MARCO DE PRUEBAS DE MUTACIÓN DE DISPARADORES EN POSTGRESQL

A MUTATION TESTING FRAMEWORK FOR TRIGGERS IN POSTGRESQL

Sergio Cristian Milani Sáez

MÁSTER EN INGENIERIA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

Convocatoria: Septiembre 2021
Calificación: 6.5

25 de septiembre de 2021

Director/es:

Mercedes García Merayo
Jesús Correas Fernández

Resumen en castellano

El uso de las pruebas de mutación ha acaparado mucha atención durante las últimas décadas como técnica para determinar la calidad de los conjuntos de pruebas utilizados durante el proceso de validación de sistemas. Las pruebas de mutación se basan en la incorporación de pequeños cambios sintácticos en el código para simular fallos que programadores experimentados podrían realizar durante la fase de desarrollo o de mantenimiento del sistema. En este trabajo se ha desarrollado un marco de pruebas de mutación para disparadores dentro del ámbito de las bases de datos. En este contexto, los disparadores son fragmentos de código que se ejecutan automáticamente cuando se producen determinadas acciones sobre las tablas a las que se encuentran asociados. Por una parte se ha definido un conjunto de operadores de mutación sobre cláusulas específicas de estos objetos. Por otra parte se ha automatizado la aplicación de la técnica de pruebas de mutación a disparadores diseñados en PostgreSQL. La herramienta desarrollada permite evaluar la calidad de diferentes conjuntos de pruebas para detectar los errores inducidos por los operadores de mutación así como compararlos.

Palabras clave

pruebas de mutación, disparador, testing, Bases de Datos

Abstract

In the last decades the use of mutation tests has been playing a very important role in determining the quality of the test cases used during the system validation process. Mutation testing consists of inserting small syntactic changes into the code to simulate some bugs that experienced programmers might make during the development or maintenance phase of a system. In this work, a set of mutation tests for triggers has been done within the scope of databases. In this scenario, triggers are snippets of code which are automatically executed when certain actions occur on the tables to which they are asociated. On the one hand, a group of mutation operators has been defined on specific clauses of these objects. On the other hand, the application of the mutation testing technique to triggers has been automated to be used in PostgreSQL. The tool created allows us to evaluate the quality of different test cases to detect the errors caused by the mutation operators as well as to compare them.

Keywords

mutation, trigger, application, test

Índice general

Índice I

Agradecimientos III

Dedicatoria IV

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 2
- 1.3. Plan de trabajo . . . 3
- 1.4. Estructura del Documento . . . 3

2. Preliminares 5
- 2.1. Pruebas de mutación . . . 5
- 2.2. Antecedentes: Aplicación de pruebas de mutación en bases de datos . . . 7
- 2.3. Disparadores en PostgreSQL . . . 10

3. Propuesta de pruebas de mutaciones para disparadores 13
- 3.1. Definición de operadores de Mutación . . . 13
  - 3.1.1. Operadores de cabecera del disparador . . . 14
  - 3.1.2. Operadores de función del disparador . . . 16
- 3.2. Generación de mutantes . . . 16
- 3.3. Ejecución del proceso de pruebas . . . 19
  - 3.3.1. Comparación de los resultados de la ejecución de disparadores . . . 19
  - 3.3.2. Mecanismo de ejecución de pruebas independientes . . . 21

4. Implementación de la propuesta 22
- 4.1. Estructura del sistema . . . 22
- 4.2. Herramientas y sistemas utilizados . . . 23
  - 4.2.1. PostgreSQL . . . 23
  - 4.2.2. Python . . . 24
  - 4.2.3. HTML . . . 24
  - 4.2.4. PHP . . . 25
  - 4.2.5. Javascript . . . 25
  - 4.2.6. Cascading Style Sheets . . . 25
  - 4.2.7. pgAdmin4 . . . 25
  - 4.2.8. Entornos de desarrollo . . . 26
- 4.3. Modelo de la base de datos interna . . . 26

4.4. Funcionalidades de la Aplicación 28
4.4.1. Acceso Usuarios 28
4.4.2. Menú principal 29
4.4.3. Gestión de disparadores 29
4.4.4. Operadores de mutación 31
4.4.5. Gestión conjuntos de pruebas 32
4.4.6. Proceso de mutación 35
4.4.7. Detalles de implementación del proceso de ejecución de pruebas de mutación 37

