaza por la expresión mutada (segundo argumento).

append(Expression...)
Este método toma una secuencia de expresiones y las une mediante el operador de navegación (*punto*), los argumentos nulos son ignorados.

isReachable(Expression, Expression)
Este método verifica si la primera expresión es alcanzable desde la segunda, es decir, si el primer elemento de la primera expresión representa un miembro del tipo del último elemento de la segunda. Por ejemplo *next.value* es alcanzable desde *this.header* en una lista (ver Figura 22).

elements(Expression)
Retorna una lista de todos los elementos que conforman una expresión con respecto al operador de navegación. Por ejemplo, la expresión *current.next.value* retornaría una lista con *current*, *next*, y *value*.

getType(Expression)
Este método retorna el tipo asociado al último elemento de una expresión. Por ejemplo, para la expresión *current.next* retornaría *Node* (ver Figura 22).

compatibleTypes(Type, Type)
Este método verifica si el primer tipo es compatible con el segundo. La compatibilidad será verificada de acuerdo a la configuración de *prvo*, si el chequeo de tipos es relajada, entonces se verifica si el primer tipo es

asignable al segundo; si por el contrario se está utilizando un chequeo de tipos estricto, entonces solo son compatibles los tipos iguales.

fieldsAndMethodsFrom(Expression)
Retorna todos los miembros accesibles desde el último elemento de una expresión (no nula).

reachableVarsFieldsAndMethods()
Retorna todas las variables, campos y métodos que son alcanzables desde el punto actual.

nextExpression(Expression)
Dada una expresión retorna *null* si la expresión no estaba conectada mediante el operador de navegación a otra, o retorna la expresión a la cual está conectada mediante el operador de navegación en caso contrario. Cabe destacar que si bien el método *elements(Expression)* retorna elementos desconectados, éstos tienen asociados a que expresiones estaban previamente conectados.

previousExpression(Expression)
Equivalente a *nextExpression(Expression)* pero retornando la expresión a la cual ésta estaba conectada (o *null* si no lo estaba).

lastExpressionFrom(Expression)
Retorna el último elemento en una expresión, el cual es o la misma expresión original si ésta no es de navegación, o el último miembro en caso contrario. Por ejemplo *lastExpressionFrom(current.next.value)* retornaría *value*.

Restricciones de puntos de mutación

El tipo de expresiones que muta *prvo* puede encontrarse en una gran cantidad de lugares distintos en el código fuente. Si bien en el ejemplo anterior las expresiones mutadas fueron siempre sobre asignaciones y solo se aplicaban mutaciones a la parte derecha de la misma, *prvo* cuenta actualmente con las siguientes restricciones configurables:

Parte izquierda de asignaciones En una asignación `a = b`, *prvo* puede o no generar mutaciones para `a`. Es necesario destacar que estas mutaciones no se aplican en declaraciones de variables, es decir, expresiones `T a = b`, dado que modificar `a` en este caso tendería a generar mutantes que no compilan. Si bien es cierto que existen fallas en donde un desarrollador puede olvidar declarar una variable local, y de esta forma hacer mención a

```
public void sameLength(Expression e, Type complyWith) {
    if (e.size() == 0) {
        List<Expression> rs = reachableVarsFieldsAndMethods();
        for (Expression r : rs) {
            Type rType = getType(r);
            if (compatibleTypes(rType, complyWith) {
                generateMutant(e, r);
            }
        }
        return;
    }
    for (Expression sub : elements(e)) {
        Expression previous = previousExpression(sub);
        Expression next = nextExpression(sub);
        List<Expression> rs;
        if (previous == null) {
            rs = reachableVarsFieldsAndMethods();
        } else {
        }  rs = fieldsAndMethodsFrom(previous);
        for (Expression r : rs) {
            if (next == null) {
                Type rType = getType(r);
                if (compatibleTypes(rType, complyWith)) {
                    generateMutant(e, e.replace(sub, r);
                }
            } else if (isReachableFrom(next, r)) {
                generateMutant(e, e.replace(sub, r));
            }
        }
    }
}
```

Figura 32: Método de generación de mutantes de *prvo* para generar expresiones donde el tamaño de la misma se mantiene.

```
public void increaseLength(Expression e, Type complyWith) {
    Expression curr = lastExpressionFrom(e);
    boolean finish = false;
    while (!finish) {
        Expression next = nextExpression(curr);
        Expression prev = previousExpression(curr);
        List<Expressions> rs;
        if (prev == null) {
            rs = reachableVarsFieldsAndMethods();
        } else {
        }   rs = fieldsAndMethodsFrom(curr);
        for (Expression r : rs) {
            if (next == null) {
                Type rType = getType(r);
                if (compatibleTypes(rType, complyWith)) {
                    Expression mutant = append(prev, curr, r);
                    generateMutant(e, mutant);
                }
            } else if (isReachable(next, r)) {
                Expression mutant = append(prev, curr, r, next);
                generateMutant(e, mutant);
            }
        }
        if (curr == null)
            finish = true;
        else
            curr = prev;
    }
}
```

Figura 33: Método de generación de mutantes de *prvo* para generar expresiones donde el tamaño de la misma se incrementa añadiendo un nuevo elemento a la expresión.

```
public void decreaseLength(Expression e, Type complyWith) {
    if (sizeOf(e) < 1) return;
    for (Expression sub : elements(e)) {
        Expression prev = previousExpression(sub);
        Expression next = nextExpression(sub);
        if (next == null) {
            Type prevType = getType(prev);
            if (compatibleType(prevType, complyWith)) {
                generateMutant(e, prev);
            }
        } else if (isReachableFrom(prev, next)) {
            Expression mutant = append(prev, next);
            generateMutant(e, mutant);
        }
    }
}
```

Figura 34: Método de generación de mutantes de *prvo* para generar expresiones donde el tamaño de la misma se decrementa eliminando un elemento de la expresión.

un atributo/campo de clase, este tipo de fallas no corresponde con las que se desea representar mediante *prvo*.

Parte derecha de asignaciones En una asignación `a = b`, *prvo* puede o no generar mutaciones para `b`.

Sentencias de retorno y expresiones internas El operador *prvo* es capaz de mutar expresiones asociadas a sentencias de retorno (`return e`), al mismo tiempo que expresiones unarias y binarias encontradas en asignaciones `(a = e && f)`, en sentencias condicionales `(while(c) ...)`, argumentos de métodos `foo(x)`, e incluso en las distintas partes de sentencias `for` como la inicialización, condición e incremento.

Restricciones de expresiones a utilizar

Incluso luego de configurar restricciones sobre cuanto disminuir o incrementar una expresión, y en que partes del código aplicar *prvo*. La cantidad de expresiones válidas disponibles para mutar la expresión, sigue siendo demasiado grande. Esto trae aparejado problemas de eficiencia (la cantidad de mutantes afecta los recursos necesarios para mutation testing), generación de mutantes triviales o con poca dificultad para ser detectados, y mutantes redundantes.

```
public void twoByOne(Expression e, Type complyWith) {
    if (sizeOf(e) < 1) return;
    Expression curr = lastExpressionFrom(e);
    boolean finish = false;
    while (!finish) {
        Expression next = nextExpression(curr);
        Expression prev = previousExpression(curr);
        Expression prevPrev = lastExpressionFrom(prev);
        List<Expressions> rs;
        if (prevPrev == null) {
            rs = reachableVarsFieldsAndMethods();
        } else {
        }   rs = fieldsAndMethodsFrom(prevPrev);
        for (Expression r : rs) {
            if (next == null) {
                Type rType = getType(r);
                if (compatibleTypes(rType, complyWith)) {
                    Expression mutant = append(prevPrev, r)
                    generateMutant(e, mutant);
                }
            } else if (isReachable(next, r)) {
                Expression mutant = append(prevPrev, r, next)
                generateMutant(e, mutant);
            }
        }
        if (prevPrev == null)
            finish = true;
        else
            curr = prev;
    }
}
```

Figura 35: Método de generación de mutantes de *prvo* para generar expresiones donde el tamaño de la misma se decrementa reemplazando dos elementos de una expressión por uno.

```
public void oneByTwo(Expression e, Type complyWith) {
    for (Expression sub : elements(e)) {
        Expression prev = previousExpression(sub);
        Expression next = nextExpression(sub);
        List<Expression> rs;
        if (prev == null) {
            rs = reachableVarsFieldsAndMethods();
        } else {
        }   rs = fieldsAndMethodsFrom(curr);
        for (Expression r : rs) {
            List<Expression> rs2 = fieldsAndMethodsFrom(r);
            for (Expression r2 : rs2) {
                if (next == null) {
                    Type r2Type = getType(r2);
                    if (compatibleTypes(r2Type, complyWith)) {
                        Expression mutant = append(prev, r, r2);
                        generateMutant(e, mutant);
                    }
                } else if (isReachable(next, r2)) {
                    Expression mutant = append(prev, r, r2, next);
                    generateMutant(e, mutant);
                }
            }
        }
    }
}
```

Figura 36: Método de generación de mutantes de *prvo* para generar expresiones donde el tamaño de la misma se incrementa reemplazando un elemento de una expressión por dos.

La expresión Object obj = current.next;, puede dar lugar a una enorme cantidad de mutaciones válidas, dado que no solo cualquier tipo no primitivo hereda de `Object`, sino que además, *Java* permite "autoboxing" de tipos primitivos. Esta técnica permite que expresiones como Integer i = 1; sean válidas e internamente se traducen a Integer i = new Integer(1);. Volviendo al ejemplo anterior podemos ver que es posible generar una gran cantidad de mutaciones completamente válidas desde un punto de vista de tipos, pero completamente independientes de los tipos que pueden ser de interés o que están involucrados en la expresión original. Ejemplos de éstas son:

```
Object obj = current.toString();
Object obj = current.getClass();
Object obj = current.hashCode();
Object obj = current.toString().length;
Object obj = current.next.toString();
...
```

Por esto, *prvo* cuenta con las siguientes restricciones configurables:

Métodos y campos restringidos Es posible definir expresiones regulares para restringir el uso de ciertos métodos y campos, por ejemplo:

```
java\\.lang\\.String\\#.*
```

restringe cualquier método y campo de *java.lang.String*. Durante la generación de mutaciones, *prvo* verifica que cualquier método o campo que vaya a utilizar para mutar una expresión, no esté restringida, la verificación se hace sobre el nombre completo de un miembro de clase, el cual se define con *<nombre completo de la clase>#<nombre del miembro>[(<tipos de los argumentos separados por coma>)]* por ejemplo *java.lang.String#substring(int, int)*.

Métodos y campos permitidos En muchos casos, la cantidad de métodos y campos a restringir es muy grande y solo se quieren permitir aquellos que pertenezcan a ciertas clases, por ejemplo, en el caso de una lista puede solo desearse utilizar aquellos que pertenezcan a la clase *Lista* y *Nodo*. Es posible definir, de la misma forma que en el caso anterior, que métodos y campos son permitidos para que *prvo* los utilice durante la generación de mutaciones. Cabe destacar que estas opciones son excluyentes, no es posible utilizar ambas al mismo tiempo.

Control de tipos En lenguajes orientados a objetos como *Java*, la herencia de clases permite que dos tipos sean compatibles aún cuando no son iguales. Dados dos tipos, `A` y `B`, tal que el segundo hereda del primero.

La asignación A a = new B(); es válida mientras que la inversa no lo es. Una expresión mutada por *prvo* tiene que ser correcta con respecto a tipos, pero con esto podemos cambiar expresiones como list .node.value a list .toString (). getClass() si `value` era de tipo `Object`. Por esto, *prvo* puede restringirse a utilizar un control estricto de tipos, en donde éstos solo se consideran compatibles sin son exactamente iguales.

Uso de literales En ciertos casos el uso de literales conforman expresiones válidas para mutar una expresión, var.toString (). length puede cambiarse a "". length. Cualquier operador que utilice valores literales, va a requerir un conjunto finito de los mismos. En el caso de *prvo*, utiliza un conjunto de literales base, *1, 0, True, False, "", null* en donde cada uno puede habilitarse o no; a su vez permite la búsqueda de literales alcanzables desde el punto en donde se está mutando; también es posible habilitar/deshabilitar la generación de variaciones en literales numéricos, es decir que para cada literal que pertenezca a un tipo primitivo numérico, se van a crear copias del mismo para los otros tipos. Por ejemplo, para `2`, un literal de tipo `int` se van a crear las variantes `2.0d`, `2.0f`, y `2l`, de tipos *double*, *float*, y *long* respectivamente.

Uso de campos estáticos El modificador *static* en *Java* define a un miembro que es compartido por todas las instancias de la clase que lo define. Estos miembros estáticos no deberían accederse mediante instancias de una clase, aunque hacerlo es posible y no representa código inválido. Sin embargo si ésto se permite, se pueden obtener muchas mutaciones asociadas al uso de constantes estáticas que no fueron restringidas mediante otra configuración. Por eso, *prvo* permite la restricción del uso de miembros estáticos en un contexto no estático.

5.3 DYNAMIC MUTANT SUBSUMPTION

El uso de *mutation score*, la relación de mutantes detectados sobre el total, como métrica de evaluación para test suites, trae varios problemas sobre que conclusiones se pueden derivar del mismo. Entre éstos, hemos mencionado el detectar mutantes triviales, incrementando el valor de mutation score sin implicar una mejora en la capacidad de detectar fallas por parte de la test suite; la existencia de mutantes que son semánticamente equivalentes al programa original, y por lo tanto imposibles de detectar, bajando el valor del mutation score sin significar una menor capacidad de detectar fallas; acoplamiento, es decir, que tan bien los mutantes representan fallas reales; y finalmente la generación de mutantes redundantes con respecto a su capacidad de ejercitar

los tests bajo evaluación. Estos problemas hacen que el análisis de *mutation testing* requiera cuidado al interpretar los resultados, y sirve de argumento para la realización de estudios más profundos para dar mayor validez a los mismos. Principalmente al utilizar mutation testing para comparar técnicas de testing, un problema reportado por Papadakis et al [Papadakis et al., 2016]. Los estudios para evaluar las propiedades que afectan al mutation score añaden complejidad al análisis y, en la mayoría de los casos, requiere resolver problemas indecidibles o altamente complejos. Como la meta de este trabajo es evaluar un nuevo operador de mutación, cuan significativo es el valor de mutation score se torna aún más importante, sin embargo, las propiedades que afectan al mismo resultan más ricas en cuanto a la evaluación que el valor en sí. *Dynamic Mutant Subsumption* se presentan como una relación que permite definir mutantes redundantes (innecesarios para la evaluación de un test suite) y mutantes indispensables. Dos resultados principales para sostener la utilización de esta relación para evaluar a *prvo* son [Ammann et al., 2014], donde se presenta el uso de esta relación como forma de determinar conjuntos minimales de tests, es decir, subconjuntos del conjunto original de tests que eliminan a aquellos tests que son redundantes (eliminarlos no elimina la capacidad de detección de fallas) a travÃľs de la determinación de conjunto minimales de mutantes, precisamente obtenidos mediante *Dynamic Mutant Subsumption* al eliminar todos los mutantes subsumidos. Que un operador tienda a generar mutantes que pertenezcan a los conjuntos minimales representa una muy buena métrica para evaluar la importancia del mismo. El otro de los resultados que utilizamos de apoyo para el uso de esta relación como métrica principal para evaluar a *prvo* es [Papadakis et al., 2016], donde muestran el efecto negativo que tienen los mutantes redundantes, aquellos que son subsumidos, al comparar técnicas de testing. Esto lleva a que la utilización de este análisis (*Dynamic Mutant Subsumption*) es entonces una buena forma de evaluar la utilidad de los mutantes generados por *prvo* con respecto a su capacidad para evaluar tests. Recordemos primero la definición de *Mutant Subsumption* antes de discutir sobre la más acotada definición de *Dynamic Mutant Subsumption*.

> Para dos mutantes m_1 y m_2, producidos a partir del programa original p. Se define la relación m_1 subsume a m_2, si existe al menos un test para el cual el comportamiento de m_1 difiere del de p, y para todo test t para el cual el comportamiento entre m_1 y p difiere, el comportamiento entre m_2 y p también debe hacerlo.

Un punto importante es que la definición anterior habla del universo de tests, todo test posible, lo cual hace de *Mutant Subsumption* un problema indecidible salvo para casos triviales (problemas con un conjunto finito de posibles escenarios). De todas formas, es posible una resolución acotada a este problema. Antes

de pasar a la implementación de este análisis, es necesario mostrar cual es la utilidad del mismo. La intuición es que si un mutante es detectado por una gran cantidad de tests mientras otros son detectados por subconjuntos que detectan al primero, entonces detectar cualquiera de éstos lleva a detectar también al primero, evaluando varias veces los mismos tests. Por otro lado, un mutante que requiere tests más específicos para ser detectado genera un mejor "feedback" sobre los tests, que uno que es detectado por casi cualquiera. Como mencionamos, en la práctica no es posible hacer un análisis basado en la definición anterior, por lo que la siguiente definición es utilizada.

> Para dos mutantes m_1 y m_2, producidos a partir del programa original p, y un conjunto de tests T. Se define la relación m_1 subsume dinámicamente a m_2, si existe al menos un test en T para el cual el comportamiento de m_1 difiere del de p, y para todo test t en T para el cual el comportamiento entre m_1 y p difiere, el comportamiento entre m_2 y p también debe hacerlo.

Esta definición da a lugar a *Dynamic Mutant Subsumption*, y es la que se implementó en *μJava++* y que va a ser utilizada como evaluación principal, en este caso para el conjunto de mutantes generados en cada experimento.

5.3.1 *Dynamic Mutant Subsumption Graph*

A partir del análisis anterior, es posible construir un grafo en el cual un nodo contiene todos los mutantes equivalentes respecto a subsuma, es decir, para cada par de mutantes m_1 y m_2 en el nodo, m_1 subsume a m_2 y viceversa. Una arista direccional conectando el nodo n_1 con el nodo n_2 representa la relación de subsuma:

> Para cada mutante m_i en el nodo n_1, y para cada mutante m_j en el nodo n_2, se cumple que m_i subsume a m_j.

Un ejemplo de la generación de estos grafos se puede ver comenzando con la Tabla 2 la cual es generada al final del análisis de mutation testing, cada mutante analizado se almacena en un nodo único con la información de que tests detectaron al mismo. A partir de estos datos se eliminan los nodos que corresponden a mutantes que sobrevivieron (recordemos que la primera condición para la relación de subsuma es que los mutantes en la relación hayan sido detectados) y todos los nodos que corresponden a mutantes detectados por los mismos tests, es decir, se subsumen el uno al otro, se fusionan en un mismo nodo, esto se puede apreciar en la Tabla 3 donde por ejemplo el nodo **4** fue

eliminado, el nodo **7** fue fusionado con el nodo **2** y lo mismo ocurrió con los nodos **3** y **5**. A partir de la última tabla (o a partir de los nodos representados en la misma) se puede construir el grafo de subsuma dinámica de la Figura 37. A partir de este grafo se observan nodos que son subsumidos pero no subsumen a nadie, a los mutantes en estos nodos se los considera los más redundantes de todos. Nodos que subsumen a otros y son a su vez subsumidos, éstos siguen siendo redundantes pero en un grado menor a los anteriores. Finalmente los nodos que subsumen a otros pero no son a su vez subsumidos, son denominados "dominadores". Dado que en esta tesis presentamos un nuevo operador, y que su evaluaci'on con respecto al conjunto de operadores suficientes corresponde al objetivo principal de nuestra evaluación, lograr que los mutantes generados por *prvo* ocupen una mayoría de los nodos dominadores, no alcanza, esto es causa de que cualquier mutante generado por otro operador e incluído en el nodo, es equivalente a los de *prvo*. Este problema se extiende a la evaluación de cualquier operador utilizando grafos de dynamic mutant subsumtion. Por esto, este análisis incluye la propiedad de pureza en los nodos.

> Un nodo en un grafo de dynamic mutant subsumption es considerado puro, si para cada mutante contenido en el mismo, éste fue generado por el mismo operador.

En la práctica éstos grafos son imprácticos de utilizar por ser demasiado grandes como para poder extraer información del mismo. Sin embargo la estructura de grafos permite fácilmente definir los conceptos de nodos dominadores y redundantes. Una clara desventaja de este análisis es que requiere ejecutar todos los tests por cada mutante, contrariamente al análisis básico de mutation testing (cuando solo interesa obtener el valor de mutation score) que solo se ejecutan todos los tests para un mutante para aquellos que sobreviven, pero solo siendo necesario ejecutar los tests hasta encontrar el primero que detecta al mutante en los casos donde éste no sobrevive.

5.3.2 *Dynamic Mutant Subsumption como métrica*

En la sección anterior presentamos las propiedades deseables en los mutantes generados por un operador de mutación, y algunas formas de evaluar el grado en que éstas se cumplen. Sin embargo, al evaluar operadores de mutación con respecto a otros, o técnicas de selección de mutantes, la utilización de métricas como diferencia entre mutation score, cantidad de mutantes equivalentes generados (o evitados), "dureza" de mutantes (cuantos tests logra sobrevivir el mutante antes de ser detectado) y cantidad de mutantes, son las más utilizadas.

Node	Mutants	Test1	Test2	Test3	Test4	Test5	Test6
1	m1	•		•	•		•
2	m2	•		•			
3	m3	•	•	•			
4	m4						
5	m5	•	•	•			
6	m6	•			•		•
7	m7	•		•			
8	m8			•			

Tabla 2: Nodos iniciales con un solo mutante asociado y los tests que detectan al mismo.

Node	Mutants	Test1	Test2	Test3	Test4	Test5	Test6
1	m1	•		•	•		•
2	m2, m7	•		•			
3	m3, m5	•	•	•			
6	m6	•			•		•
8	m8			•			

Tabla 3: Nodos equivalentes fusionados y nodos que representan mutantes sobrevivientes eliminados.

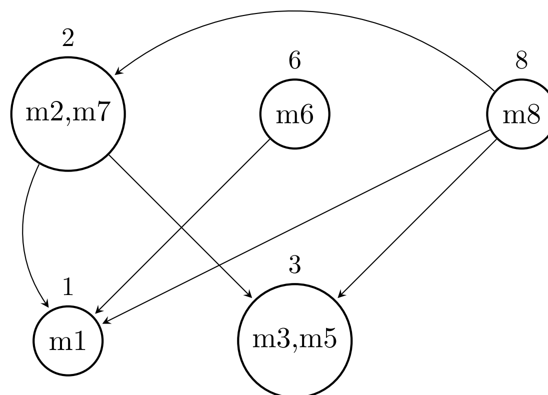


Figura 37: Grafo de *Dynamic Mutant Subsumption* generado mediante la información de la Tabla 3

En esta tesis se plantea utilizar *Dynamic Mutant Subsumption* como una de las métricas principales para evaluar a *prvo* con respecto al conjunto de operadores suficientes implementados en *µJava*. La razón de esto se debe a la cantidad de información que permanece oculta al utilizar métricas usuales.

Para utilizar un caso particular sobre el cual discutir las desventajas de las métricas usuales vamos a referirnos a un trabajo publicado sobre un operador de mutación para fortalecer y debilitar expresiones booleanas [Gutiérrez Brida and Scilingo, 2017]. Éste surge de la intuición de que los operadores que afectan expresiones booleanas al modificar operadores (*ROR*, *COR*, *COI*, entre otros) o reemplazar partes de la expresión, como el caso de *ROR* que reemplaza una expresión boolena por las constantes *True* y *False*, generan cambios "muy gruesos" como cambiar una condición de **a > b** por **a < b**, mientras que una modificación del estilo **a > b && c** nos parece que puede llevar a mutantes más sutiles. En este estudio utilizamos el mismos conjunto de operadores contra los que comparamos *prvo*, y evaluamos como la utilización de *BEE* afectaba a las siguientes métricas:

MUTATION SCORE
: El decremento en el valor obtenido de mutation score al utilizar a *BEE* serviría como indicador de que mutantes más difíciles fueron generados. Mientras que un incremento indicaría el aumento de mutantes más triviales de detectar.

CANTIDAD DE MUTANTES
: Dado que en el peor caso es necesario ejecutar todos los tests para cada mutante (suponiendo un análisis solo centrado en mutation score) los recursos utilizados por mutation testing, entre ellos tiempo, están directamente relacionados a la cantidad de mutantes generados, todo operador debería ser evaluado con respecto al costo asociado, es decir, cuantos mutantes agrega al análisis.

DUREZA (TOUGHNESS)
: La cantidad de tests que un mutante es capaz de "sobrevivir", no ser detectado por, puede servir como una métrica para evaluar cuan difícil de detectar resulta. Analizar como cambia la dureza promedio de los mutantes detectados al agregar un operador (en este caso *BEE*) permite evaluar la dificultad de detección de los mutantes agregados por el mismo.

MUTANTES EQUIVALENTES
: Una característica negativa de un operador es cuantos mutan-

tes equivalentes produce, éstos son mutantes que si bien son distintos sintácticamente, son semánticamente equivalentes al programa original, y por lo tanto imposibles de detectar.

MUTANTES DIFÍCILES DE MATAR (STUBBORN)
: La generación de mutantes difíciles de matar (stubborn) es una característica altamente deseable para un operador de mutación. Sin embargo representa una propiedad muy compleja para analizar, principalmente por la dificultad en dar una definición apropiada, una de éstas corresponde a la propuesta por Xiangjuan Yao et al. [Yao et al., 2014]: un mutante difícil de matar es aquel para el cual existe un tests que es capaz de detectarlo pero no está presente en un test suite "suficientemente bueno" el cual luego define como un test suite con 100 % de cobertura de ramas. Aunque finalmente se utilizan definiciones más relajadas como la utilización de testing exhaustivo acotado (considerar todas las entradas dentro de ciertas cotas) o generación de test suites mediante herramientas como *EvoSuite* con una cobertura de ramas por encima del 80 %.

La evaluación presentada en [Gutiérrez Brida and Scilingo, 2017] utiliza estas métricas para la evaluación del nuevo operador, y si bien los resultados no fueron negativos, éstos no parecen mostrar una clara ventaja de utilizar el operador, aunque la motivación detrás del mismo así como su definición, parecería indiciar que el mismo tiene potencial. Un problema que tienen estos resultados es la cantidad de información que permanece oculta. Como mencionamos en la sección 3.2, el valor de mutation score se ve altamente afectado por: mutantes equivalentes, que disminuyen el valor sin significar una baja calidad de parte de los tests; mutantes triviales, que aumentan el valor mediante mutaciones que son triviales de detectar y no resultan en una evaluación apropiada de los tests; acoplamiento entre mutantes, cuando la detección de una mutación implica la detección de otra o cuando dos mutantes son equivalentes entre sí incrementan el mutation score por (en cierto modo) mutantes repetidos y lo mismo puede causar un decremento repetido en este valor.

El análisis de mutantes equivalentes es un problema indecidible, análisis manuales siguen siendo uno de los enfoques más utilizados aunque sigue siendo una propiedad comúnmente evaluada y con una gran importancia en comparación de operadores de mutación o técnicas de selección de mutantes.

La dificultad de un mutante por el otro lado, es una propiedad que si bien sería interesante evaluar, resulta muy difícil de definir, Visser presenta en [Visser,

2016] un estudio sobre la dificultad de detectar mutantes el cual utiliza *model counting* para evaluar la cantidad de escenarios (de manera exhaustiva, en un conjunto acotado) que logran diferenciar a un mutante del programa original. Este enfoque no resulta práctico, algo que es mencionado en el trabajo anterior, y solo es posible en programas muy simples.

El uso de *Dynamic Mutant Subsumption* a sido utilizado previamente para minimizar conjuntos de mutantes en [Ammann et al., 2014] y para medir la efectividad de un conjunto de mutantes en [Gopinath et al., 2016], este último trabajo sugiere la utilización de dynamic mutant subsumption como forma de comparar operadores de mutación y/o técnicas de selección de mutantes. Una ventaja inicial de utilizar esta técnica es que la definición es clara, un mutante subsume a otro si ambos son detectados y los tests que detectan al primero también detectan al segundo. A su vez, la conclusión de que los mutantes subsumidos son redundantes se relaciona a la existencia de otro/s mutantes que ejercitan al conjunto de tests de manera más específica (son detectados por menos tests), y los mutantes equivalentes con respecto a subsuma ejercitan varias veces al mismo conjunto de tests, pudiendo elegir un solo representante de éstos. La dificultad de detección puede asociarse a subsuma argumentando que si un mutante subsume a otros, es detectado por menos tests, dando la intuición de que requiere tests más específicos que los mutantes que subsume.

Retomando el estudio sobre el operador *BEE*, cuando realizamos un análisis de subsuma logramos obtener resultados mas significativos con respecto a la utilidad del operador para mutation testing. Las preguntas de investigación de este trabajo son:

> *Nuestro operador (BEE) genera mayor cantidad de mutantes difíciles de matar (stubborn) que los operados existentes que debilitan o fortalecen expresiones booleanas?*
>
> *Como nuestro operador (BEE) afecta al análisis de mutación?*

Siendo los resultados obtenidos los de las Tablas 4, 5, 6, 7, y 8. Estos resultados dieron lugar a la conclusión de que la respuesta a la primer pregunta de investigación es negativa (*BEE* no parece generar una cantidad importante de mutantes difíciles de matar), mientras que la segunda pregunta se responde de una manera más positiva al indicar que la dificultad de matar los mutantes generados por *BEE* parece mantenerse con respecto al resto de los mutantes (el valor de *toughness* prácticamente no cambia) y que la cantidad de mutantes equivalentes generados se deben a particularidades de algunos casos. Un problema con los resultados utilizados es que parecen estar ocultando información. Por esto,

	Mutantes	Sobrevivientes	Toughness (killed)	Mutation score	Stubborn
SIN BEE	196	14	0.50	92.85 %	6
CON BEE	336	36	0.53	89.28 %	8

Tabla 4: Datos de evaluación de *BEE* para *ArrayPartition* de acuerdo al trabajo presentado en [Gutiérrez Brida and Scilingo, 2017]

	Mutantes	Sobrevivientes	Toughness (killed)	Mutation score	Stubborn
SIN BEE	454	111	0.73	75.55 %	65
CON BEE	889	341	0.72	61.64 %	163

Tabla 5: Datos de evaluación de *BEE* para *BinaryHeap* de acuerdo al trabajo presentado en [Gutiérrez Brida and Scilingo, 2017]

al realizar un análisis de los mismos casos utilizados en el trabajo publicado, pero utilizando Dynamic Mutant Subsumption, obtenemos los resultados en la Figura 38 que permiten observar que en tres de los cinco casos de estudio (*BinaryHeap*, *OrdSet*, y *TreeMap*) los mutantes generados por *BEE* son dominadores, es decir, corresponden a mutantes no redundantes para evaluar a los tests. En los casos de *BinaryHeap* y *OrdSet* se observa que al menos el 50 % de los mutantes dominadores generador por *BEE* no son equivalentes a otros mutantes dominadores, lo que permite concluir que representan fallas nuevas no generadas por los otros operadores.

Esta comparación sirve de argumento para utilizar Dynamic Mutant Subsumption como una de las principales técnicas para evaluar un operador de mutación.

	Mutantes	Sobrevivientes	Toughness (killed)	Mutation score	Stubborn
SIN BEE	1278	179	0.83	85.99 %	160
CON BEE	2248	620	0.83	72.41 %	403

Tabla 6: Datos de evaluación de *BEE* para *OrdSet* de acuerdo al trabajo presentado en [Gutiérrez Brida and Scilingo, 2017]

	Mutantes	Sobrevivientes	Toughness (killed)	Mutation score	Stubborn
SIN BEE	576	160	0.93	72.22 %	114
CON BEE	2536	1376	0.94	45.74 %	631

Tabla 7: Datos de evaluación de *BEE* para *TreeMap* de acuerdo al trabajo presentado en [Gutiérrez Brida and Scilingo, 2017]

	Mutantes	Sobrevivientes	Toughness (killed)	Mutation score	Stubborn
SIN BEE	435	109	0.98	74.94 %	56
CON BEE	55	115	0.98	79.27 %	62

Tabla 8: Datos de evaluación de *BEE* para *TriTyp* de acuerdo al trabajo presentado en [Gutiérrez Brida and Scilingo, 2017]

Figura 38: Dynamic subsumption analysis para el operador *BEE*

6 Evaluación

En la sección 1.1 definimos las motivaciones y objetivos de esta tesis, los cuales constan del diseño y desarrollo de un operador de mutación, altamente configurable, para expresiones de navegación. Luego, en la sección 3.2 definimos que propiedades afectan el análisis de *mutation testing*, cantidad de mutantes, mutantes equivalentes, dificultad de detección, acoplamiento con fallas reales, y redundancia de mutantes (o acoplamiento entre mutantes). En este capitulo realizamos la evaluación de nuestro "meta-operador", *prvo*, para determinar la utilidad práctica de la misma.

6.1 CASOS DE ESTUDIO

Como se mencionó en la introducción 1.2, los casos de estudio sobre los cuales se va a evaluar a *prvo* en el contexto de mutation testing, son estructuras que representan colecciones implementadas en *Java* y con un uso significativo de características de programas orientados a objetos, entre ellas, y la más importante para nuestro estudio, el uso de expresiones de navegación.

TreeList Una implementación de listas sobre árboles AVL, es decir, árboles binarios de búsqueda balanceados (la diferencia de altura entre el sub-árbol izquierdo y el sub-árbol derecho es siempre menor o igual a 1). La implementación proviene de *Apache Collections*.

AvlTree Una implementación de árboles binarios de búsqueda balanceados (la diferencia de altura entre el sub-árbol izquierdo y el sub-árbol derecho es siempre menor o igual a 1).

	TreeList	AvlTree	BinHeap	TreeSet	NCL	BSTree	Queue
LOC	378	71	119	204	201	91	36
Methods	60	18	9	21	48	10	9

Tabla 9: Casos de estudio y cantidad de líneas de código (LOC) y métodos de cada uno.

BinomialHeap Una implementación de *Binomial Heap*, una secuencia creciente de árboles binomiales con respecto al orden de los mismos. Un árbol binomial es o bien un nodo, en cuyo caso se trata de un árbol de grado 0; o bien un nodo donde los hijos representan árboles binomiales de grado *k-1* hasta 0, para un árbol de grado *k*. En un *Binomial Heap* los árboles binomiales cumplen con la propiedad de que cada nodo en un árbol tiene un valor asociado mayor o igual al de su nodo padre. Y en el conjunto de árboles binomiales que conforman al *Binomial Heap*, no puede haber más de un árbol con un determinado orden.

TreeSet Un conjunto implementado sobre árboles rojos y negros. Éstos permiten mantener un árbol balanceado de tal forma que el camino desde la raíz hasta la hoja más lejana tiene a lo sumo dos veces la distancia de la raíz a la hoja más cercana. Las restricciones de este tipo de árboles son:

a) Cada nodo es rojo o negro.

b) El nodo raíz es negro.

c) El camino desde cualquier nodo a cualquiera de sus hojas, tiene la misma cantidad de nodos negros.

La implementación de *TreeSet* fue sacada de *Java Collections*.

NodeCachingList Una lista doblemente encadenada (cada nodo, excepto el primero y el último, tienen una referencia al nodo siguiente y al anterior) que mantiene y utiliza una *cache* de nodos. El uso de esta cache permite disminuir significativamente las operaciones de alocación de nodos y a su vez disminuir el uso de memoria (si bien los nodos eliminados son finalmente eliminados de la memoria, este proceso es realizado por el *Garbage Collector*, al usar una cache disminuye la necesidad de limpiar la memoria). La implementación está basada en *Apache Collections*, pero fue modificada por el hecho de que la implementación original está dividida en diversas clases, a la vez que la utilizada reúne todo el código relacionado a esta implementación, en tres clases (una clase lista, una nodo, y otra iterador).

BinarySearchTree Una implementación de un árbol binario de búsqueda. Se trata de árboles binarios en donde se cumple la propiedad de:

> Para cada nodo $\mathbf{n}$, todos los nodos alcanzables desde el sub-árbol izquierdo ($\mathbf{n}_i$) tienen un valor asociado menor al de $\mathbf{n}$; y todos los nodos alcanzables desde el sub-árbol derecho ($\mathbf{n}_d$) tienen un valor asociado mayor al de $\mathbf{n}$.

Queue Una implementación de una cola, basada en el ejemplo de motivación provisto en 1.1 (1). Ésta, al contrario de la provista como motivación, fue implementada de manera correcta.

Es necesario aclarar la razón para elegir un conjunto de casos de estudio que en principio parece "armado" para *prvo*. De acuerdo a la configuración que vamos a utilizar, las mutaciones se aplicarán a expresiones de navegación, es decir, si no hay expresiones de este tipo no van a haber mutaciones. El hecho de que los casos de estudio involucren estructuras de datos en donde hay una gran cantidad de expresiones de navegación no representa de por sí una ventaja para *prvo*, ya que generar una mayor cantidad de mutantes, impone un mayor peso a cubrir las otras propiedades bajo análisis, es decir, mientras más mutantes generemos vamos a necesitar que éstos tengan un mayor grado de dominancia y menor cantidad de mutantes equivalentes.

6.2 GENERACIÓN DE TESTS

Las test suites necesarias para poder realizar mutation testing fueron generadas por herramientas de generación automática (*EvoSuite* [Fraser and Arcuri, 2011] y *Randoop* [Pacheco and Ernst, 2007]). La razón se basa en que ambas son herramientas de generación automática de tests comúnmente utilizadas en trabajos de investigación; y además nos permiten eliminar cualquier parcialidad introducida si los tests fueran generados manualmente.

Ambas herramientas son no determinísticas, principalmente por utilizar procedimientos aleatorios. El impacto de esto en los experimentos puede ser minimizado al proveer a estas herramientas con un "seed", un número que va a ser utilizado como semilla para los generadores aleatorios que utilizan ambas herramientas. A su vez, los experimentos fueron repetidos 30 veces con distintas semillas para garantizar que una semilla particular no afectaba los resultados. Éstas, fueron generadas una vez, mediante generación aleatoria, y se guardaron en un archivo para poder ser reutilizadas. La forma en que se ejecutan los experimentos garantiza que cada vez que se ejecute el i-ésimo experimento, se va a utilizar la misma semilla.

Tanto *EvoSuite* como *Randoop* permiten definir un "presupuesto" o *budget* de tiempo para la generación de tests. En experimentos previos, al correr las herramientas con 30, 60, y 120 segundos, pudimos comprobar que la diferencia que puede haber entre los dos primeros tiempos, con respecto a mutation testing y cobertura de ramas, es pequeña (beneficiando al segundo caso), mientras que

entre 60 y 120 segundos, la diferencia es despreciable. Es por esto que decidimos utilizar **60 segundos** como presupuesto de tiempo para ambas herramientas.