5. Experimentos y pruebas realizadas **41**
5.1. Experimento 1 42
5.1.1. Experimento 1B 45
5.2. Experimento 2 46
5.2.1. Experimento 2B 51
5.3. Experimento 3 51

6. Conclusiones y trabajo futuro **57**
6.1. Conclusiones 57
6.2. Trabajo futuro 59

7. Introduction **61**
7.1. Motivation 61
7.2. Objectives 62
7.3. Workplan 63
7.4. Structure of the Document 63

8. Conclusions and future work **65**
8.1. Conclusions 65
8.2. Future work 66

Agradecimientos

En primer lugar me gustaría agradecer a D. Jesús Correas Fernández y D.ª María de las Mercedes García Merayo, directores de este Trabajo Fin de Máster su ayuda y dedicación a lo largo de todo este proyecto. Gracias por la oportunidad que me habéis brindado para poder llevar a cabo este trabajo. Por todo lo que me habéis enseñado y aportado durante este camino.

Quiero agradecer a mis padres y mis hermanos que han sido los que me han estado ayudando cuando las cosas no salían como esperaba y han sufrido como los que más para ayudarme a sacar adelante mi etapa académica.

Gracias a todas las personas que me han ayudado a formarme, tanto a profesores como a mis compañeros de clase, que de una manera u otra han sido partícipes de mi formación. Seguro que me olvido de gente, que durante todo este camino, han sido muy importantes. A todos vosotros gracias por ser parte fundamental en mi crecimiento como estudiante, y sobre todo como persona.

Dedicatoria

A mis padres y a mis hermanos, pero especialmente a mi tía Reyes, que ya no está con nosotros.

Capítulo 1

Introducción

En este capítulo se introducen las motivaciones fundamentales para la realización de este trabajo, así como los objetivos del mismo. También se describe el plan de trabajo que se ha llevado a cabo durante el desarrollo de este Trabajo de Fin de Máster y la estructura de esta memoria.

1.1. Motivación

Las pruebas de mutación (*mutation testing*[14,19]) es una técnica de pruebas de caja blanca bien establecida para el diseño y evaluación de la calidad de conjuntos de pruebas de sistemas software. El objetivo principal de esta técnica es determinar la capacidad de un conjunto de pruebas para la detección de pequeños errores que podría introducir un programador *competente* durante la construcción o mantenimiento de un sistema software. La idea básica para evaluar los conjuntos de pruebas mediante pruebas de mutación consiste en introducir pequeños cambios sintácticos en el sistema original generando un gran número de versiones *erróneas* denominadas mutantes. Los mutantes se generan automáticamente aplicando operadores de mutación que definen de forma precisa los cambios introducidos en el sistema. Ejemplos típicos de operadores de mutación son la sustitución de un operador aritmético por otro o la eliminación de un elemento sintáctico de una sentencia del programa que se está evaluando. Una vez se dispone de un conjunto de mutantes, se aplica el conjunto de pruebas a evaluar tanto al sistema original como a cada uno de los mutantes. Si el con-

junto de pruebas del sistema está bien diseñado, debería detectar los errores introducidos en estos mutantes, produciendo salidas diferentes a las del sistema original. En este caso se dice que el mutante ha sido *matado*. Si la aplicación de cada caso de prueba al mutante genera exactamente la misma salida que el sistema original, entonces se puede concluir que el conjunto de pruebas no es capaz de detectar el error introducido en el mutante, y en este caso se dice que el mutante esta *vivo*. La capacidad de detección del conjunto de pruebas viene dada mediante una métrica que corresponde al porcentaje de mutantes que han sido matados. Si el valor es muy bajo, el conjunto de pruebas debería ser extendido para poder detectar los errores reflejados en los mutantes vivos. La hipótesis que plantea esta técnica es que, si un conjunto de pruebas es capaz de distinguir un sistema de sus mutantes, será adecuado para detectar fallos introducidos inadvertidamente en el sistema.

1.2. Objetivos

La técnica de pruebas de mutación se ha estado utilizado en las últimas décadas en muy diversos ámbitos del desarrollo de software. En particular, existen múltiples aplicaciones de esta técnica en sistemas de bases de datos, fundamentalmente orientadas a la detección de errores en sentencias SQL de consulta de datos. Además, algunos autores han aplicado esta técnica sobre sentencias SQL de modificación de datos. Este tipo de aplicaciones son especialmente sofisticadas, pues la determinación de si un mutante ha sido matado requiere la comparación del estado de la base de datos. Sin embargo, en la literatura existente en este área no se ha encontrado ninguna aplicación de esta técnica a los disparadores de un sistema de bases de datos. El objetivo fundamental de este trabajo consiste en estudiar la aplicación de esta técnica a los disparadores de base de datos, proponiendo un marco de pruebas de mutación para disparadores integrados en bases de datos y desarrollando una herramienta para poder evaluar su funcionamiento.