A su vez, *EvoSuite* permite definir que métricas de evaluación de tests se van a utilizar durante la generación de tests, siendo *EvoSuite* una técnica que se basa en algoritmos genéticos, la forma de guiar la evolución de tests es al intentar maximizar las métricas seleccionadas. Nuevamente, basados en estudios previos (trabajos de investigación del grupo) notamos que utilizar **cobertura de ramas** y **weak mutation**[1] resultan en suites de mejor calidad (mayor capacidad de detectar mutantes y mejor cobertura de ramas).

Para garantizar que la evaluación entre usar o no *prvo* sea lo menos afectada posible por el no determinismo de la generación de tests, éstos se van a generar una única vez para cada caso de estudio (y cada semilla) para luego utilizar exactamente la misma suite para evaluar ambos casos (utilizando solo los operadores suficientes, y agregando *prvo*).

6.3 CONFIGURACIÓN DE PRVO

Antes de evaluar a *prvo*, es necesario que definamos una configuración particular para el mismo. Teniendo en cuenta que la configuración sobre miembros de clases a utilizar o evitar, es decir, una de las propiedades principales que regula el conjunto de expresiones a utilizar durante la mutación, es algo que debe definirse por cada proyecto, vamos entonces para este caso, a definir de manera general que criterio vamos a utilizar para configurar esta propiedad para cada proyecto. El primer problema a resolver para poder definir esta configuración y el criterio que controla las expresiones a utilizar, es especificar que tipo de fallas estamos interesados en representar, puesto que lo mejor no es una configuración que intente representar múltiples fallas sino una que se centre en un tipo de falla específica. De acuerdo a nuestro ejemplo motivador, dado en la sección 1.1, el tipo de fallas que queremos representar son aquellas que involucran expresiones de navegación, es decir, expresiones que involucran al menos un operador de navegación (operador *punto*), y que resultan de el acceso al miembro incorrecto de una clase. Estas fallas no parecen estar de acuerdo con el ejemplo, recordemos que éste utiliza un método *dequeue()* que solo tiene una línea donde reasigna el valor a un puntero, y la falla está en la falta de código, como controles y reasignación de otros punteros, sin embargo, modificar un miembro en las expresiones halladas en la única sentencia del método, genera

[1] Esta técnica difiere del concepto usual de mutation testing (Strong mutation) al considerar un mutante detectado si en la siguiente instrucción luego de la mutación, el estado del programa es distinto al del original

la necesidad de mejorar la test suite. Por parte de las fallas mencionadas como no acopladas actualmente a fallas reales, descriptas en 4.4, dentro de éstas, nos vamos a enfocar en las siguientes:

1) Llamada a un método similar de la misma librería

2) Acceso directo a un campo[2]

Pasemos entonces a definir la configuración a utilizar:

Expresiones objetivo

Las expresiones que vamos a mutar son aquellas que sean expresiones de navegación y se encuentren en cualquier contexto, asignaciones (parte izquierda y derecha), expresiones unarias y binarias, sentencias de retorno, ciclos y llamadas a métodos, etc.

Expresiones a utilizar

Con respecto a las expresiones que se van a utilizar al mutar una expresión, se va a restringir el uso a solo aquellas que pertenezcan a la clase principal bajo evaluación o aquellas relacionadas a la estructura principal que ésta representa, recordemos que nuestros casos de estudio van a estar compuestos por estructuras de datos. Ejemplo de esto es, para la clase *LinkedList* solo se van a poder utilizar miembros de la misma y de la clase *Node* utilizada por ésta. Es necesario aclarar que las variables alcanzables también van a ser consideradas.

El control de tipos va a ser estricto, lo que significa que una subexpresión solo puede intercambiarse por otra con **exactamente** el mismo tipo. Al tener como objetivo a expresiones de navegación, el uso de literales no tiene mucho sentido, esto no significa no utilizar constantes como por ejemplo definir el puntero al comienzo de la lista con *final*, o definir un nodo ficticio de nuevo con el mismo modificador. El uso de campos estáticos solo se va a permitir dentro de un contexto estático, por ejemplo, en el método `public static int foo()`, se van a permitir miembros estáticos, pero en el método `public int bar()`, no.

Si bien parece una configuración muy acotada en cuanto a las mutaciones que produce, y a cuantos tipos de fallas reales intenta representar. Incluso con las restricciones impuestas en la implementación de *prvo*, una configuración poco restrictiva puede dar lugar a una explosión en la cantidad de mutantes

[2] Solo para campos de las clases bajo evaluación

```
1  public boolean find( int x ) {
2    Node current = root;
3    while (current != null) {
4      if (current.key == x) {
5        return true;
6      }
7      if (x < current.key) {
8        current = current.left;
9      } else {
10       current = current.right;
11     }
12   }
13   return false;
14 }
```

Figura 39: Método *find(int)* de *BSTree*

	Sin restricciones	Sin literales (salvo null)	Control de tipos estricto	Solo bstree.*	Solo navegación	Configuración tesis
prvo mutants	141	114	83	93	42	6

Tabla 10: Mutantes generados para el método *find(int)* 39

generados. Una elevada cantidad de mutantes no solo tiene como desventaja las implicaciones sobre los recursos necesarios, la cantidad de mutantes irrelevantes que se generan aumenta. Para ver un ejemplo de esto, al mutar con *prvo* el código del método *find(int)* de *BSTree*, el que se muestra en la Figura 39, usando la configuración más permisiva podemos apreciar en la Figura 40 algunas de las mutaciones generadas, muchas de éstas resultando completamente irrelevantes. En la Tabla-10 se expone como cada configuración afecta a la cantidad de mutantes producidos. Cabe destacar que la cantidad posible de configuraciones para *prvo* es mucho más grande que las expuestas en la tabla, por ejemplo la capacidad de ajustar las restricciones de tamaño descriptas en 5.2.1 no está mencionada.

6.4 OPERADORES A UTILIZAR

Para nuestros experimentos vamos a seleccionar todos los operadores básicos, que aplican a nivel de método, implementados en *μJava++*. Éstos son un

```
Línea 2:
Node current = null;
Node current = root.right;
Node current = root.left;

Línea 3:
while (new java.lang.Double( 1.0d ) != null) {
while (getClass().getConstructors() != null) {
while (current != root) {
while (current.left != null) {
while ("" != null) {
while (current.toString() != null) {

Línea 4:
if (hashCode() == x) {
if (current.key == 1.0f) {
if (current.left.key == x) {

Línea 5:
return false;

Línea 8:
current = root.left;
current = current.left.left;
current = current.left.right;
root.right = current.left;
```

Figura 40: Mutaciones *prvo* (sin restricciones) del método *find(int)* de *BSTree* 39

conjunto que incluye al conjunto de operadores suficientes definidos en 3.1, excepto por *ABS* (el cual cambia expresiones aritméticas a 0, un valor positivo, y un valor negativo) por no estar implementado en *μJava* y por lo tanto no lo está en *μJava++*.

AODS — Elimina los operadores aritméticos de incremento y decremento (pre-decremento: *--exp*; pre-incremento: *++exp*; pos-decremento: *exp--*; y pos-incremento: *exp++*).

AODU — Elimina los operadores aritméticos unarios (*+exp* y *-exp*).

AOIS — Inserta los operadores aritméticos de incremento y decremento en variables y campos de clases.

AOIU — Inserta los operadores aritméticos unarios en expresiones artiméticas.

AORB — Reemplaza operadores aritméticos binarios por otros, en una expresión binaria.

AORS — Reemplaza operadores aritméticos de incremento y decremento.

AORU — Reemplaza operadores aritméticos unarios en expresiones.

ASRS — Reemplaza los operadores de asignación compuestos por otros del mismo tipo, éstos operadores de asignación, permiten reemplazar expresiones del tipo **x = x op y** por **x op= y**, por ejemplo **x += y**.

COD — Elimina el operador unario de negación condicional.

COI — Inserta el operador unario de negación condicional a expresiones de tipo booleano.

COR — Reemplaza operadores condicionales binarios por otros (*&&*, *||*). Si bien no considerados condicionales, los operadores lógicos pueden ser utilizados también por este operador, incluyendo el “y” (*&*), “o” (*|*), y “xor” (ˆ) bit a bit.

LOD — Elimina el operador lógico de negación binaria (∼).

LOI — Inserta el operador lógico de negación binaria.

LOR — Reemplaza operadores lógicos binarios por otros (*&*, *|*, ˆ).

ROR Reemplaza operadores relacionales con otros ($<$, $<=$, $>$, y $>=$), y reemplaza todo un predicado por *verdadero* y *falso*.

Si bien la elección de este conjunto de operadores parece dejar de lado desarrollos más recientes en mutation testing, como por ejemplo, mutantes de alto orden *HOMs* [Jia and Harman, 2009b], una técnica que combina mutantes de primer orden, como los generados por las herramientas actuales de mutation testing (μ*Java*, *PITest*, *Major*, entre otras) para producir mutantes más complejos y (en principio) más difíciles de detectar con menos mutantes equivalentes. La razón se basa en que por un lado las herramientas de mutation testing disponibles (principalmente para *Java*) no suelen implementar esta técnica; por otra parte, no existen operadores de alto orden, sinó mutantes que son generados mediante distintas técnica como combinación aleatoria, algoritmos genéticos, combinación de mutantes correspondientes a distintas sentencias, etc. Esto hace que comparar con *HOMs* requiera comparar con distintas técnicas que a su vez dependen de que operadores se utilizan para generar los mutantes de primer orden. A la vez que comparar mutantes generados solo por una mutación de *prvo* con aquellos donde se combinan varios mutantes pondría en desventaja a *prvo* (se está comparando contra los operadores más la técnica de selección), y si se usa *prvo* en combinación con otros entonces solo se estaría evaluando cuanto aporta nuestro operador sin poder realizar una análisis de subsuma por que los mutantes son mixtos (se componen por varios mutantes generados por los mismos o distintos operadores). Nosotros creemos que dada la "estabilidad" del conjunto de operadores suficientes con respecto a que el mismo se ha mantenido casi sin cambios durante años (al contrario de las técnicas de generación de *HOMs* que aún están bajo constantemente evolución) y el hecho de que la mayoría de las herramientas de mutation testing no solo que no generan *HOMs*, sino que además utilizan operadores que en su mayoría corresponden al conjunto de los operadores suficientes, el utilizar el conjunto anterior representa una comparación justa y relevante para nuestro operador.

6.5 EXPERIMENTOS

La ejecución de los experimentos fue automatizada mediante scripts, para por un lado, evitar errores humanos al seguir los pasos de los mismos, y por el otro para no tener que controlar constantemente el estado de los experimentos. Cada experimento se define por un conjunto de propiedades que definen la clase a evaluar, datos de localización de fuentes y binarios, y configuración de *prvo* y μ*Java++*. A partir de éstas, los pasos a seguir son los siguientes:

Generar tests
: Si los tests para un caso particular, con un seed particular aún no han sido generados, entonces se generaran a los mismos utilizando *EvoSuite* y *Randoop*. Si los tests ya han sido generados previamente, este paso será ignorado (por ejemplo, al comparar los efectos de *prvo* para un caso de estudio y un seed particular los tests solo serán generados una vez y reutilizados la segunda).

Evaluar la calidad de los tests
: Para tener una medida sobre la calidad de las test suites generadas, vamos a utilizar cobertura de ramas mediante la herramienta *JaCoCo*. Ésta genera un archivo *xml* con los datos de cobertura (así como archivos *html* para visualizar los datos), aunque nosotros vamos a utilizar una herramienta propia para recopilar un resumen de los datos generados para su rápida visualización.

Ejecutar *µJava++*
: Como ya hemos mencionado, *prvo* está implementado en esta herramienta, por eso la evaluación va a ser realizada con la misma. Ésta funciona tomando un archivo de propiedades como entrada y ejecutando el o los análisis especificados por las mismas. En la Figuras 41 y 42 se muestra un ejemplo de un archivo de propiedades y más adelante se da una explicación de las mismas. En la primera se ven las configuraciones generales del análisis y en la segunda la configuración de los operadores.

Recopilación de información
: Si bien *µJava++* genera una salida con la información pertinente a su ejecución, es conveniente poder recopilar en forma resumida la información más pertinente para rápido acceso.

Si bien hay muchas más propiedades en la configuración que se muestra de las que se puedan (o que tenga sentido) explicar, vamos a mencionar las que afectan significativamente a nuestros experimentos.

Con respecto a las opciones generales, **GENERAL OPTIONS** 41, *µJava++* se va a ejecutar sin tener en cuenta las anotaciones *//mutGenLimit K* que limitan donde y cuantas mutaciones consecutivas se pueden aplicar, esto se debe a que: (**i**), queremos generar mutantes para todas las sentencias posibles; y (**ii**), solo vamos a generar una sola generación de mutantes (no más de una mutación simultanea), definido por la propiedad *mutation.advanced.generations*. Para cada experimento se va a mutar una única clase, definida por la propiedad *mutation.basic.class*, y todos los métodos de la misma, definido por la propiedad *mutation.basic.methods* que al estar en blanco indica que todos los métodos serán

```
//===============GENERAL OPTIONS===============

mutation.advanced.fullVerbose= true
mutation.advanced.ignoreMutGenLimit= true
mutation.advanced.generations= 1

//path to original source folder (e.g.: src/)
path.original.source= benchmarks/src/

//path to original bin folder (e.g.: bin/)
path.original.bin= benchmarks/bin/

//path to tests bin folder
path.tests.bin= tests/treelist/

//where to save mutants
path.mutants= mutants/benchmarks/treelist/

//class to mutate (full qualified name)
mutation.basic.class= treelist.TreeList

//methods to mutate (separated by spaces)
mutation.basic.methods=

//operators to use (separated by spaces)
mutation.basic.operators= AODS AODU AOIS AOIU
AORB AORS AORU ASRS COD COI COR LOD LOI LOR
ROR SOR PRVOU_REFINED PRVOR_REFINED PRVOL_SMART

//test classes to run (names separated by spaces)
mutation.basic.tests= treelist.TreeListESTest
RegressionTest

//===============ADVANCED OPTIONS===============

mutation.advanced.timeout= 1000
mutation.basic.mutationScore= true
mutation.advanced.toughness= true
mutation.advanced.dynamicSubsumption= true
mutation.advanced.dynamicSubsumption.reduceGraph= true
mutation.advanced.dynamicSubsumption.output= subsumption/
mutation.basic.showSurvivingMutants= true
mutation.advanced.quickDeath= false
mutation.advanced.outputMutationsInfoInMutationScore= true
mutation.advanced.useExternalJUnitRunner= true
mutation.advanced.useParallelJUnitRunner= true
mutation.advanced.parallelJUnitRunnerThreads= 10
mutation.basic.useSockets= true
mutation.basic.writePrologue= true
```

Figura 41: Configuración de ejemplo para μ*Java++* [parte 1]

```
//===============OPERATORS OPTIONS===============

mutation.advanced.allowFieldMutations= false
mutation.advanced.allowClassMutations= false
mutation.advanced.allowedMembers= treelist(\\.([^#]+))?\\#.*

//AOIS
mutation.advanced.aois.skipFinal= true

//AODS
mutation.advanced.aods.skipExpressionStatements= true

//PRVO
mutation.advanced.prvo.enableSameLenght= true
mutation.advanced.prvo.enableIncreaseLenght= false
mutation.advanced.prvo.enableDecreaseLenght= false
mutation.advanced.prvo.enableOneByTwo= false
mutation.advanced.prvo.enableTwoByOne= false
mutation.advanced.prvo.enableAllByOneLeft= false
mutation.advanced.prvo.enableAllByOneRight= false
mutation.advanced.prvo.enableLiteralMutations= false
mutation.advanced.prvo.enableSuper= false
mutation.advanced.prvo.enableThis= false
mutation.advanced.prvo.enableReplacementWithLiterals= false
mutation.advanced.prvo.enableObjectAllocationMutations= false
mutation.advanced.prvo.enableArrayAllocationMutations= false
mutation.advanced.prvo.enableNonNavigationExpressionMutations= false
mutation.advanced.prvo.enableNullLiteral= false
mutation.advanced.prvo.enableTrueLiteral= false
mutation.advanced.prvo.enableFalseLiteral= false
mutation.advanced.prvo.enableEmptyString= false
mutation.advanced.prvo.enableZeroLiteral= false
mutation.advanced.prvo.enableOneLiteral= false
mutation.advanced.prvo.enableStringLiterals= false
mutation.advanced.prvo.allowFinalMembers= true
mutation.advanced.prvo.enableRelaxedTypes= false
mutation.advanced.prvo.enableAutoboxing= false
mutation.advanced.prvo.enableInheritedElements= false
mutation.advanced.prvo.enableStaticFromNonStaticExp= true
mutation.advanced.prvo.applyRefinedPRVOInMethodCallStatements= true
mutation.advanced.prvo.allowNumericLiteralVariations= false
mutation.advanced.prvo.disablePrimitiveToObjectAssignments= true
mutation.advanced.prvo.wrapPrimitiveToObjectAssignments= false
mutation.advanced.prvo.smartmode.arithmeticOpShortcuts= true
mutation.advanced.prvo.smartmode.assignments= true

//ROR
mutation.advanced.ror.replaceWithTrue= true
mutation.advanced.ror.replaceWithFalse= true
mutation.advanced.ror.smartLiteralReplacement= true

//COR
mutation.advanced.cor.andOperator= true
mutation.advanced.cor.orOperator= true
mutation.advanced.cor.xorOperator= true
mutation.advanced.cor.bitAndOperator= false
mutation.advanced.cor.bitOrOperator= false
```

Figura 42: Configuración de ejemplo para *μJava++* [parte 2]

mutados. Los operadores a utilizar, previamente nombrados y descriptos en 6.4, están definidos por la propiedad *mutation.basic.operators*. Finalmente los tests a utilizar son los generados por *EvoSuite* y *Randoop*, definidos respectivamente en la propiedad *mutation.basic.tests*.

Algunas propiedades avanzadas que se van a utilizar en los experimentos, definidas en la sección **ADVANCED OPTIONS**, son: los tests van a ser ejecutados con un timeout de entre 300ms a 1 segundo según el caso de estudio (en las propiedades mostradas el segundo valor es el utilizado), esto se debe a que teniendo en cuenta los casos de estudio utilizado, esta cantidad de tiempo es suficiente para detectar mutantes que causen ciclos infinitos sin obtener una gran cantidad de falsos positivos (mutantes que solo causan que el algoritmo tarde un poco más pero que no produzca ciclos infinitos), el timeout está definido por la propiedad *mutation.advanced.timeout*, aunque solo afecta a aquellos tests que no tengan un timeout definido en el test, en la Figura-46 el test **test1** tiene un timeout de 2 segundos, mientras que **test2** al no tener un timeout definido sería ejecutado con el que defina la opción previa; se van a realizar los análisis de mutación (calcular mutation score), dureza (cuantos tests es capaz de sobrevivir un mutante), dynamic mutant subsumption, definido por las propiedades *mutation.basic.mutationScore*, *mutation.advanced.toughness*, y *mutation.advanced.dynamicSubsumption* respectivamente; finalmente, *μJava++* va a ejecutar el análisis de cada mutante en un proceso aparte y de hasta 10 mutantes a la vez en paralelo, definido por las propiedades *mutation.advanced.useExternalJUnit*, para el análisis de mutantes mediante un proceso externo, y las propiedades *mutation.advanced.useParallelJUnitRunner* y *mutation.advanced.parallelJUnitRunnerThreads* para definir la ejecución en paralelo de hasta 10 mutantes.

Con respecto a las propiedades que afectan a los operadores, definidas en la sección **OPERATORS OPTIONS** 42, *mutation.advanced.allowFieldMutations* permite activar o desactivar la mutación de campos de clase, esto afecta tanto a operadores que aplican exclusivamente a declaraciones de miembros de una clase (campos, métodos y subclases) haciendo que para algunos de éstos, utilizar esta propiedad logra deshabilitarlos por completo; como a operadores como *prvo* que podrían aplicarse a las expresiones utilizadas en la inicialización de campos de clase. Para el caso de la propiedad *mutation.advanced.allowClassMutations*, solo sirve como forma de deshabilitar a todos los operadores que afectan a la declaración de una clase (como eliminar métodos sobreescritos), cuando esta propiedad está en *true* las mutaciones de clases siguen dependiendo de que los operadores correspondientes hayan sido agregados a la configuración. A continuación se explican las propiedades que afectan a distintos operadores, vale destacar que en muchos casos no resulta razonable la existencia de ciertas

```
1  public class Foo {
2    private int field1 = 0;
3    private final int field2 = 0;
4
5    public void bar(int var1, final int var2) {
6        int l1 = var1;
7        int l2 = var2;
8        int l3 = field1;
9        int l4 = field2;
10   }
11 }
```

Figura 43: Ejemplo de un método donde la mitad de los mutantes generados por *AOIS* fallarían la compilación

propiedades, principalmente cuando se trata de aquellas que permiten activar o desactivar una mejora, la razón de éstas es para poder mantener una versión "legacy" de cada operador.

Para el operador *AOIS*, el cual inserta operadores aritméticos de incremento y decremento (tanto en sus versiones pre y post) en expresiones aritméticas, la propiedad *mutation.advanced.aois.skipFinal* permite configurar el operador para que ignore las expresiones constantes. Por ejemplo, en la Figura-43, cualquier mutante generado por este operador en las líneas 7 y 9 (un 50 % de todos los mutantes) no compilaría por que no es posible incrementar una variable declarada con *final* (básicamente una constante).

Para el operador *AODS*, el cual elimina operadores aritméticos de incremento y decremento (tanto en sus versiones pre y post) de expresiones aritméticas, la propiedad *mutation.advanced.aods.skipExpressionStatements* permite configurar el operador para que ignore sentencias que se conforman únicamente de una expresión aritmética con un operador aritmético de incremento o decremento. Por ejemplo en la Figura-44, la mutación generada en la línea 5, si bien no es difícil de detectar (causa un ciclo infinito), genera un mutante que compila, mientras que mutar la línea 6 eliminando el operador de incremento causa que el mutante no compile (una variable sola no representa una sentencia válida).

Para el operador *prvo*, existen numerosas propiedades que pueden configurarse[3], las primeras 5 *enableSameLenght*, *enableIncreaseLenght*, *enableDecrease-*

[3] Dada la cantidad y que todas comparten el mismo prefijo (*mutation.advanced.prvo*), se utilizará la última parte de éstas para hacer referencia a las mismas

```
1   public int foo(int[] a) {
2     int result = 0;
3     int idx = 0;
4     while (idx < a.length) {
5       if (a[idx++] % 2 == 0) {
6         result++;
7       }
8     }
9   }
```

Figura 44: Ejemplo de un método donde la mitad de los mutantes generados por *AODS* fallarían la compilación

Lenght[4], *enableOneByTwo*, y *enableTwoByOne* corresponden a las restricciones sobre que modificaciones de tamaño se aplican durante la mutación, correspondiendo respectivamente a mantener el tamaño de la expresión, incrementar la expresión en 1, decrementar en 1, reemplazar una subexpresión de tamaño 0 con una de tamaño 1, y reemplazar una subexpresión de tamaño 1 con una de tamaño 0. Las dos propiedades *enableAllByOneLeft* y *enableAllByOneRight* permiten habilitar o deshabilitar la generación de mutaciones en sentencias de asignación, donde una expresión es mutada a una de tamaño 0 independientemente de su tamaño original, respectivamente estas propiedades afectan la parte izquierda y derecha de una asignación[5]. La propiedad *enableLiteralMutations* permite habilitar o deshabilitar la mutación de literales, es decir, considerar las expresiones constituídas por un valor literal como objetivo de *prvo*. Tanto *enableThis* como *enableSuper* permiten considerar (o no) a *this* y *super* como parte de la expresión, cuando éstas se encuentran desactivadas, el uso de las expresiones especiales *this* y *super* solo se van a utilizar cuando sea necesario:

1. Existan dos expresiones con el mismo nombre donde una define a un campo de clase (en cuyo caso se utiliza *this*).

2. Existan dos expresiones con el mismo nombre donde una define un miembro heredado (en cuyo caso se utiliza *super*).

La propiedad *enableReplacementWithLiterals* en conjunto con las propiedades *enableNullLiteral*, *enableTrueLiteral*, *enableFalseLiteral*, *enableEmptyString*, *enableZeroLiteral*, *enableOneLiteral*, y *enableStringLiteral*, permite reemplazar

[4] Vale aclarar que éstas propiedades comparten todas el mismo typo
[5] Estas dos propiedades son principalmente reservadas para contextos de reparación

expresiones de tamaño 0 con literales, donde cada propiedad habilita o deshabilita el uso de ciertos literales por defecto. Cuando se utiliza el reemplazo por literales, *prvo* realiza una búsqueda de literales alcanzables desde el punto donde se va a mutar, la propiedad *enableStringLiteral* permite incluir literales de tipo *String* en la búsqueda. En conjunto con estas propiedades, *allowNumericLiteralVariations* permite la generación de variantes de literales numéricos (para cada literal numérico perteneciente a un tipo particular, se generan las variantes del mismo valor para los otros tipos disponibles).

Las propiedades *enableObjectAllocationMutations*, *enableArrayAllocationMutations*, *enableNonNavigationExpressionMutations*, y *applyRefinedPRVOInMethodCallStatements*, permiten aplicar restricciones sobre los objetivos donde va a ser aplicado *prvo*. Las primeras dos restringen respectivamente, si *prvo* va ser aplicado a expresiones de creación de objetos (expresiones que utilizan el operador *new*), y a expresiones de creación de arreglos, en cuyo caso la expresión de alocación se asume como una de tamaño 0. La aplicación de *prvo* a expresiones de tamaño 0 puede ser restringida con la tercer propiedad, cuando ésta se desactiva solo se podrán mutar expresiones de navegación (expresiones de tamaño 1 o más). Finalmente, la última propiedad permite aplicar *prvo* dentro de los argumentos de llamadas a métodos cuando éstas llamadas conforman una sentencia, por ejemplo **foo(a, current.next);**.

Es posible evitar el uso de campos y variables constantes mediante la propiedad *allowFinalMembers*. El uso de métodos y campos heredados con *enableInheritedElements*, en cuyo caso permite un control sobre la cantidad de expresiones utilizables. Como se mencionó anteriormente, utilizar expresiones estáticas en un contexto no estático puede ser muy interesante para detectar casos donde faltó definir a la expresión como constante o se utilizó el modificador *static* de manera incorrecta, los tests que permiten detectar uso indebido de expresiones estáticas requieren en general múltiples instancias de la misma clase.

Las propedades *enableAutoboxing*, *disablePrimitiveToObjectAssignments* y *wrapPrimitiveToObjectAssignments* permiten lidiar con el uso de tipos primitivos y no primitivos, específicamente con *Object*. Un tipo primitivo no es un objeto, nunca puede ser nulo, y usado en conjunto con el modificador *final* da lugar a una constante propiamente dicha. En muchos casos, un objeto de tipo *Object* es compatible con cualquier otro tipo, incluído los primitivos, lo que inicialmente lleva a permitir una gran cantidad de mutantes. Por el otro lado, existen casos donde es necesario "envolver" un valor de un tipo primitivo en su contrapartida no primitiva, por ejemplo *new Integer(1)* "envuelve" el valor primitivo 1 en un tipo no primitivo *Integer*. A su vez, *Java* tiene un mecanismo llamado *autoboxing* en el cual una expresión como *Integer i = 1;*, es considerada válida al considerar al 1 como un *Integer*. La primer propiedad permite habilitar o no

```
1   public void foo(Integer a, Double b) {
2     Integer l1 = a;
3     Integer l2 = b;
4     Double l3 = b;
5     Number l4 = a;
6     Number l5 = b;
7   }
```

Figura 45: Ejemplo de control relajado y estricto de tipos

```
@Test(timeout = 2000)
public void test1() {
        NCLL list = new NCLL();
        list.add(1);
        assertTrue(list.contains(1));
}

public void test2() {
        NCLL list = new NCLL();
        list.add(1);
        list.remove(1);
        assertFalse(list.contains(1));
}
```

Figura 46: Ejemplo de tests afectados por la propiedad de timeout

este tipo de control de tipo, relajando o restringiendo al mismo, en caso de estar deshabilitado, *prvo* no podría generar una expresión como la anterior. Las dos otras propiedades permiten controlar si se permiten o no generar asignaciones de valores primitivos a variables de tipo *Object*, y respectivamente si en caso de permitirlo, éstas se deben "envolver" en el tipo no primitivo correspondiente.

El control de tipos que utiliza *prvo*, es decir, el mecanismo por el cual verifica que intercambiar una expresión *e* por otra *f* es un intercambio válido con respecto a tipos, está controlada por la propiedad *enableRelaxedTypes*. Dado dos clases *A* y *B*, usar un control estricto lleva a que éstas son compatibles si son exactamente el mismo tipo. En la Figura 45, la asignación en la línea 3 resulta incompatible sin importar si el control es estricto o relajado, mientras que las asignaciones en las líneas 5 y 6 solo son válidas si se utiliza un control de tipos relajado. Esta propiedad permite restringir la cantidad de mutantes generados, incluso puede considerarse una de las propiedades que más afecta a la cantidad de mutaciones generadas.

Las últimas dos propiedades de *prvo*, *smartmode.arithmeticOpShortcuts* y *smartmode.assignments* representan mejoras para evitar mutantes que no compilan. En el primer caso, la propiedad permite evitar el uso de expresiones constantes (declaradas como *final*) para reemplazar el último elemento en una expresión a la cual se le está aplicando un operador aritmético de incremento o decremento. La segunda propiedad permite evitar el uso de expresiones constantes en la parte izquierda de una asignación. Como se explicó anteriormente, si bien estas propiedades parecen innecesarias ya que mejoran al operador para evitar la generación de mutantes que no compilan, se mantienen como opciones para mantener un comportamiento "legacy".

El operador *ROR* reemplaza operadores relacionales por otros disponibles, pero también reemplaza toda una expresión booleana por los valores *true* y *false*. Las propiedades *mutation.advanced.ror.replaceWithTrue* y *mutation.advanced.ror.replaceWithFalse* permiten habilitar o deshabilitar el uso de estos valores para reemplazar una expresión. Existen casos donde reemplazar una expresión por *true* o *false* generan código inalcanzable lo que genera un error al compilar dicho código. La tercer propiedad disponible para este operador, *mutation.advanced.ror.smartLiteralReplacement* permite a *ROR* realizar un análisis para determinar que constantes puede utilizar al reemplazar una expresión[6]:

While
: Una sentencia *while* ejecuta el cuerpo de la misma mientras la condición asociada sea verdadera. El valor *false* es directamente desechado por que causaría código muerto de manera inmediata. Sin embargo, el uso de *true* podría o no causarlo, si no existe ninguna sentencia *break* entonces el código que sigue luego del *while* queda inalcanzable, en cambio si existe una de estas sentencias el compilador no arrojara el error.

For
: Una sentencia *for* es similar al *while* salvo por contar con una inicialización previa, una condición asociada a la ejecución o no del *for*, y un incremento (sentencias que modifican al menos una variable utilizada para iterar). El análisis de *ROR* es similar al caso anterior.

DoWhile
: El caso de un *dowhile* es opuesto a los anteriores, en el sentido de que el código asociado se ejecuta al menos una vez, con la reejecución similar a los casos anteriores (se realiza siempre que la condición asociada sea verdadera).

[6] Las sentencias *if then [else]* no resultan en error de compilación incluso si generan código no alcanzable

> En este caso, *ROR* siempre permite el uso de *false*, pero solo permite el uso de *true* si existe una sentencia *break* en el cuerpo del *dowhile*.

Finalmente, el operador *COR* el cual reemplaza operadores condicionales en expresiones booleanas, tiene asociadas las propiedades que permiten habilitar o deshabilitar cada operador disponible para ser utilizado por el mismo. En nuestra evaluación, las propiedades *mutation.advanced.cor.bitAndOperator* y *mutation.advanced.ror.bitOrOperator* que controlan el uso de los operadores *&* y / que representan los operadores de conjunción y disyunción binaria respectivamente. Si las expresiones a mutar son booleanas, usar éstos resulta en mutantes duplicados (distintos sintácticamente pero equivalentes semánticamente), por eso se los deshabilita.

6.6 EVALUACIÓN

El objetivo de la evaluación es analizar si *prvo*, en su configuración anterior, enfocada en expresiones de navegación, es de hecho una contribución a *mutation testing*. La contribución teórica de nuestro "meta-operador" queda clara en nuestras motivaciones y el diseño de *prvo*:

> Existen fallas reales relacionadas a expresiones de navegación que no son representadas por mutantes generados por operadores de mutación existentes.

Sin embargo, la contribución en la práctica puede no existir, *las fallas que representa prvo están subsumidas por aquellas que representan operadores existentes*; *las mutaciones generadas por prvo son triviales de detectar, o son equivalentes*; *la cantidad de mutantes generados por prvo excede los potenciales beneficios de su uso*. Es por esto que nuestra evaluación se centra en las siguientes preguntas:

- **RQ1** El uso de *prvo* complementa de manera significativa a los operadores de mutación tradicionales?
- **RQ2** Cual es el costo adicional de utilizar *prvo*, en mutation testing?

La motivación para **RQ1** está basada en que un nuevo operadores no sería útil si fuera redundante con respecto a los existentes. Para responder **RQ1** vamos a utilizar *dynamic mutant subsumption*. Este análisis permite, para un programa particular y un conjunto de tests particulares, evaluar que mutantes resultan indispensables y cuales son redundantes para la evaluación del conjunto de tests. Vamos a utilizar principalmente como benchmark, implementaciones

de estructuras de datos orientadas a objetos, en *Java*. La razón para considerar un benchmark así, es que este tipo de implementaciones hace un uso substancial de expresiones de navegación, incluyendo tipos recursivos de datos. Nuestro benchmark va a estar compuesto entonces mayormente por implementaciones de collecciones: *Apache's TreeList*, una implementación de listas utilizando árboles; *Apache's NodeCachingList*, una lista encadenada con una caché de nodos; *AvlTree*, una implementación clásica de árboles balanceados; *BinomialHeap*, una implementación basada en referencias de montículos; y *TreeSet*, una implementación de conjuntos utilizando árboles red-black.

Para responder a **RQ1** vamos a utilizar mutation score, toughness (cuantos tests un mutante sobrevive antes de ser detectado), y dynamic mutant subsumption para detectar mutantes dominadores. Además se van a analizar los mutantes no detectados para detectar aquellos que son equivalentes, esto solo se va a realizar para aquellos generados por *prvo*. A su vez, si bien en principio evaluar cuantos nodos dominadores involucran a *prvo* parece una buena médida, es importante destacar que un nodo puede contener varios mutantes equivalentes entre sí, entonces utilizar directamente la relación `(nodos dominadores que involucran a prvo)/total de nodos dominadores`, no sirve de mucho por que la información obtenida no va a necesariamente reflejar cuanto domina *prvo* sobre todos los mutantes. Por esto, vamos a utilizar el concepto de nodos puros descripto previamente en 5.3.1 para poder medir la relación de nodos puros dominadores de *prvo*.

Vamos a necesitar proveer conjuntos de tests para realizar nuestra evaluación. Sobre todo, para que nuestro análisis sea significativo, necesitaríamos utilizar conjuntos de tests relativamente “buenos”, dado que si fuéramos a utilizar test suites pobres (con baja cobertura de código por ejemplo), podríamos no obtener relaciones entre el uso o no de *prvo*, o resultados que solo se mantienen al usar tests de mala calidad.

Dynamic mutant subsumption es un análisis que se ve beneficiado por tener gran cantidad de tests. Entonces es necesario establecer como vamos a generar nuestros tests. Manualmente, tiene la ventaja de hacer tests teniendo un conocimiento de no solo la semántica del programa bajo prueba, sinó un también de la sintáxis. Las desventajas vienen por dos lados, por uno, generar manualmente una gran cantidad de tests, representa un trabajo complejo y tedioso; por el otro lado, no es posible garantizar imparcialidad, incluso de manera subconsciente uno podría caer en la tentación de escribir tests sabiendo el impacto que los mismos van a tener en la evaluación de *prvo*. Automáticamente, tiene la ventaja de que permite generar una gran cantidad de tests, tan grande como uno quiera para algunas herramientas, y brinda imparcialidad. Al mismo tiempo, utilizar herramientas automáticas permite generar varias test suites

distintas con la misma “calidad”, lo cual nos sirve para poder repetir los experimentos y eliminar, al menos en cierto grado, la incertidumbre de si nuestros resultados son buenos por que usamos un conjunto de tests particularmente favorable para nosotros.

Dentro de las herramientas de generación automática de tests, creemos que *Evosuite* y *Randoop* son dos de las mejores en la actualidad, al menos para *Java*. Evosuite utiliza algorítmos genéticos para, a partir de un conjunto inicial de test suites, hacer evolucionar a estas hasta lograr la “mejor”. Si bien Evosuite suele generar buenos tests, el inconveniente para nuestro estudio es que éstos son relativamente pocos, incluso si se incrementan los recursos disponibles para la herramienta, lo único que se mejora son la calidad de los tests generados (con respecto a uno o varios criterios de evaluación), pero como dijimos, necesitamos conjuntos grandes de tests. Es por esto que creemos que complementar los tests de Evosuite con los generados por Randoop, una herramienta que genera tests mediante aleatoriedad en donde la cantidad de tests generados se puede controlar con los recursos utilizados, es la mejor forma para generar las test suites que vamos a utilizar para evaluar *prvo*.

Nuestro análisis tienen que tener en cuenta a *prvo* como un operador adicional[7]. Los otros operadores de mutación, con los que vamos a comparar, son aquellos considerados en el conjunto de operadores de mutación suficientes [Offutt et al., 1996; Namin et al., 2008] e implementados en *muJava*. Detalles de todos los operadores a nivel de método soportados por *muJava* y subsecuentemente por *muJava++* pueden encontrarse en [Ma and and, 2018].

Si bien cuando mencionamos las propiedades a tener en cuenta en el diseño de operadores de mutación, vimos que la mayoría de las mismas tienen un gran impacto en el significado del mutation score obtenido, éste sigue siendo un valor comúnmente utilizado en estudios, y representa **el** indicador asociado a *mutation testing*. Por lo tanto nos interesa observar como éste es afectado por la inclusión de *prvo*. Teniendo en cuenta que, una suba en el mismo representaría el hecho de que *prvo* generó mutantes que en su mayoría fueron detectados; mientras que una baja, representaría que *prvo* generó mutantes que representan nuevas fallas no detectadas por el test suite. Es necesario aclarar, que en cualquier caso, no es posible sacar conclusiones solamente basados en estos cambios, por eso analizamos equivalencia y dominancia.

Tal como describimos en la sección 5.3.1, nuestro anális va a utilizar información obtenida de los grafos de *dynamic mutant subsumption*, particularmente como cambia la representación de los operadores en los nodos dominantes, con y sin *prvo*. La pureza de los nodos nos va a ser de gran utilidad en este anális para

[7] La familia de operadores de *prvo* la consideremos como un solo operador.

poder eliminar los casos en donde mutantes generados por distintos operadores tienen gran presencia en los nodos dominantes, pero son equivalentes entre sí.