Los objetivos específicos de este trabajo son los siguientes:

- Estudiar la literatura relacionada con el uso de las pruebas de mutación en el ámbito

de las bases de datos.

- Definir un conjunto de operadores de mutación para un lenguaje de diseño de disparadores.
- Desarrollo de una herramienta que implemente la aplicación de los operadores de mutación para la generación de mutantes.
- Evaluar los resultados obtenidos con el marco desarrollado.

1.3. Plan de trabajo

Las tareas necesarias para desarrollar este trabajo se han dividido en las siguientes fases:

- Fase 1: Análisis y revisión de trabajos previos.
- Fase 2: Selección del sistema gestor de bases de datos que se considerara para el desarrollo del trabajo.
- Fase 3: Diseño de operadores de mutación específicos.
- Fase 4: Diseño e implementación de los módulos que componen la herramienta que automatiza el marco de pruebas de mutaciones.
- Fase 5: Evaluación del marco propuesto.

La supervisión y seguimiento de este trabajo se realiza mediante reuniones semanales que permitan revisar los avances realizados, identificar problemas durante el desarrollo, determinar el alcance de los objetivos y planificar pautas para continuar con el desarrollo.

1.4. Estructura del Documento

Esta memoria trabajo se encuentra estructurada en seis capítulos.

- Capítulo 1 - Introducción: Expone la motivación del trabajo realizado junto a los objetivos planteados.
- Capítulo 2 - Preliminares: Expone los conceptos fundamentales en los que se basa el marco propuesto y explora la literatura relacionada. Incluye los elementos básicos utilizados para la comprensión del trabajo realizado.
- Capítulo 3 - Propuesta de pruebas de mutaciones para disparadores. En este capítulo se presentan los operadores de mutación que se proponen para su aplicación en el marco de pruebas de mutaciones propuesto y se describen los procesos de generación de mutantes y de ejecución de pruebas.
- Capítulo 4 - Implementación de la propuesta: Describe el proceso de desarrollo de la herramienta, así como las distintas funcionalidades que integra.
- Capítulo 5 - Evaluación: Se presentan diversos experimentos realizados para la evaluación de la propuesta y análisis de los resultados obtenidos.
- Capítulo 6 - Conclusiones y Trabajo Futuro: Expone las conclusiones que se pueden extraer del trabajo realizado. Adicionalmente, se indican posibles modificaciones sobre la herramienta que podrían mejorar su funcionamiento.

Capítulo 2

Preliminares

En este capítulo se presentan los fundamentos de pruebas de mutaciones utilizados en el desarrollo de este trabajo, así como los conceptos básicos relativos al diseño de *disparadores* en el sistema de gestión de bases de datos relacionales PostgreSQL.

2.1. Pruebas de mutación

Las pruebas de mutación es una técnica de pruebas de software basada en fallos[14]. El objetivo principal de las pruebas de mutación es evaluar la idoneidad de un conjunto de pruebas para detectar errores. La idea es que pequeños cambios sintácticos se introducen en un programa para producir un conjunto de versiones llamadas *mutantes*. Los cambios insertados en el programa original simulan errores comunes y se definen mediante *operadores de mutación*. Podemos distinguir entre mutantes de primer orden, que se generan aplicando un operador de mutación solo una vez, y mutantes de orden superior que se generan mediante la aplicación de más de un operador de mutación. Las pruebas de mutación se basan en dos principios fundamentales. Por una parte, la *hipótesis del programador competente*, que establece que los programadores competentes cometen pequeños errores, lo que justifica el uso de operadores de mutación que inyectan estos errores en los programas[14]. Un ejemplo puede ser una mutación del operador aritmético de suma, consistente en la sustitución de una aparición del símbolo $+$ por $*$. Por otra parte, la *hipótesis del efecto de acoplamiento*[22] establece que "los mutantes complejos están acoplados a mutantes simples de tal manera

que un conjunto de casos de prueba que detecta todos los mutantes simples en un programa detectará un gran porcentaje de los mutantes complejos". Los estudios empíricos evidenciaron este efecto[22,28] lo que justificó que las pruebas de mutación se centren en mutantes de primer orden.