Para poder evitar casos donde el test suite utilizado es particularmente beneficioso o desfavorable para *prvo*, vamos a realizar múltiples experimentos para el mismo programa con distintas test suites generadas de manera automática. Dado que las herramientas utilizadas tienen ambas distintos grados de aleatoriedad, vamos a utilizar un conjunto fijo de “semillas” para garantizar cierto grado de replicación. Incluso con la misma semilla, al ser herramientas que se ejecutan con un presupuesto de tiempo determinado, la cantidad de operaciones que logran realizar bajo éste, no siempre es la misma, generando no determinismo. Creemos que la repetición de nuestros experimentos es suficiente para contrarrestar los efectos del mismo.

Respecto a **RQ2**, el foco está en la eficiencia; incluso si *prvo* (o cualquier operador) resulta ser útil en cuanto a los mutantes que genera, su uso se vuelve prohibitivo si los costos asociados a su utilización son muy altos. Este costo está típicamente asociado a dos factores: la complejidad en la generación de los mutantes, y el número de mutantes generados. El primero es comúnmente ignorado, dado que salvo casos muy raros, los recursos utilizados son usualmente despreciables. Si bien *prvo* tiene una complejidad mayor a la de los operadores comúnmente utilizados, principalmente por los controles de alcanzabilidad y compatibilidad de tipos, ésta no es significativa con respecto a los recursos extra que son necesario para analizar una mayor cantidad de mutantes. El segundo factor es en general el que representa la preocupación principal para el diseño de un operador de mutación, dado que en el peor caso, todos los tests deben ser ejecutados para todos los mutantes al computar el *mutation score*, esto es por que en general un mutante al ser detectado por un test ya no requiere seguir ejecutando el resto, solo cuando se utilizan análisis más complejos (como dynamic mutant subsumption) es necesario ejecutar todos los tests por cada mutante. Es por eso que cuantos mutantes genera *prvo* es la métrica principal para analizar su costo. Si bien en el caso de *prvo* la relación entre la configuración utilizada y la cantidad de mutantes generados está directamente relacionada, lo que se puede apreciar en la Tabla-10, la cantidad de mutantes va depender de la configuración y el código bajo análisis, no solo eso, sinó que estamos interesados en la cantidad de mutantes generados y cualquier análisis posible para determinar a la misma va a tener la misma o mayor complejidad que realizar el análisis de mutation testing y revisar la cantidad de mutantes generados.

6.7 RESULTADOS EXPERIMENTALES

En esta sección vamos a discutir los resultados de nuestra evaluación, enfocándonos primero en **RQ1**. La figura 48 resume los resultados de nuestro análisis de subsuma. Los resultados se muestran por caso de estudio. Cada gráfico incluye el porcentaje de mutantes dominantes por operador de mutación, usando la media de todas las repeticiones para el mismo caso de estudio. Éstas representan los experimentos, que como anteriormente explicado, incluyen una ejecución usando *prvo* junto a los operadores descriptos en 6.4, y otra donde solo éstos fueron utilizados.

Los resultados para los experimentos que no utilizan *prvo* se muestran con barras verdes. Por ejemplo, para *TreeList*, los mutantes generados por el operador *LOI* (el cual inserta el operador de complemento binario) representan un poco más del 35 % de los mutantes dominadores, cuando *prvo* no es utilizado. Al tener en cuenta *prvo*, y ejecutar nuevamente los experimentos, estos porcentajes cambian. Los resultados correspondientes a los experimentos donde el operador *prvo* es utilizado, se muestran en barras rojas. Por ejemplo, para el mismo caso anterior (TreeList), el porcentaje de mutantes dominadores *LOI* disminuyen a menos del 30 %. En este mismo caso, los mutantes dominadores de *prvo* representan un 25 % de todos los dominadores.

Es necesario resaltar en los resultados mostrados en las tablas de la Figura 48, el valor correspondiente a *dominadores puros*. Como se describió en 5.3.1, al construir el grafo de subsuma de mutantes, aquellos mutantes que son equivalentes entre sí, es decir, se subsuman el uno al otro, se encapsulan en el mismo nodo del grafo. Esto, lleva a que si un nodo es considerado dominador (no es subsumido por ningún otro nodo, o, lo que es lo mismo a que no existen arcos que partan de otro nodo hacia éste) todos los mutantes dentro del nodo son, a su vez, considerados dominadores. La existencia de nodos dominadores constituídos por mutantes que fueron generados por otros operadores, afecta de manera negativa a nuestras conclusiones, podríamos decir que *prvo* domina cuando en realidad existen mutantes de otros operadores que son equivalentes, con respecto a *dynamic mutant subsumption*. Es por esto que introdujimos el concepto de *dominadores puros*, para poder evaluar cuando mutantes de un operador dominan sin ser equivalentes a otros. Esta información se muestra con barras azules. Por ejemplo, para *TreeList*, más del 20 % de los mutantes dominadores de *prvo* son considerados puros.

Los grafos asociados a *Dynamic Mutant Subsumption* no se incluyen por que son demasiado grandes. En la Figura 47 se muestra un grafo reducido del generado para *TreeList* con *prvo* incluído. El grafo original tiene 178 nodos con

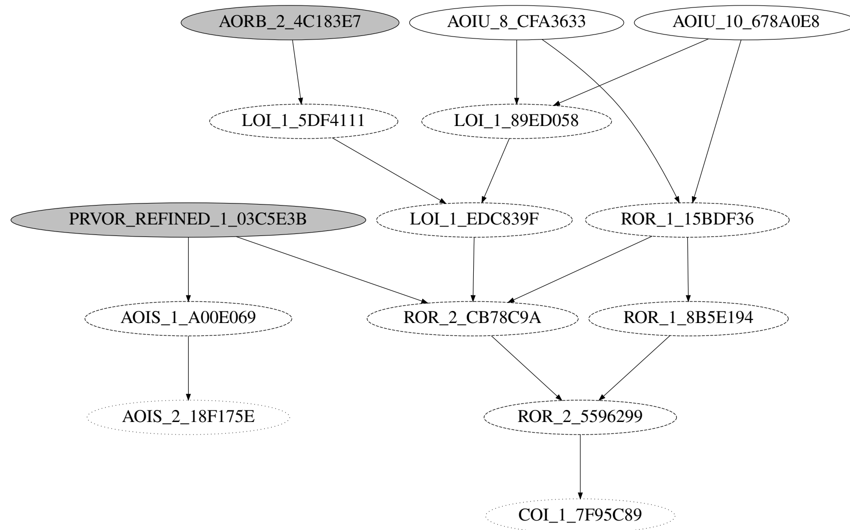


Figura 47: Grafo reducido de *Dynamic Mutant Subsumption* para *TreeList* con *prvo*.

3571 aristas, mientras que el grafo reducido solo tiene 14 nodos y 25 aristas. En el grafo se pueden apreciar nodos dominadores puros (en elipses con fondo gris); nodos dominadores (en elipses con fondo blanco); nodos que subsuman a otros y son a su vez subsumidos, éstos pueden considerarse como "medianamente redundantes" (en elipses con fondo blanco y borde discontinuado); finalmente los nodos que solo son subsumidos por otros y pueden considerarse como definitivamente redundantes (en elipses con fondo blanco y borde punteado). El nombre de los nodos contiene el nombre del operador que generó el primer mutante con el que se formó el mismo, seguido de la cantidad de mutantes equivalentes (con respecto a subsuma dinámica), y finalmente un hash obtenido del primer mutante.

La observación más importante al analizar los resultados es que, en casi todos los casos (con la excepción de *NodeCachingList*), *prvo* se posiciona entre los mejores, con respecto a dominancia, incluso es uno de los mejores tres operadores dominantes, junto a operadores típicos del conjunto de *suficientes*, como *ROR*, *LOI*, y la familia de operadores de mutación que corresponden a la inserción

de operadores aritméticos (*AOIS*, *AOIU*, y *AOIB*). Es también importante de remarcar que, en presencia de *prvo*, la dominancia de otros operadores es en general reducida, y su dominancia sobre ciertos nodos es "transferida" hacia *prvo*. Más precisamente, *prvo* domina sobre ciertos mutantes previamente (en ausencia de *prvo*) dominantes.

Claramente, nuestros resultados experimentales resultan en una respuesta positiva a **RQ1**: *prvo* complementa de manera significativa al conjunto de operadores tradicionales; los mutantes producidos son en general no dominados por operadores existentes, y exhibe un alto número de mutantes dominantes, lo que nos permitiría considerar, al menos en el contexto de nuestros experimentos, a *prvo* como un operador "suficiente", junto al conjunto de operadores suficientes tradicionales.

Analicemos ahora **RQ2**. La Figura 49 resume los resultados para generación de mutantes, comparando el número de mutantes obtenidos cuando *prvo* no es incluído (barras verdes), con los casos en donde *prvo* es considerado (barras azules). Reportamos también, en estas tablas, el número de mutantes equivalentes de *prvo* (mutantes que son semánticamente equivalentes al programa original, y por lo tanto imposibles de detectar con cualquier test) producidos para cada caso de estudio.

Para la mayoría de los casos de estudio, el número de mutantes que genera *prvo* representa solo un pequeño incremento sobre la cantidad de mutantes generados por los operadores tradicionales. Los casos de estudio *TreeSet* y *BinomialHeap* son dos excepciones, en donde la cantidad de mutantes generados por *prvo* representa prácticamente la misma que la generada por todos los operadores tradicionales por si solos. La razón es que estos casos de estudios se caracterizan por una gran cantidad de métodos conteniendo, a su vez, una gran cantidad de expresiones de navegación en condiciones y otras comparaciones, del tipo `expr == null`. Estas expresiones admiten una sola mutación de parte de *ROR* (reemplazar el igual por un distinto), pero en donde *prvo* produce muchas mutaciones. Sin embargo, en estos dos estudios, hay que notar que *prvo* es claramente dominante (como se ve en la Figura 48); en particular, es notable como la dominancia de otros operadores es disminuída significativamente cuando *prvo* es utilizado. Nuestra respuesta entonces para **RQ2**, es que el costo adicional, con respecto a cantidad de mutantes, de utilizar *prvo*, es en general bajo, aunque existen casos en donde el número de mutaciones generadas por *prvo* puede "explotar". Estos casos muestran, en nuestros experimentos, una gran dominancia por parte de *prvo* sobre mutaciones producidas por otros operadores, sugiriendo que existe un margen para realizar optimizaciones mediante priorización de tests/mutantes, una técnica que consiste en ordenar y/o seleccionar tests o mutantes para ser evaluados antes que otros, o en lugar de otros, así

disminuir los recursos necesarios. De todas formas, estos resultados sugieren realizar refinamientos a *prvo*, ya sea añadiendo análisis durante la generación de mutaciones, o implementando nuevas propiedades para configurar a *prvo* de manera más apropiada para cada caso. Esto, puede llevar a lograr una mejor eficiencia, es decir una menor cantidad de mutantes, por parte de *prvo*.

Con respecto a mutantes equivalentes, los resultados son muy interesantes. La cantidad de mutantes equivalentes producidos por *prvo* fue muy poca. Esto, es muy importante, dado que cuando un operador produce muchos mutantes equivalentes, es decir, mutantes para los cuales no existe ningún escenario o entrada para el cual éste se comporte de manera diferente al programa original, lo que lo hace imposible de detectar, bajando el valor de mutation score correspondiente de manera artificial, resultando en una evaluación engañosa del test suite. Tener entonces una pequeña cantidad de dichos mutantes es un buen resultado para *prvo*. Vale la pena aclarar que si bien lo deseable es evitar por completo la generación de mutantes equivalentes, esto es en general imposible, principalmente por que al ser un problema indecidible (el detectar si un programa es equivalente a otro), no es posible tener una implementación de una herramienta de mutación que evite por completo la generación de los mismos.

Los resultados obtenidos de *Mutation Score* 50, *Toughness* 51, y *Toughness de solo detectados* 52, se condicen con los resultados de dominancia y mutantes equivalentes generados por *prvo*. El mutation score al utilizar *prvo* (en color rojo) se mantiene en general levemente por encima del obtenido al utilizar solamente los operadores suficientes (en color verde), esto junto a los resultados de dominancia, indica que la cantidad de mutantes triviales generados por *prvo* es poca (lo que llevaría a un aumento del mutation score), y que la generación de mutantes equivalentes también lo es, dado que éstos llevarían a una disminución del valor de mutation score obtenido.

Los resultados de *Toughness*, tanto si se consideran todos los mutantes o solo se consideran aquellos que fueron detectados, apoyan la conclusión anterior de que *prvo* no solo que no genera mutantes triviales, sino que además, éstos son en general difíciles de matar. La utilización de test suites conformadas por una gran cantidad de tests con mayormente una cobertura de ramas alta 11 (lo que podría llevar a considerar a éstas como "suficientemente buenas") permitiría considerar como stubborn a los mutantes de *prvo* que sobrevivieron y no fueron determinados como equivalentes.

Project	Apache	AvlTree	BinHeap	Queue	TreeSet	NCLL	BSTree
Tests	1292	2564	2851	2375	2662	2300	3023
Branch Cov	95.33 %	99.9 %	84.32 %	87 %	68.21 %	81.38 %	88 %

Tabla 11: Cantidad promedio de tests y cobertura de ramas para cada caso de estudio.

6.8 AMENAZAS A LA VALIDEZ

Nuestra evaluación experimental está limitada a implementaciones relativamente pequeñas de colecciones en *Java*. Las razones por las que seleccionamos estas implementaciones como casos de estudio fueron parcialmente descriptas anteriormente, y esencialmente tiene que ver con que estos proyectos son buenos representantes de implementaciones orientadas a objetos con usos sofisticados de expresiones de navegación (por ejemplo, utilizan tipos de datos recursivos, y múltiples referencias a objetos distintos del mismo tipo). Otra razón por la que nos limitamos a estos casos de estudio tiene que ver con el alto costo de análisis de mutación. Especialmente en nuestro contexto, en donde las características de nuestra evaluación nos lleva a la necesidad de realizar numerosos análisis de mutación, en particular, computar los grafos de subsuma de mutantes haría inviable esta evaluación si fuéramos a utilizar proyectos más grandes.

Nuestro análisis está también atado a una forma específica de generar tests, ésta se basa en el uso de dos herramientas particulares, Evosuite y Randoop específicamente. Esto está motivado por el tipo de test suites que resultan de relevancia para análisis de subsuma. Dado el impacto que tiene el presupuesto de tiempo que se brinda a las herramientas que utilizamos para generar las test suites, nuestros resultados podrían estar siendo afectados por el mismo. Para contrarrestar esta potencial amenaza, corrimos los experimentos con distintos presupuestos de tiempo, lo que llevó a test suites de distintos tamaños, pero sin embargo no obtuvimos resultados diferentes. El costo de tiempo llevó a realizar esta evaluación previa en un subconjunto de los casos de estudio.

La mayor parte de nuestra evaluación experimental es objetiva, es decir, provienen de análisis algorítmicos que retornan un valor particular, por ejemplo la noción de subsuma y la cantidad de mutantes generados, éstas están calculadas por nuestra implementación de *μJava++* o por los scripts utilizados para automatizar los experimentos. Hicimos nuestro mejor esfuerzo para garantizar la validez de nuestros resultados, pero aún así nuestra implementación podría contener defectos que afecten a nuestros resultados. En contraposición, la equivalencia de programas, requerida para analizar mutantes equivalentes, es un problema indecidible. Varias técnicas han sido propuestas para resolver

	TreeList	AvlTree	BinHeap	TreeSet	NCLL	Queue	BSTree
Sin PRVO	380	233	352	410	504	68	274
Con PRVO	458	304	788	700	519	78	334

Tabla 12: Mutantes generados por caso de estudio

	Apache	AvlTree	BinHeap	TreeSet	NCLL	Queue	BSTree
Mutantes totales	78	71	436	290	15	10	60
Equivalentes	2	14	27	31	2	0	0

Tabla 13: Mutantes *prvo* equivalentes por caso de estudio

este problema (obviamente de manera aproximada), sin embargo, la verificación manual sigue siendo una de las más utilizada en la práctica, y es la que hemos utilizados nosotros. Afortunadamente este análisis estuvo limitado a un número muy pequeño de mutantes y cualquier duda sobre la equivalencia de uno, fue siempre anotada en desventaja de *prvo*, es decir, ante la duda, se consideró el mutante como equivalente. Creemos que los errores que podemos haber cometido con respecto a equivalencia de mutantes, no afectan de manera significativa nuestras conclusiones.

Nuestro análisis está también limitado a un conjunto arbitrario de operadores de mutación. Podríamos haber considerado otros operadores, por ejemplo los que actúan a nivel de clases (en contraposición de los que usamos que actúan a nivel de métodos) [Kim et al., 2000]. Éstos, están relacionados a nuestro trabajo, en el sentido de que se aplican a programas orientados a objetos, pero se enfocan en aspectos ortogonales como visibilidad (cambiando la visibilidad de métodos y campos en una clase). Decidimos concentrarnos en los operadores que se consideran tradicionales, en el contexto de mutation testing, y creemos que hemos tenido en cuenta a los más significativos, basado en los que están disponibles en las herramientas de mutation testing, y en estudios de operadores suficientes.

Finalmente necesitamos remarcar que subsuma de operadores, no es lo mismo que subsuma de mutantes. El primero es una relación entre operadores que es independiente de que programa se esté mutando, se trata de una relación que o bien se explica por la misma definición de los operadores, o bien (como sucede en la mayoría de los casos) requiere numerosos experimentos para determinar si es posible concluir si un operador subsume a otro. Sin embargo nuestra evaluación de subsuma dinámica de mutantes, permite, de manera indirecta, una evaluación de la primera.