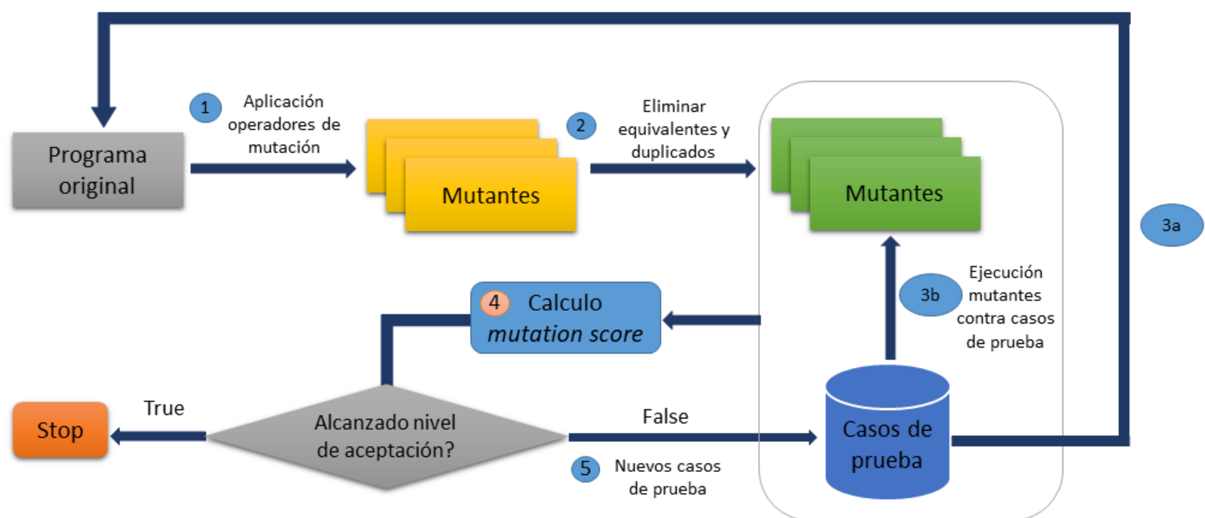


Figura 2.1: *Proceso de pruebas de mutación.*

La figura 2.1 muestra las diferentes fases de ejecución de la técnica de pruebas de mutación. Una vez que se ha generado un conjunto de mutantes de un programa mediante la aplicación de los operadores de mutación (1) se deben eliminar aquellos mutantes *problemáticos*. Estos mutantes son de dos tipos: algunos mutantes son semánticamente *equivalentes* al programa original a pesar de ser sintácticamente diferentes; por otra parte, los mutantes *duplicados* son los mutantes que son equivalentes a otros mutantes pero no al programa original. Los mutantes problemáticos deben eliminarse (2) antes de ejecutar tanto el programa original (3a) como los mutantes (3b) contra el conjunto de casos de prueba. Estos mutantes no deben tenerse en cuenta a la hora de determinar la idoneidad del conjunto de pruebas. Si el comportamiento de un mutante frente a algún caso de prueba es diferente del comportamiento del programa original, se dice que el mutante ha sido *matado* (*killed*). Si ninguno de los casos de prueba puede matar a un mutante, entonces se dice que el mutante está *vivo*.

La idoneidad del conjunto de casos de prueba corresponde a la proporción de mutantes que son matados por el conjunto de pruebas (4). Si esta medida no alcanza un nivel aceptable, se podrán diseñar nuevos casos de prueba para tratar de matar a los mutantes vivos (5).

Para comprender mejor la propuesta que se presenta en este trabajo, un marco de pruebas de mutación para disparadores, vamos a realizar una breve introducción de los principales conceptos de esta técnica en dicho ámbito. El concepto más importante es el de mutante, que en este trabajo corresponde a un disparador que ha sido modificado con un cambio sintáctico como resultado de la aplicación de un operador de mutación. El segundo de los conceptos a destacar es el de caso de prueba. En el contexto de los disparadores, un caso de prueba es una sentencia `INSERT`, `UPDATE` o `DELETE`, que son las que pueden provocar la ejecución de un disparador.

2.2. Antecedentes: Aplicación de pruebas de mutación en bases de datos

Existen muchos trabajos en la literatura que proponen el uso de técnicas de análisis de mutaciones en el ámbito de las bases de datos. La mayoría de ellos se centran en consultas SQL (Strutured Query Language)[10]. Uno de los trabajos más relevantes[28] propone una colección de operadores de mutación para las diferentes cláusulas disponibles en la sentencia `SELECT`, con el objetivo de determinar la adecuación de los casos de prueba para detectar el tipo de errores simulados por estos operadores. Cabe mencionar que los casos de prueba en este caso se corresponden con el estado de la base de datos. La herramienta SQLMutation[27] automatiza el proceso de generación de mutantes de esta propuesta. La validez y el rendimiento de este marco ha sido evaluado en un sistema de base de datos industrial[15]. Esta propuesta ha sido utilizada en otros enfoques. Por ejemplo, en el uso de un algoritmo genético que tiene como objetivo seleccionar una instancia efectiva de una base de datos para detectar fallos en las sentencias `SELECT`[21]. La consulta SQL y los mutantes correspondientes se ejecutan contra individuos (subconjuntos de la base de datos inicial) y la idoneidad de

los mismos se utiliza para determinar el *fitness* de cada individuo.

Otras propuestas abordan el problema de la generación de casos de prueba para chequear consultas SQL y aplicaciones de bases de datos. Un subconjunto de los operadores de mutación propuestos en[21], llamado el conjunto suficiente de operadores de mutación SQL, se ha utilizado para aplicar pruebas de mutación que determinen la calidad de los casos de prueba generados para aplicaciones de bases de datos[24]. La propuesta tiene como objetivo obtener un conjunto de pruebas de alta calidad. En esta solución se utiliza un resolutor de restricciones para identificar automáticamente nuevos casos de prueba cuando la idoneidad del conjunto de pruebas no alcanza un umbral aceptable. X-data[17] genera conjuntos de pruebas que matan mutantes de consultas SQL correspondientes tanto a mutaciones de cláusulas `JOIN` como a sustitución de operadores relacionales en la cláusula `WHERE`. La técnica propuesta genera un conjunto de restricciones a partir de los mutantes para ser procesadas por un resolutor de restricciones que producirá casos de prueba. Este enfoque se ha ampliado[13] para tratar con mutaciones de las operaciones de agregación, operaciones de conjuntos, condiciones de unión, cláusulas `GROUP BY` y `DISTINCT`, así como mutaciones de *strings* y valores `NULL`. JDAMA[31,33], un analizador de mutaciones de aplicaciones de bases de datos Java, aplica el enfoque introducido en[27] para tratar con programas Java que interactúan con una base de datos. MutaGen[23] considera dos aspectos que afectan a la generación de instancias de bases de datos para probar aplicaciones que interactúan con las mismas: el código del programa y las consultas SQL. El enfoque se basa en el diseño de restricciones para matar mutantes tanto del código de programa como de las consultas SQL derivadas de las mutantes correspondientes. Un resolutor de restricciones genera los casos de prueba necesarios para eliminar ambos tipos de mutantes.

Todos los trabajos mencionados anteriormente se centran en la aplicación de pruebas de mutación a la sentencia `SELECT`. No obstante, existen algunos trabajos que proponen la aplicación de esta técnica en sentencias SQL que modifican el estado de la base de datos (`INSERT`, `DELETE` y `UPDATE`). Uno de los enfoques define una colección de operadores de

mutación para este tipo de sentencias e introduce un algoritmo que permite verificar si los mutantes generados son eliminados sin ejecutarlos realmente[32].

Otras propuestas consideran la aplicación de las pruebas de mutaciones a la estructura de la base de datos. Uno de estos trabajos[30] propone el uso de *mutant schemata*[29] que codifica todos los mutantes en un *meta-mutante*, y la paralelización para el proceso de ejecución. Este enfoque reduce el tiempo requerido para ejecutar el proceso de las pruebas de mutación. Otro enfoque[20] introduce una técnica de análisis de mutaciones para evaluar la calidad de los conjuntos de pruebas generados por una técnica basada en búsquedas. Se define una colección de operadores de mutación para las diferentes restricciones de integridad (clave primaria, claves únicas, restricciones de chequeo, claves externas y restricciones `NOT NULL`). En[12] se define un conjunto de operadores de mutación basados en diferentes tipos de restricciones del modelo entidad-relación extendido, como las restricciones de participación o cardinalidad. Estos operadores de mutación se utilizan para crear mutantes de sentencias SQL integradas en aplicaciones de bases de datos.