Tabla 14: Resultados de *Dynamic Mutant Subsumption* para *TreeList*, sin *prvo*

DOM NODES	AODU	AOIS	AOIU	AORB	AORS	AORU	ASRS	COD	COI	COR	LOI	ROR
21	0	6	5	4	1	0	0	0	2	2	6	6
16	0	2	4	4	1	0	0	0	2	1	6	4
23	0	8	7	4	2	0	1	0	2	1	9	8
19	0	3	7	4	1	0	0	0	0	1	8	4
18	0	4	4	4	0	0	1	1	2	1	6	5
28	0	11	5	4	2	0	3	1	2	1	8	9
25	1	8	3	4	1	1	0	0	2	0	9	5
19	0	5	3	4	2	0	0	0	2	2	6	5
20	0	6	5	4	2	0	1	1	2	2	8	5
25	1	7	6	4	2	1	1	0	2	2	9	9
18	0	4	5	4	2	0	2	1	3	0	8	6
23	0	5	5	2	0	0	1	0	1	2	7	9
26	1	3	6	3	2	1	2	0	1	2	7	9
24	1	6	3	2	1	1	0	0	1	2	8	9
21	0	8	3	3	1	0	0	0	2	1	8	3
26	1	5	6	2	3	1	1	0	2	1	7	10
24	0	8	6	5	1	0	1	1	2	3	8	6
23	0	6	7	3	2	0	3	0	1	1	9	9
21	1	3	4	4	1	1	0	0	1	3	6	8
20	0	6	4	3	2	0	0	1	2	2	6	2
18	1	4	6	3	1	1	0	0	4	1	6	6
28	0	8	5	4	3	0	2	0	1	1	10	8
20	0	5	4	3	0	0	3	1	1	0	7	6
20	0	4	5	3	2	0	1	0	1	2	7	8
22	0	7	6	5	1	0	1	1	3	2	8	7
26	1	6	7	3	1	1	2	1	2	1	8	6
15	1	5	7	3	1	1	1	2	4	2	5	11
22	0	6	5	5	1	0	3	0	1	1	10	7
21	0	5	4	5	2	0	2	0	2	1	9	5
22	0	6	3	2	1	0	1	0	2	2	7	6

Tabla 15: Resultados de *Dynamic Mutant Subsumption* para *TreeList*, con *prvo*

DOM NODES	PRVO	AODU	AOIS	AOIU	AORB	AORS	AORU	ASRS	COD	COI	COR	LOI	ROR	PRVO PURE
22	5	0	4	3	3	1	0	0	0	1	2	7	7	4
30	5	0	7	6	4	1	0	2	1	1	2	11	6	5
28	2	0	8	6	4	3	0	2	1	2	2	9	12	2
23	7	0	4	3	2	1	0	1	0	2	1	6	3	6
17	4	1	9	5	3	2	1	0	0	3	0	6	8	0
32	8	1	5	2	2	1	1	1	0	2	1	8	8	8
21	6	0	3	4	4	0	0	0	0	1	1	5	2	5
27	6	0	8	5	4	1	0	1	0	2	2	8	7	6
29	7	0	4	4	2	2	0	1	0	2	1	6	7	6
23	3	0	5	5	3	1	0	1	1	3	3	8	6	2
29	9	0	8	2	2	2	0	2	0	3	1	8	6	9
26	6	0	4	4	3	2	0	1	0	1	1	6	7	5
23	8	0	3	3	3	0	0	3	1	2	1	7	5	8
25	6	1	3	4	2	0	1	0	0	2	2	6	7	6
24	5	0	3	2	3	2	0	0	0	2	3	6	7	4
21	5	0	8	4	4	2	0	1	1	2	1	9	4	2
31	5	1	9	3	4	1	1	1	0	2	0	11	9	5
20	6	0	4	4	2	1	0	0	1	2	1	4	4	5
23	6	0	5	3	2	1	0	1	0	3	2	7	7	6
22	6	0	5	3	3	2	0	1	1	1	2	7	5	5
28	7	0	7	5	3	1	0	1	0	2	1	8	8	7
30	5	0	8	6	2	1	0	2	1	2	2	8	8	2
29	9	0	4	5	3	2	0	3	1	1	0	8	7	8
23	5	0	7	4	5	3	0	2	1	2	3	6	6	3
24	3	0	8	5	3	0	0	2	0	3	1	7	7	1
24	5	0	9	4	3	1	0	1	0	0	0	9	4	4
36	8	0	10	4	4	2	0	1	0	2	2	7	11	8
22	4	0	4	4	3	1	0	0	1	2	2	5	4	3
24	6	1	5	2	2	0	1	1	0	3	2	6	7	6
20	6	0	3	3	2	2	0	0	0	0	0	6	2	4

Tabla 16: Resultados de *Dynamic Mutant Subsumption* para *AvlTree*, sin *prvo*

DOM NODES	AOIS	AOIU	AORB	COI	LOI	ROR
16	4	1	3	0	5	6
22	6	0	3	0	5	10
14	1	1	3	0	4	6
19	5	1	4	0	4	6
14	3	1	3	0	3	5
18	4	0	1	0	4	10
17	5	1	3	1	3	6
19	5	1	3	0	5	6
12	2	0	3	1	2	6
13	5	0	2	0	0	6
17	7	2	2	0	2	5
11	2	0	3	1	3	4
9	2	0	3	0	1	3
13	3	0	3	1	2	6
19	5	2	2	0	4	7
23	8	1	3	0	4	9
18	6	0	2	1	2	10
12	2	0	3	0	4	3
16	2	0	3	1	4	8
14	3	0	2	0	4	6
16	6	1	3	1	1	6
12	2	0	2	0	3	5
17	5	0	4	0	2	6
10	2	0	3	0	2	4
16	3	1	3	0	4	5
20	4	2	3	0	3	9
22	6	1	2	1	4	11
9	2	0	2	0	1	4
17	3	1	3	0	4	7
16	5	0	4	1	2	6

Tabla 17: Resultados de *Dynamic Mutant Subsumption* para *AvlTree*, con *prvo*

DOM NODES	PRVO	AOIS	AOIU	AORB	COI	LOI	ROR	PRVO PURE
19	10	2	0	2	0	2	4	9
24	9	5	1	2	0	2	6	8
23	12	4	0	2	1	1	6	11
20	11	2	1	2	0	2	5	8
16	8	2	0	2	0	2	5	5
21	11	4	0	1	0	2	5	9
19	12	4	1	0	0	3	3	11
18	9	1	1	3	0	3	4	6
15	10	2	0	1	0	1	2	9
22	10	3	1	4	0	2	4	8
14	10	2	0	1	0	0	3	8
20	12	3	0	2	1	1	5	10
20	11	2	0	1	1	3	6	11
23	13	3	1	1	0	2	6	11
22	10	5	1	2	0	2	4	8
18	8	2	0	4	0	2	3	7
16	11	1	0	1	0	1	3	10
18	12	4	0	2	0	0	3	9
16	8	4	0	2	0	0	3	7
15	10	1	0	0	0	1	5	8
14	8	2	0	1	1	2	4	6
23	9	4	2	3	0	1	6	7
19	12	4	0	1	0	1	3	10
21	12	2	0	2	1	2	7	9
22	12	2	0	3	1	2	6	10
21	11	5	1	2	0	1	3	9
24	13	4	0	3	0	2	5	10
12	7	2	0	1	0	1	2	6
24	10	6	1	2	0	4	5	8
15	10	2	0	2	0	1	2	8

Tabla 18: Resultados de *Dynamic Mutant Subsumption* para *TreeSet*, sin *prvo*

DOM NODES	AOIS	AOIU	COI	COR	LOI	ROR
14	0	0	10	0	0	11
12	2	0	7	0	1	8
11	0	0	8	0	0	10
13	2	0	6	1	1	10
12	0	0	5	0	0	11
18	2	0	11	0	1	13
15	2	0	5	0	1	12
17	1	1	7	0	0	15
14	1	0	7	0	0	12
16	1	0	11	0	0	11
12	1	0	5	2	0	11
12	1	0	6	1	0	10
10	1	0	5	1	0	8
22	2	0	11	0	1	16
13	2	0	7	1	0	10
18	2	1	7	1	1	13
12	0	0	7	0	0	12
14	1	0	6	1	0	12
14	1	0	6	0	0	12
14	0	0	8	1	0	13
18	1	0	8	1	0	15
11	1	0	6	0	1	9
11	1	0	8	1	0	9
13	0	0	9	0	0	10
17	2	0	8	0	1	13
20	3	0	9	1	1	13
17	1	1	9	1	0	13
14	1	0	6	1	0	11
16	2	0	7	1	1	12
17	4	1	5	1	2	11

Tabla 19: Resultados de *Dynamic Mutant Subsumption* para *TreeSet*, con *prvo*

DOM NODES	PRVO	AOIS	AOIU	COI	COR	LOI	ROR	PRVO PURE
25	19	0	0	7	0	0	10	15
20	14	1	0	8	1	0	9	9
17	13	0	0	7	0	0	10	7
16	14	0	0	7	0	0	9	6
22	18	0	0	5	2	0	13	9
19	13	0	0	7	1	0	9	8
20	15	1	0	6	1	0	9	9
21	12	2	0	5	1	1	11	6
20	18	0	0	6	0	0	7	13
30	22	2	0	8	0	1	12	15
16	12	0	0	7	1	0	7	7
18	9	3	0	3	1	1	8	6
14	11	1	0	5	0	1	7	6
22	17	1	0	4	1	0	10	11
22	15	1	0	4	1	0	8	12
26	17	2	0	8	1	1	11	10
17	14	0	0	4	0	0	7	10
18	12	2	0	9	1	0	11	5
15	9	0	1	6	1	0	9	4
20	15	0	0	7	0	0	10	8
18	14	1	0	7	1	0	11	5
23	17	2	0	7	1	0	10	12
17	15	1	0	6	1	0	9	7
24	20	0	0	7	0	0	9	14
22	13	1	0	9	1	0	10	7
20	15	0	0	6	0	0	11	8
25	16	2	0	6	0	1	11	10
18	13	2	0	7	0	1	8	6
22	17	0	0	5	1	0	7	13
24	20	0	0	8	0	0	8	13

Tabla 20: Resultados de *Dynamic Mutant Subsumption* para *NodeCachingList*, sin *prvo*

DOM NODES	AOIS	AOIU	AORB	AORS	COD	COI	COR	LOI	ROR
53	14	7	2	4	0	5	2	7	21
52	12	9	3	5	0	3	2	6	20
60	19	12	2	2	2	7	1	15	17
46	8	6	6	3	0	6	1	7	17
57	12	12	4	4	0	4	2	12	20
51	11	11	3	4	0	6	0	6	19
51	12	11	2	3	0	5	2	6	21
51	12	9	2	6	0	6	2	9	20
60	12	16	4	4	0	7	2	9	17
49	12	8	2	1	0	5	1	9	19
48	16	7	6	2	0	5	0	12	11
46	12	7	3	2	0	5	1	11	18
57	13	15	2	5	0	6	1	7	17
54	13	13	7	0	0	3	3	12	14
44	13	9	3	3	0	6	1	7	12
62	12	12	4	2	0	5	3	13	20
58	17	6	4	2	0	6	2	12	14
50	10	15	2	2	0	5	2	7	18
57	16	11	5	5	0	5	3	7	19
51	12	8	2	5	0	6	3	8	21
47	12	9	4	2	0	6	3	9	14
49	14	10	5	2	0	4	2	11	18
49	11	9	6	2	0	6	1	9	16
52	6	14	2	6	0	7	2	10	17
54	11	12	3	2	0	8	2	9	17
65	16	18	2	7	1	6	1	14	12
61	11	16	2	3	1	8	1	6	21
60	16	8	5	4	0	4	3	8	20
44	7	8	3	4	0	6	1	7	15
49	9	11	2	3	0	7	3	7	18

Tabla 21: Resultados de *Dynamic Mutant Subsumption* para *NodeCachingList*, con *prvo*

DOM	NODES	PRVO	AODU	AOIS	AOIU	AORB	AORS	COD	COI	COR	LOI	ROR	PRVO	PURE
	72	1	0	17	15	5	4	0	7	2	18	17		1
	65	1	0	17	10	6	2	0	5	1	11	20		1
	54	1	0	16	10	7	3	0	2	2	6	18		1
	56	2	0	15	11	6	5	1	7	1	6	15		2
	54	2	0	17	5	5	1	1	3	2	14	15		2
	50	1	0	13	9	2	2	0	5	2	10	16		1
	57	1	0	14	9	3	1	0	7	3	11	22		1
	59	3	0	18	6	6	2	0	7	0	14	15		2
	70	3	0	17	9	5	4	0	3	3	16	21		3
	53	0	0	12	10	4	3	0	3	2	10	20		0
	60	2	0	16	11	3	3	0	4	3	13	17		2
	52	2	0	11	9	3	1	0	5	3	11	19		2
	63	2	0	15	11	5	2	1	5	2	14	16		2
	60	3	0	17	9	7	3	0	5	2	9	16		3
	53	1	0	13	7	5	3	0	3	3	8	18		1
	63	1	0	14	15	6	4	0	3	1	13	17		1
	62	2	0	18	8	5	4	0	7	3	10	18		2
	51	3	0	16	3	3	4	0	5	3	9	19		2
	59	2	0	13	10	4	3	0	7	3	6	19		2
	64	1	0	15	15	3	3	0	4	3	12	24		1
	69	1	0	11	10	3	5	0	7	3	15	22		1
	56	2	0	10	10	4	0	0	5	2	11	22		2
	53	2	1	6	8	4	2	1	5	1	10	21		2
	49	2	0	11	12	5	3	0	4	1	10	15		1
	62	1	0	13	11	2	4	0	7	2	13	24		1
	59	1	0	13	11	4	1	0	6	1	10	19		1
	56	2	0	11	12	5	3	1	6	1	6	16		2
	65	2	0	17	10	5	3	0	6	3	8	18		2
	63	2	0	12	11	5	4	0	3	2	17	17		2
	58	2	0	15	10	3	4	1	5	3	14	19		2

Tabla 22: Resultados de *Dynamic Mutant Subsumption* para *Queue*, sin *prvo*

DOM NODES	AOIS	AOIU	AORS	COI	LOI	ROR
8	2	1	1	2	1	4
10	2	2	1	3	1	5
9	2	1	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5
10	2	2	1	3	1	5
10	2	2	1	3	1	5
10	2	2	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5
10	2	2	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5
9	2	1	1	3	1	5
11	2	3	1	3	1	5
10	2	2	1	3	1	5

Tabla 23: Resultados de *Dynamic Mutant Subsumption* para *Queue*, con *prvo*

DOM NODES	PRVO	AOIS	AOIU	AORS	COI	LOI	ROR	PRVO PURE
13	5	2	2	1	2	1	3	5
12	5	2	2	1	1	1	2	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
12	5	2	2	1	1	1	2	5
11	5	2	1	1	1	1	2	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
12	5	2	1	1	2	1	3	5
12	5	2	1	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5
13	5	2	2	1	2	1	3	5

Figura 48: Summary of dynamic subsumption analysis

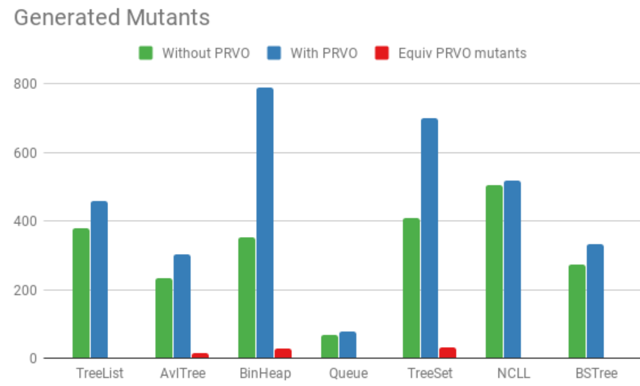


Figura 49: Comparación de mutantes generados.

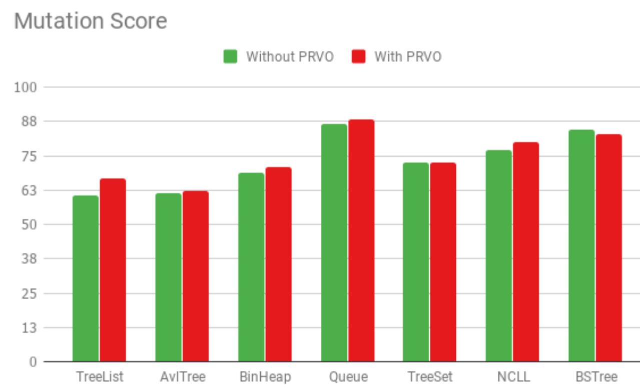


Figura 50: Comparación de *Mutation score*.

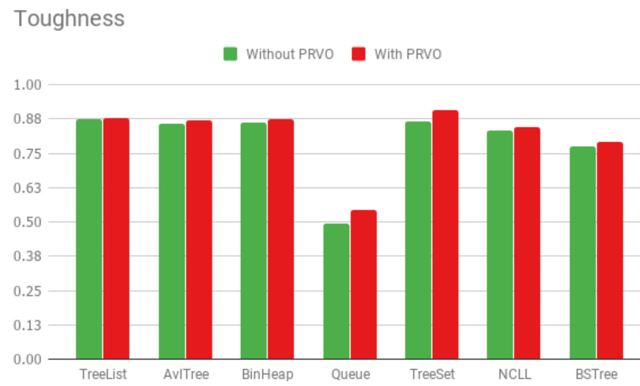


Figura 51: Comparación de *Toughness*, cuantos tests "resiste" un mutante para ser detectado, un valor de 1,0 representa un mutante no detectado.

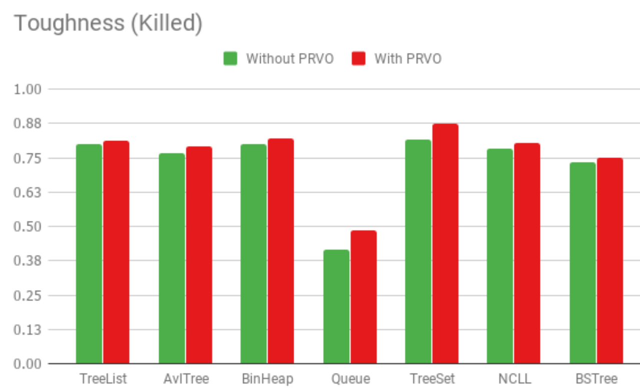


Figura 52: Comparación de *Toughness*, cuantos tests "resiste" un mutante que fue detectado.

7

Mutación de expresiones de navegación en reparación de programas

7.1 Reparación automática de programas

La aparición de técnicas automáticas para testing y localización de fallas, es decir, detectar en donde puede estar una falla, una vez detectada, llevó a la investigación de técnicas para, una vez detectada, poder reparar dicha falla de manera automática. Pero que significa reparar una falla?

> Dado un programa **P** y una especificación **E** del mismo, si **P** no cumple con sus especificaciones (**E**), reparar al programa significa encontrar un variante **P'** mediante modificaciones sintácticas tal que ésta cumpla con las especificaciones dadas por **E** [1].

A partir del trabajo de [Arcuri and Yao, 2008], muchas herramientas que intentan atacar a este problema han surgido. Incluso cuando la idea de reparación automática de fallas es atractiva, reparar automáticamente defectos de programas arbitrarios es inviable. Por lo tanto, la reparación automática de programas debe sacrificar completitud. Varias técnicas efectivas para reparación de programas recurren a explorar un conjunto grande (aunque limitado) de candidatos de reparación obtenidos por modificaciones sintácticas de un programa con fallas. Además, para que estas técnicas escalen razonzablemente, el espacio de candidatos a reparación debe ser acotado, limitando el conjunto de modificaciones sintácticas a considerar (por ejemplo, no considerar modificaciones a partes de una sentencia), o no explorando exhaustivamente el conjunto acotado de candidatos a reparación (por ejemplo, usando algorítmos genéticos en lugar de búsqueda exhaustiva).

[1] Esta definición no considera reparaciones en tiempo de ejecución, es decir, aquellas que no producen una variante del programa original como reparación

7.1.1 *Oráculo de reparación*

En general el proceso de reparación sigue, a rasgos muy generales, el siguiente algoritmo:

```
repair(P, E) {
  current = P
  while (!isValid(current, E) && !boundsReached()) {
    current = nextFix();
  }
  return current;
}
```

Donde **P** representa el programa a reparar, y **E** las especificaciones que se deben satisfacer. El proceso búsca candidatos a reparaciones, evaluando a éstos con respecto a las especificaciones provistas. Claramente el proceso está acotado a un conjunto finito de posibles reparaciones.

Evidentemente, es necesario definir que se usa como especificaciones para el programa. Los primeros trabajos sobre reparación automática [Staber et al., 2005; Arcuri and Yao, 2008], utilizaban especificaciones formales en forma de pre y post condiciones, o descripciones lógicas provistas en algún formalismo lógico apropiado. Sin embargo, muchos de las últimas técnicas utilizan tests como especificaciones. La razón principal detrás de esto, es el argumento de que la producción de especificaciones formales es un proceso costoso, a la vez que es muy raro encontrarlas ya provistas previo al proceso de reparación, mientras que los tests son menos costosos de proveer y es mucho más común encontrarlos ya provistos antes del proceso de reparación.

Por ejemplo, *GenProg* [Le Goues et al., 2012] usa computación evolutiva para evolucionar sintácticamente un programa hasta que una reparación aceptable es encontrada. Cada reparación candidata (modificación sintáctica) es aplicada al programa original para producir uno nuevo cuya aptitud es evaluada utilizando un test suite. Modificaciones sintácticas en partes de una sentencia no son consideradas para limitar el espacio de candidatos, y la función de aptitud es utilizada para mantener una población reducida de candidatos a lo largo del proceso de computación evolutiva.

Tests como especificaciones

Las herramientas actuales, en general, utilizan tests como especificaciones parciales del programa a reparar. Esto, tiene como consecuencia la generación de reparaciones espurias, es decir, reparaciones que si bien logran hacer “pasar”

```
public boolean find(int e) {
  if (isEmpty()) return false;
  Node current = header;
  boolean found = false;
  while(true) {
    found = current.elem == e;
    current = current.next;
  }
  return found;
}
```

Figura 53: Un método con un defecto que causa *NullPointerException*

a los tests, no resuelven el problema. En muchos casos, enmascarar un defecto puede causar que los tests que antes detectaban una falla, ahora dejen de hacerlo. A modo de ejemplo, en la Figura 53 se muestra un método *find(int)* que tiene un defecto, la condición del **while** debería incluir que **current** no sea *null* y que el elemento aún no fue encontrado. Sin embargo, la reparación en la Figura 54 enmascara al defecto. Si nuestros tests solo utilizan listas en donde el elemento no está, y por lo tanto se espera que el método retorne *false*; o el elemento a buscar está en la última posición, y se espera que retorne *true*. La reparación anterior es espuria, es decir, si bien logra que los tests terminen correctamente, no repara realmente el programa.

Esta observación llevó a un trabajo de investigación en donde se evaluaron un conjunto de herramientas actuales de reparación automática de programas basadas en tests como oráculo de reparación [Zemín et al., 2017]. Al contrario de trabajos previos que se basaron en tests o inspección manual de las reparaciones provistas por las herramientas evaluadas (*GenProg*, *Angelix*, *AutoFix*, y *Nopol*, entre otras), en éste, se utilizaron especificaciones formales y *PEX* [Tillmann and de Halleux, 2008] para evaluar las reparaciones obtenidas.

Los resultados obtenidos, logrados sobre el benchmark de reparación de programas *IntroClass* [Le Goues et al., 2015], indican que al evaluar las reparaciones producidas utilizando especificaciones formales, éstas son inválidas, es decir, el programa obtenido no cumple con las especificaciones del problema que deben resolver. El porcentaje de reparaciones para *GenProg*, que son consideradas como efectivas, es solo un 1.57 % (18 de 1.143 fallas fueron reparadas) de las cuales 8 reparan errores de typos en cadenas asociadas a los mensajes de salida del programa. *Angelix* solo es capaz de reparar 41 de 232 (17,67 %), *Nopol* solo 6 de 297 (2 %), y finalmente *AutoFix* no pudo reparar ningún programa de manera

```
public boolean find(int e) {
  if (isEmpty()) return false;
  Node current = header;
  boolean found = false;
  while(true) {
    try {
      found = current.elem == e;
      current = current.next;
    } catch (NullPointerException e) {
      break;
    }
  }
  return found;
}
```

Figura 54: Reparación que enmascara el defecto en 53

correcta[2]. Por el otro lado, incrementar la cantidad de tests disminuía drásticamente las reparaciones espurias, pero también las reparaciones en general. En otros casos, las herramientas no soportaban el aumento en la cantidad de tests.

7.1.2 *Granularidad de las reparaciones*

Como mencionamos anteriormente, el conjunto de candidatos a reparación a tener en cuenta, debe ser finito y su tamaño afecta directamente a los recursos necesarios para reparar el programa. Esto, genera una necesidad de balancear, capacidad de reparación, es decir, que tipo de fallas se pueden reparar, con recursos necesarios. Por ejemplo, el caso de *GenProg*, se basa en mover, duplicar, o eliminar código. La idea es que muchas veces un desarrollador escribe varias veces el mismo código, por ejemplo, sumarle 1 a una variable. Por otro lado, en *SPR* [Long and Rinard, 2015], se generan reparaciónes abstractas como agregar una condición **C** antes de la ejecución de una sentencia, la cual luego será reemplazada por una condición concreta en una etapa posterior, esto sumado a la eliminación, y duplicación de código existente. En *PAR* [Kim et al., 2013], las modificaciones para reparar el programa son aprendidas a partir de patrones en reparaciones escritas manualmente. Por esto, el número de candidatos a

[2] Algunas herramientas no soportaban ciertas características (retornar una cadena), y en otras que requerían traducir a otro lenguaje, la misma reparaba el defecto

```
boolean add(Object arg) {
  SListNode freshNode = new SListNode();
  freshNode.value = arg;
  boolean added = false;
  if (this.header == null) {
    this.header = freshNode;
    added = true;
  } else {
    SListNode current = this.header.next; //BUG: ignora primer nodo
    while (current.next != null && current.value != arg) {
      current = current.next;
    }
    if (current.value != arg) {
      current.next = freshNode;
      added = true;
    }
  }
  if (added) {
    size = size - 1; //BUG: decrementa size
  }
  return added;
}
```

Figura 55: Ejemplo de código con dos bugs que requieren modificaciones intra-sentencia

considerar como reparaciones es significativamente reducido, lo que a su vez reduce el tipo de errores que la técnica puede ser capaz de reparar.

Modificaciones a partes de sentencias, es decir, aquellas que alteran expresiones dentro de una sentencia, son en general no consideradas por técnicas de reparación de programas. Una limitación principal al considerar a éstas, es la explosión en el espacio de candidatos de reparación. Técnicas que utilizan operadores de mutación para producir las modificaciones sintácticas, y que incluyen modificaciones a partes de sentencias, requieren limitar el conjunto de mutaciones (por ejemplo, [Gopinath et al., 2011]), reduciendo la clase de fallas que éstas pueden intentar reparar.

Los efectos que tienen estas restricciones sobre el espacio de candidatos a considerar para la reparación, pueden observarse en la Figura 55 donde se muestra el método *add(Object)* de un conjunto. Éste contiene dos defectos: primero, cuando el conjunto no está vació, se comienza el recorrido del mismo a partir del nodo siguiente al inicial (el cual podría ser *null*, resultando en un error en la línea siguiente); segundo, cuando un nuevo nodo es agregado a la lista sobre la cual está implementado el conjunto, el tamaño de la misma (atributo *size*) se decrementa. Para reparar al método es necesario realizar dos cambios,

ambos dentro de una sentencia. Si bien sería posible conseguir el código correcto en otro lado del programa, **SListNode current = this.header;** y **size = size + 1** respectivamente.

Mutación en reparación

El uso de mutación en reparación de programas parece razonable, si se utiliza mutación para emular defectos reales, podría utilizarse para emular reparaciones. Herramientas de reparación basada en mutación existen, [Debroy and Wong, 2010; Wang et al., 2018a] son ejemplos recientes. El argumento de reparación basada en mutación, es que existen mutaciones que si se fueran a combinar, se cancelarían, por ejemplo:

```
for (int i = 0; i < lenght; i++)...
Δfor (int i = 0; i > lenght; i++)...
```

donde la primera sentencia se puede mutar, aplicando un cambio de operador relacional, a la segunda, marcada por Δ, que a su vez se puede mutar al código original aplicando el mismo operador de mutación. Aunque no siempre se puede deshacer una mutación aplicando otra que sea sintácticamente inversa. Volviendo al ejemplo anterior, el mutante:

```
for (int i = 0; i > lenght; i++)...
```

puede ser restaurado, semánticamente, generando el mutante:

```
for (int i = 0; i != lenght; i++)...
```

Existen también casos donde varias mutaciones pueden corregir el comportamiento. Esto lleva a la idea de que si consideramos el programa con fallas P_b y el original sin fallas P_o, se puede definir el segundo en términos del primero como P_b = mutate(P_o, M) donde M representa una secuencia de mutaciones y `mutate` es un programa que aplica dicha secuencia a un programa. En general como dijimos anteriormente, muchas mutaciones tienen su inversa, ya sea sintáctica o semántica, lo que lleva a definir el problema de reparación como encontrar una secuencia de mutaciones M′ tal que P_o = mutate(P_b, M′).

La reparación de programas mediante mutación puede describirse como una búsqueda exhaustiva que, dado un programa defectuoso y una especificación del mismo (que como vimos puede ser formal o mediante tests), y un conjunto de operadores de mutación:

1. Considera el programa a reparar como el candidato a reparación inicial.

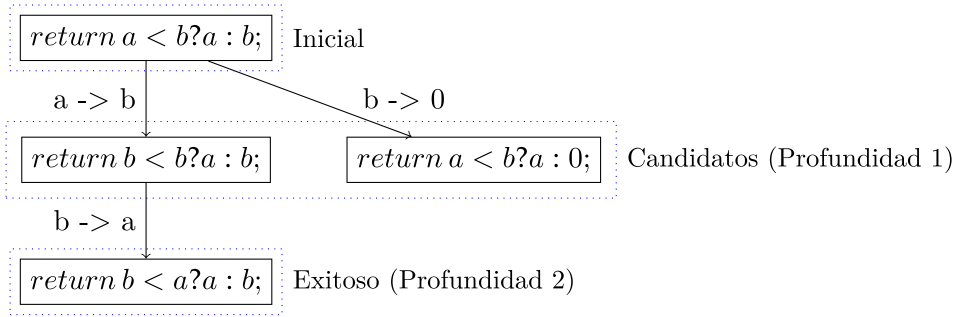


Figura 56: Reparación del un programa que calcula el máximo de dos números

Search Depth	No. of Mutants (Fix Candidates)
1	40
2	1,604
3	64,684
4	$>$ 20 million

Tabla 24: Mutantes generados para *getNode* 57, considerando 4 líneas y 18 operadores, a medida que la profundidad aumenta.

2. Si **P** es un candidato a reparación, y **Q** es el resultado de aplicar una mutación a **P**, entonces **Q** también es un candidato a reparación.

3. Un candidato **S** es exitoso si satisface las especificaciones provistas.

La definición previa de reparación basada en mutación deja en claro que el espacio de candidatos a reparación depende del número de mutaciones (**b**) a considerar, lo que influye en cuan "ancho" es el árbol de búsqueda, y el número máximo de mutaciones sucesivas permitidas (**d**) para generar a los candidatos, lo que afecta la profundidad de la búsqueda. El espacio de búsqueda queda entonces definido por una suma geométrica $\frac{b^{d+1}-1}{b-1}$ $(O(b^d))$. Si consideramos el método en la Figura 57, y solo cuatro líneas del mismo, al utilizar 18 operadores de mutación (mediante la herramienta *μJava++*), podemos ver en la Tabla 24, como a medida que aumentamos la cantidad de mutaciones consecutivas (la profundidad en reparación mediante mutación), la cantidad de candidatos a evaluar se vuelve rápidamente inmanejable.

7.1.3 *Striker*

Striker es una herramienta de reparación automática de programas cuya meta principal es:

```
1   Node getNode(int i) {
2     Node current = this.head;
3     Node result = null;
4     int current_index = 0;
5     while (result == null && current != null) {
6       if (i == current_index ) {
7         result = current;
8       }
9       current_index = current_index + 1;
10      current = current.next;
11    }
12    return result;
13  }
```

Figura 57: Código de ejemplo, método que obtiene el i-ésimo nodo de una lista

> Proveer una técnica automatizada y eficiente para reparar programas anotados con especificaciones (en términos de pre y post condiciones) corrigiendo errores que resultan como una consecuencia de ocurrencias simultaneas de un número de errores sintácticos dentro de sentencias de un programa.

Es necesario remarcar la necesidad de especificaciones asociadas al programa. Así también, los errores que se consideran son defectos, o mutaciones, dentro de sentencias. En particular no se apunta a reparar programas que requieran agregar, o eliminar, código. Algunos de los defectos reparables por *Striker* se muestran en la Figura 58.

Si bien muchas herramientas de reparación automática de programas utilizan mutación, *Striker* es una de las más flexibles con respecto a operadores soportados, entre ellos *prvo*; soporte para modificaciones a partes de sentencias; y fallas que requieren las modificación de varias sentencias. Striker utiliza *TACO* [Galeotti et al., 2013] para evaluar candidatos a reparación, lo que requiere proveer especificaciones formales en lugar de un test suite, y JML RAC [Leavens et al., 2002] como técnica de evaluación rápida basada en escenarios previamente encontrados para los cuales un candidato a reparación no cumplía con las especificaciones. Sin entrar en detalles, esta herramienta es capaz de expandir las fallas soportadas para reparar, agregando varios operadores en los que incluye a *prvo*, y permitir la reparación de aquellas que requieren modificaciones intra-sentencias así como en múltiples sentencias, mediante el uso de técnicas de poda innovadoras.

$$
\begin{aligned}
5_a:&\quad (result \mathrel{!=} \mathsf{null} \mathrel{\&\&} current \mathrel{!=} \mathsf{null})\\
5_b:&\quad (result == \mathsf{null} \mathrel{\&\&} current == \mathsf{null})\\
5_c:&\quad (result == \mathsf{null} \mathrel{||} current \mathrel{!=} \mathsf{null})\\
6_a:&\quad (i == current_index + 1)\\
6_b:&\quad (i \mathrel{!=} current_index)\\
9_a:&\quad current_index = current_index - 1\\
10_a:&\quad current.next = current
\end{aligned}
$$

Figura 58: Algunos defectos en el método *getNode* 57 que pueden modelarse mediante mutaciones y ser reparados por *prvo*.

Bajo esta herramienta es que se pudo observar como *prvo* era capaz de reparar ciertas fallas que de otra forma no eran reparables, como por ejemplo el caso de *add(int)* en la Figura 55, en donde si bien el segundo defecto es reparable por *AORB* el cual reemplaza el operador incorrecto (-) por el que corresponde (+), el primer defecto solo puede ser reparado por *prvo* al eliminar el acceso al campo *next*. Si bien el enfoque principal de esta tesis se encuentra en *prvo* como un operador de mutación en el contexto de mutation testing, es interesante mostrar su contribución en el contexto de reparación.

7.1.4 *Evaluación de Stryker*

Si bien la evaluación de *Stryker* se encuentra enfocada hacia la técnica de poda del espacio de búsqueda de candidatos a reparación, lo que a su vez permite la utilización de una mayor cantidad de operadores de mutación orientados hacia la reparación de fallas intra-sentencia, en el contexto de esta tesis nos interesan los resultados asociados a *prvo* y su utilidad en el campo de reparación automática de programas mediante mutación.

En los experimentos realizados para la evaluación de *Stryker*, se utilizaron implementaciones en *Java* de estructuras que representan colecciones. Estas clases, para las cuales se definieron especificaciones en forma de contratos *JML*, incluyendo clausulas `requires/ensures`, funciones de variantes de ciclos e invariantes de clases, son las siguientes:

- **SinglyLinkedList :** Una implementación de listas simplemente encadenadas. Donde consideramos un método *contains* para evaluar pertenencia, *getNode* para obtener el i-ésimo elemento en la lista, y el método *insert* para agregar un nuevo elemento a la lista.
- **NodeCachingLinkedList :** Una lista circular, doblemente encadenada y con una cache de nodos. Considerando los métodos *contains*, *inserte*,

y *remove*. Siendo el último de particular interés por almacenar nodos removidos en la cache.

- **BinarySearchTree :** Una implementación de un árbol de búsqueda binario con los métodos *contains*, *insert* y *remove*.
- **BinomialHeap :** Una implementación de colas de prioridad utilizando *binomial heaps*. Considerando los métodos *findMin* (para obtener el mínimo elemento almacenado), *insert*, y *extractMin* para obtener y eliminar el mínimo elemento almacenado.

Los casos de estudio para reparación fueron obtenidos a partir de las versiones originales (correctamente implementadas) al insertar hasta 4 mutaciones por método, y luego se eligieron 5 versiones al azar por cada número de mutaciones insertadas, es decir, 5 programas incorrectos con un bug, 5 con 2, y así hasta 4 bugs.

Por cada sentencia (o línea) mutada se agrego un comentario de línea `//mutGenLimit K` donde `K` corresponde a la cantidad de bugs artificiales introducidos en la misma. Algunos ejemplos de los casos de estudios utilizados se muestran a continuación.

```
/*@ requires true;
  @ ensures \old(Nodes) != null ==>
      \old(\reach(Nodes, BinomialHeapNode, child + sibling)).has(\result);
  @ ensures (\forall BinomialHeapNode n;
      \reach(Nodes, BinomialHeapNode, child + sibling).has(n);
      \result.key <= n.key
      );
  @ ensures (\forall BinomialHeapNode n;
      \reach(Nodes, BinomialHeapNode, child + sibling).has(n);
      \old(n.key) == n.key
      );
@*/
public /* @ nullable @ */BinomialHeapNode extractMin() {
    if (Nodes == null) {
        return null;
    }
    BinomialHeapNode temp = Nodes;
    BinomialHeapNode prevTemp = null;
    BinomialHeapNode minNode = null;
    minNode = Nodes.findMinNode();
    //@decreasing \reach(temp, BinomialHeapNode, sibling).int_size();
    while (temp.key != minNode.key) {
        prevTemp = temp;
        temp = temp.sibling;
    }
    if (prevTemp == null) {
        Nodes = temp.sibling;
    } else {
        prevTemp.sibling = temp.sibling;
    }
    temp = temp.child;
    BinomialHeapNode fakeNode = temp;
    //@decreasing \reach(temp, BinomialHeapNode, sibling).int_size();
    while (temp != null) {
        temp.parent = null;
        temp = temp.sibling;
    }
    if (Nodes == null && fakeNode == null) {
        size = 0;
    } else {
        if (Nodes == null && fakeNode != null) {
       Nodes = fakeNode.reverse( null );
       size--;
        } else {
       if (Nodes != null && fakeNode == null) {
           size--;
       } else {
           unionNodes( fakeNode.reverse( null ) );
           size--;
       }
        }
    }
    return minNode;
}
```

Figura 59: Versión original del método *extractMin* de *BinomialHeap* con los contratos asociados al método.

```
public BinomialHeapNode extractMin() {
    if (Nodes == null) {
        return null;
    }
    BinomialHeapNode temp = Nodes;
    BinomialHeapNode prevTemp = null;
    BinomialHeapNode minNode = null;
    minNode = Nodes.findMinNode();
    while (temp.key != minNode.key) {
        prevTemp = temp;
        temp = this.Nodes.sibling; //mutGenLimit 1
    }
    if (prevTemp == null) {
        Nodes = temp.sibling;
    } else {
        prevTemp.sibling = temp.sibling;
    }
    temp = temp.parent.child; //mutGenLimit 1
    BinomialHeapNode fakeNode = temp;
    while (temp != null) {
        temp.parent = null;
        temp = temp.sibling;
    }
    if (Nodes == null && fakeNode == null) {
        size = 0;
    } else {
        if (Nodes == null && fakeNode != null) {
       fakeNode = fakeNode.reverse( null ); //mutGenLimit 1
       size--;
        } else {
       if (Nodes != null && fakeNode == null) {
           size--;
       } else {
           unionNodes( fakeNode.reverse( null ) );
           size--;
       }
        }
    }
    return prevTemp; //mutGenLimit 1
}
```

Figura 60: Versión con 4 bugs artificiales del método *extractMin* de *BinomialHeap*.

```
/*@
  @ requires true;
  @
  @ ensures (\exists BinTreeNode n;
  @       \old(\reach(root, BinTreeNode, left + right)).has(n) == true;
  @   n.key == k) ==> size == \old(size);
  @
  @   ensures (\forall BinTreeNode n;
  @       \old(\reach(root, BinTreeNode, left + right)).has(n) == true;
  @   n.key != k) ==> size == \old(size) + 1;
  @
  @ ensures (\exists BinTreeNode n;
  @      \reach(root, BinTreeNode, left + right).has(n) == true;
  @       n.key == k);
  @
  @ signals (RuntimeException e) false;
@*/
public boolean insert( int k ) {
   BinTreeNode y = null;
   BinTreeNode x = root;
   //@decreasing \reach(x, BinTreeNode, left+right).int_size();
   while (x != null) {
       y = x;
       if (k < x.key) {
      x = x.left;
       } else {
      if (k > x.key) {
          x = x.right;
      } else {
          return false;
      }
       }
   }
   x = new BinTreeNode();
   x.key = k;
   if (y == null) {
       root = x;
   } else {
       if (k < y.key) {
      y.left = x;
       } else {
      y.right = x;
       }
   }
   x.parent = y;
   size += 1;
   return true;
}
```

Figura 61: Versión original del método *insert* de *BinarySearchTree* con los contratos asociados al método.

```
public boolean insert( int k ) {
    BinTreeNode y = null;
    BinTreeNode x = root;
    while (x != null) {
        y = null; //mutGenLimit 1
        if (k < x.key) {
        x = x.left;
        } else {
        if (k > x.key) {
            x = x.right;
        } else {
            return false;
        }
        }
    }
    x = new BinTreeNode();
    x.key = -k; //mutGenLimit 1
    if (y == null) {
        root = x;
    } else {
        if (k < y.left.key) { //mutGenLimit 1
        y.left = x;
        } else {
        y.right = x;
        }
    }
    x.parent = y;
    size += 1;
    return true;
}
```

Figura 62: Versión con 3 bugs artificiales del método *insert* de *BinarySearchTree*.

```
/*@
  @ requires true;
  @ ensures \result == true <==> (
       \exists LinkedListNode n;
          \reach(header, LinkedListNode, next).has(n)
          &&
          n != header;
          n.value == arg
       );
  @ ensures (\forall LinkedListNode n;
       \old(\reach(header, LinkedListNode, next)).has(n);
       \reach(header, LinkedListNode, next).has(n)
       );
  @ ensures (\forall LinkedListNode n;
       \reach(header, LinkedListNode, next).has(n);
       \old(\reach(header, LinkedListNode, next)).has(n)
       );
  @ signals (Exception e) false;
@*/
public boolean contains( Object arg ) {
   LinkedListNode node = header.next;
        int counter = 0;
        //@decreasing size - counter;
        while (node != header && node.value != arg) {
            node = node.next;
            counter++;
        }
        if (node != header && node.value == arg) {
            return true;
        }
        return false;
}
```

Figura 63: Versión original del método *contains* de *NodeCachingLinkedList* con los contratos asociados al método.

```
public boolean contains( Object arg ) {
    LinkedListNode node = header; //mutGenLimit 1
    int counter = this.maximumCacheSize; //mutGenLimit 1
    while (node == header && node.value != arg) { //mutGenLimit 1
        this.firstCachedNode = node.next; //mutGenLimit 1
        counter = counter - 1; //mutGenLimit 1
    }
    if (node != header && node.value != arg) { //mutGenLimit 1
        return true;
    }
    return false;
}
```

Figura 64: Versión con 4 bugs artificiales del método *contains* de *NodeCachingLinkedList*.

8

Conclusiones

Medir la calidad de un conjunto de tests evaluando su habilidad de detectar fallas artificiales, tal como propone *mutation testing*, ha demostrado se una de las métricas más efectivas en testing, brindando una mejor correlación, que otras métricas, con detección de fallas reales. Sin embargo, algunas limitaciones importantes han sido identificadas en relación a mutación, más notablemente la inhabilidad de operadores de mutación actuales en representar ciertos defectos reales en programas, y los problemas de eficiencia asociados con medir la calidad de un test suite, mediante análisis de mutación. En esta tesis, hemos contribuído a la primera de estas limitaciones, proveyendo un nuevo operador de mutación que aplica a expresiones comúnmente encontradas en programas orientados a objetos, específicamente *expresiones de navegación*. Hemos provisto de motivaciones para el diseño de este operador, al cual llamamos *prvo*, y hemos realizado un análisis que nos permitió mostrar que este operador no es subsumido por operadores existentes, es decir, genera mutantes que constituyen nuevas “obligaciones” para los test suites, dentro de mutation testing. Además, nuestro análisis muestra que, en presencia de *prvo*, el grado de dominancia de otros operadores disminuye mientras que la dominancia de los mutantes de *prvo* es comparable con la de varios de los operadores mas relevantes dentro de los *operadores suficientes*. También analizamos el impacto de *prvo* en la eficiencia de análisis de mutación, al evaluar el incremento en los mutantes generados que implica el añadido de *prvo* al conjunto de operadores a utilizar. Nuestros resultados para este caso van desde casos de estudio para los cuales los mutantes adicionales representan un pequeño porcentaje sobre los generados por operadores tradicionales, a casos en donde *prvo* genera un incremento del doble en la cantidad original de mutantes.

9 Trabajo futuro

El trabajo presentado en esta tesis abre algunas líneas de trabajo futuro. Por un lado, planeamos analizar más profundamente cuales de las clases de fallas no acopladas a operadores actuales, identificadas en la sección 4.4, están acopladas a *prvo*. Creemos que algunas están directamente cubiertas por nuestro operador. Actualmente un estudio sobre este posible acoplamiento está siendo realizado como parte del trabajo final de Licenciado en Ciencias de la Computación de un alumno de la Universidad Nacional de Río Cuarto bajo la dirección de Pablo Ponzio y mía, éste está basado en el trabajo realizado por René Just [Just et al., 2014] e intenta encontrar si existe un acoplamiento de *prvo* a fallas aún no acopladas en ese trabajo, en particular a aquellas mencionadas en 4.4.

Nuestro operador *prvo* se presenta como un "meta-operador" altamente configurable, donde cada configuración se puede ver como un operador particular. La configuración del mismo se hace, en este trabajo, de una manera manual. Como automatizar esta configuración basándose en características del programa al cual está siendo aplicado, forma parte de un camino de investigación a explorar.

En esta tesis se utilizó *Dynamic Mutant Subsumption* como una de los análisis principales para evaluar el desempeño de *prvo* en el contexto de mutation testing. Dentro de éste, las nociones de mutante dominador y mutantes dominadores puros fueron las de mayor importancia para determinar si *prvo* podía considerarse dentro de los operadores suficientes o si por el contrario generaba mutantes redundantes. Afortunadamente los resultados indican que esto no sucede y los mutantes de *prvo*, al menos dentro de los casos de estudio utilizados, genera mutantes que tienden a ser dominadores. Un futuro análisis sobre cuales son los operadores cuyos mutantes *prvo* tiende a dominar, podría permitir decidir a cuales de éstos es posible reemplazar con *prvo*. A su vez, creemos que, tal como se expuso en 5.3.2, Dynamic Mutant Subsumption representa una métrica más confiable que *Mutation Score* y *Toughness* como forma de evaluar operadores de mutación y/o técnicas de selección de mutantes. Ésta es solo una intuición que requeriría mayores experimentos y análisis, pero que sin embargo creemos que podría beneficiar a estudios futuros.

9.1 PRVO EN REPARACIÓN AUTOMÁTICA DE PROGRAMAS

Como mencionamos en el capítulo 7, la reparación automática de programas utilizando mutación, es un área de investigación activa y en donde se debe balancear la capacidad de reparación, es decir, el conjunto de defectos que pueden ser reparados y la cantidad de sentencias que pueden estar involucradas en la reparación, con los recursos necesario para encontrar a la misma. Así como nuestra motivación para proponer a *prvo* como un operador a ser utilizado en *mutation testing*, se basa principalmente en cuan extenso es el uso de expresiones de navegación en programas orientados a objetos, la misma puede aplicarse para motivar a *prvo* como un operador a ser utilizado en la reparación de programas orientas a objetos. En la sección 7.1.4 se presentan algunos resultados que es capaz de conseguir *Stryker* en cuanto a reparación de múltiples defectos junto con el uso de una gran cantidad de operadores de mutación. La gran flexibilidad provista por *prvo*, principalmente en cuanto a acotar las expresiones a utilizar, podría ser explotada para diminuir considerablemente el espacio de candidatos a reparación con respecto a los generados por este operador. La relación entre restricciones a la configuración de *prvo* con respecto a las expresiones a utilizar, los potenciales beneficios (menor espacio de búsqueda), y desventajas (la perdida de la capacidad de reparar ciertos defectos), representa una investigación muy interesante en el área de reparación automática de programas.

Bibliografía

[DBL, 2010] (2010). *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society. 134, 138

[Ammann et al., 2014] Ammann, P., Delamaro, M. E., and Offutt, J. (2014). Establishing theoretical minimal sets of mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 21–30. IEEE Computer Society. 33, 63, 69

[Ammann and Offutt, 2016] Ammann, P. and Offutt, J. (2016). *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2nd edition. 14

[Andrews et al., 2005a] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005a). Is mutation an appropriate tool for testing experiments? In [Roman et al., 2005], pages 402–411. 3

[Andrews et al., 2005b] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005b). Is mutation an appropriate tool for testing experiments? In [Roman et al., 2005], pages 402–411. 5

[Arcuri and Yao, 2008] Arcuri, A. and Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China*, pages 162–168. IEEE. 5, 113, 114

[Boyapati et al., 2002] Boyapati, C., Khurshid, S., and Marinov, D. (2002). Korat: automated testing based on java predicates. In Frankl, P. G., editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM. 17

[Daran and Thévenod-Fosse, 1996] Daran, M. and Thévenod-Fosse, P. (1996). Software error analysis: A real case study involving real faults and mutations. In Zeil, S. J. and Tracz, W., editors, *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996*, pages 158–171. ACM. 5

[Debroy and Wong, 2010] Debroy, V. and Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. In [DBL, 2010], pages 65–74. 118

[DeMillo et al., 1978] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41. 3

[Dijkstra, 1972] Dijkstra, E. W. (1972). Structured programming. chapter Chapter I: Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK. 14

[(FAA), 2015] (FAA), F. A. A. (2015). Airworthiness directives; the boeing company airplanes. https://www.federalregister.gov/documents/2015/05/01/2015-10066/airworthiness-directives-the-boeing-company-airplanes. Document citation: 80 FR 24789; CFR: 14 CFR 39; Document Number: 2015-10066; RIN: 2120-AA64. 1

[Fraser and Arcuri, 2011] Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In [Gyimóthy and Zeller, 2011], pages 416–419. 74

[Galeotti et al., 2013] Galeotti, J. P., Rosner, N., Pombo, C. G. L., and Frias, M. F. (2013). TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307. 120

[Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of software engineering*. Prentice Hall. 14

[Gopinath et al., 2011] Gopinath, D., Malik, M. Z., and Khurshid, S. (2011). Specification-based program repair using SAT. In Abdulla, P. A. and Leino, K. R. M., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 173–188. Springer. 117

[Gopinath et al., 2016] Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., and Groce, A. (2016). Measuring effectiveness of mutant sets. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, pages 132–141. IEEE Computer Society. 69

[Grün et al., 2009] Grün, B. J. M., Schuler, D., and Zeller, A. (2009). The impact of equivalent mutants. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pages 192–199. IEEE Computer Society. 4, 32

[Gutiérrez Brida and Scilingo, 2017] Gutiérrez Brida, S. and Scilingo, G. (2017). Boolean expression extender - A mutation operator for strengthening and weakening boolean expressions. In Monteverde, H. and Santos, R., editors, *2017 XLIII Latin American Computer Conference, CLEI 2017, Córdoba, Argentina, September 4-8, 2017*, pages 1–10. IEEE. 51, 67, 68, 70, 71

[Gyimóthy and Zeller, 2011] Gyimóthy, T. and Zeller, A., editors (2011). *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM. 134, 135

[Harman, 2010] Harman, M. (2010). Automated patching techniques: the fix is in: technical perspective. *Commun. ACM*, 53(5):108. 4

[Harman et al., 2011] Harman, M., Jia, Y., and Langdon, W. B. (2011). Strong higher order mutation-based test data generation. In [Gyimóthy and Zeller, 2011], pages 212–222. 4

[Hierons et al., 1999] Hierons, R. M., Harman, M., and Danicic, S. (1999). Using program slicing to assist in the detection of equivalent mutants. *Softw. Test., Verif. Reliab.*, 9(4):233–262. 32

[Jackson, 2006] Jackson, D. (2006). *Software Abstractions - Logic, Language, and Analysis*. MIT Press. 11

[Jalote, 1997] Jalote, P. (1997). *An Integrated Approach to Software Engineering*. Springer-Verlag, Berlin, Heidelberg, 2nd edition. 14

[Jalote, 2005] Jalote, P. (2005). *An Integrated Approach to Software Engineering*. Texts in Computer Science. Springer. 14

[Jia and Harman, 2008] Jia, Y. and Harman, M. (2008). Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*, pages 249–258. IEEE Computer Society. 4

[Jia and Harman, 2009a] Jia, Y. and Harman, M. (2009a). Higher order mutation testing. *Information & Software Technology*, 51(10):1379–1393. 4

[Jia and Harman, 2009b] Jia, Y. and Harman, M. (2009b). Higher order mutation testing. *Information & Software Technology*, 51(10):1379–1393. 80

[Just et al., 2013] Just, R., Ernst, M. D., and Fraser, G. (2013). Using state infection conditions to detect equivalent mutants and speed up mutation analysis. *CoRR*, abs/1303.2784. 4, 32

[Just et al., 2014] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In Cheung, S., Orso, A., and Storey, M. D., editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM. 3, 4, 5, 34, 41, 130

[Just et al., 2017] Just, R., Kurtz, B., and Ammann, P. (2017). Inferring mutant utility from program context. In Bultan, T. and Sen, K., editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 284–294. ACM. 33

[Kim et al., 2013] Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In Notkin, D., Cheng, B. H. C., and Pohl, K., editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 802–811. IEEE Computer Society. 5, 116

[Kim et al., 2000] Kim, S., Clark, J. A., and McDermid, J. A. (2000). Class mutation : Mutation testing for object-oriented programs. 99

[King and Offutt, 1991] King, K. N. and Offutt, A. J. (1991). A fortran language system for mutation-based software testing. *Softw., Pract. Exper.*, 21(7):685–718. 8

[Le Goues et al., 2015] Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P. T., Forrest, S., and Weimer, W. (2015). The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256. 115

[Le Goues et al., 2012] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72. 5, 114

[Leavens et al., 2002] Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. (2002). How the design of JML accomodates both runtime assertion checking and formal verification. In de Boer, F. S., Bonsangue, M. M., Graf,

S., and de Roever, W. P., editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 262–284. Springer. 120

[Leveson and Turner, 1993] Leveson, N. G. and Turner, C. S. (1993). Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41. 1

[Long and Rinard, 2015] Long, F. and Rinard, M. (2015). Staged program repair with condition synthesis. In Nitto, E. D., Harman, M., and Heymans, P., editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM. 116

[Ma et al., 2002] Ma, Y., Kwon, Y. R., and Offutt, J. (2002). Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA*, pages 352–366. IEEE Computer Society. 35

[Ma et al., 2005] Ma, Y., Offutt, J., and Kwon, Y. R. (2005). Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133. 11, 49

[Ma and and, 2018] Ma, Y.-S. and and, J. O. (2016 (accessed November 10, 2018)). *Description of muJavaâĂŹs Method-level Mutation Operators*. 92

[Ma and Offutt, 2018] Ma, Y.-S. and Offutt, J. (2014 (accessed November 10, 2018)). *Description of Class Mutation Operators for Java*. 35

[Namin et al., 2008] Namin, A. S., Andrews, J. H., and Murdoch, D. J. (2008). Sufficient mutation operators for measuring test effectiveness. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 351–360. ACM. 4, 30, 92

[Namin and Kakarla, 2011] Namin, A. S. and Kakarla, S. (2011). The use of mutation in testing experiments and its sensitivity to external threats. In Dwyer, M. B. and Tip, F., editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 342–352. ACM. 5

[Offutt, 1989] Offutt, A. J. (1989). The coupling effect: Fact or fiction. In Kemmerer, R. A., editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989*, pages 131–140. ACM. 3

[Offutt, 1992] Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20. 3

[Offutt and King, 1987] Offutt, A. J. and King, K. N. (1987). A fortran 77 interpreter for mutation analysis. In Wexelblat, R. L., editor, *Proceedings of the Symposium on Interpreters and Interpretive Techniques, 1987, St. Paul, Minnesota, USA, June 24 - 26, 1987*, pages 177–188. ACM. 8

[Offutt et al., 1996] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118. 4, 29, 33, 92

[Pacheco and Ernst, 2007] Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In Gabriel, R. P., Bacon, D. F., Lopes, C. V., and Jr., G. L. S., editors, *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 815–816. ACM. 74

[Papadakis et al., 2016] Papadakis, M., Henard, C., Harman, M., Jia, Y., and Traon, Y. L. (2016). Threats to the validity of mutation-based test assessment. In Zeller, A. and Roychoudhury, A., editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365. ACM. 63

[Pressman, 2001] Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition. 14

[Roman et al., 2005] Roman, G., Griswold, W. G., and Nuseibeh, B., editors (2005). *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM. 133

[Schuler and Zeller, 2010] Schuler, D. and Zeller, A. (2010). (un-)covering equivalent mutants. In [DBL, 2010], pages 45–54. 4, 32

[Staber et al., 2005] Staber, S., Jobstmann, B., and Bloem, R. (2005). Finding and fixing faults. In Borrione, D. and Paul, W. J., editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 35–49. Springer. 5, 114

[Tillmann and de Halleux, 2008] Tillmann, N. and de Halleux, J. (2008). Pex-white box test generation for .net. In Beckert, B. and Hähnle, R., editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy,*

April 9-11, 2008. Proceedings, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer. 115

[Visser, 2016] Visser, W. (2016). What makes killing a mutant hard. In Lo, D., Apel, S., and Khurshid, S., editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 39–44. ACM. 32, 68

[Wang et al., 2018a] Wang, K., Sullivan, A., and Khurshid, S. (2018a). Automated model repair for alloy. In Huchard, M., Kästner, C., and Fraser, G., editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 577–588. ACM. 11, 118

[Wang et al., 2018b] Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., and Khurshid, S. (2018b). Systematic generation of non-equivalent expressions for relational algebra. In Butler, M. J., Raschke, A., Hoang, T. S., and Reichl, K., editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings*, volume 10817 of *Lecture Notes in Computer Science*, pages 105–120. Springer. 11

[Yao et al., 2014] Yao, X., Harman, M., and Jia, Y. (2014). A study of equivalent and stubborn mutation operators using human analysis of equivalence. In Jalote, P., Briand, L. C., and van der Hoek, A., editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 919–930. ACM. 32, 68

[Zemín et al., 2017] Zemín, L., Gutiérrez Brida, S., Godio, A., Cornejo, C., Degiovanni, R., Regis, G., Aguirre, N., and Frias, M. F. (2017). An analysis of the suitability of test-based patch acceptance criteria. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 14–20. IEEE. 115