Las pruebas de mutación también se han utilizado en el ámbito de la inyección de vulnerabilidades en SQL para abordar la generación de casos de pruebas que ayudan a detectarlas. MUSIC[26] genera mutantes obtenidos de la aplicación de una colección de operadores de mutación a *Java Pages Servers* que solo se eliminan mediante pruebas correspondientes a los ataques de inyección SQL. μ4SQLi[11] tiene como objetivo crear nuevos casos de pruebas que desencadenen ataques de inyección SQL. La técnica se basa en un conjunto de operadores de mutación que manipulan parámetros de entrada en los que los atacantes inyectan fragmentos de código SQL malicioso. Además del uso de pruebas de mutación en sentencias SQL, se han propuesto algunos trabajos para lenguajes No-SQL, como *Cassandra Query Language*[25]. En este caso, se propone un conjunto de operadores de mutación y diferentes criterios de cobertura. Recientemente, se ha definido un conjunto de operadores de mutación específicos para *Google Query Language*[18].

Como se puede comprobar, existe una extensa bibliografía relacionada con la aplicación

de la técnica de pruebas de mutación a bases de datos, pero ninguno de estos trabajos aborda el problema de aplicar esta técnica a los disparadores. Los disparadores son pequeños programas procedurales que se ejecutan automáticamente cuando se producen determinados eventos en el funcionamiento de una base de datos, como puede ser la inserción, modificación o borrado de filas en una tabla, o bien eventos del sistema, como la conexión de un usuario o la ejecución de una sentencia DDL. Aunque los disparadores son programas procedurales, tienen características específicas que los distinguen claramente de los lenguajes de programación sobre los que habitualmente se ha aplicado la técnica de pruebas de mutación. Por tanto, en esta aplicación particular se debe combinar la técnica de pruebas de mutación aplicada a programas procedurales con los aspectos específicos del lenguaje de programación de disparadores.

2.3. Disparadores en PostgreSQL

En este trabajo, se ha seleccionado el sistema de gestión de bases de datos relacionales (RDBMS) PostgreSQL[7] para implementar el marco de prueba de mutaciones propuesto. Las razones principales son que PostgresSQL es de código abierto y después de un largo período de desarrollo se considera confiable y robusto.

Independientemente del RDBMS que consideremos, un disparador es un procedimiento que la base de datos ejecuta automáticamente cada vez que se realiza un determinado tipo de operación. Los disparadores generalmente están asociados a tablas, pero también pueden estar relacionados con vistas. Los disparadores se pueden configurar para ejecutarse antes o después de las operaciones `INSERT`, `UPDATE` o `DELETE`. También podemos indicar si el disparador se ejecuta una vez por cada fila afectada por la instrucción SQL que activó el disparador o una vez por instrucción. Además, es posible establecer una condición `WHEN` para activar el disparador solo en el caso de que se incluyan determinadas condiciones. Los activadores asociados a las vistas se ejecutan en lugar de las operaciones `INSERT`, `UPDATE` o `DELETE`. El listado 2.1 presenta una versión simplificada de la sintaxis de la sentencia de

creación de disparadores en PostgreSQL. Hemos omitido los elementos que no se consideran en nuestro marco.

```
CREATE disparador name {BEFORE|AFTER|INSTEAD OF} {event[ OR...]}
    ON table
    [FROM referenced_table_name ]
    [FOR [EACH] {ROW|STATEMENT}]
    [WHEN (condition)]
    EXECUTE PROCEDURE function_name()
where event can be
    INSERT
    UPDATE [OF column_name[, ... ]]
    DELETE
```

Listado 2.1: *Instrucción para creación de disparadores en PostgreSQL.*

Para crear un disparador, PostgreSQL requiere una función de disparador asociada. La sintaxis básica para crear una función de disparador se proporciona en el listado 2.2.

```
CREATE [OR REPLACE] FUNCTION name() RETURNS trigger {...}
```

Listado 2.2: *Creación función de disparador en PostgreSQL.*

La función del disparador debe definirse como una función que no toma argumentos y devuelve un disparador. La función del disparador se puede escribir utilizando cualquier lenguaje compatible con PostgreSQL (TCL, Python, PL/pgSQL y Perl). En nuestro marco utilizamos PL/pgSQL, que está diseñado específicamente para PostgreSQL y admite el uso de todos los tipos de datos, operadores y funciones del RDBMS de PostgreSQL. Diferentes variables están disponibles en las funciones de disparador, que permiten acceder a distintos elementos de la definición del mismo (`TG_NAME`, `TG_WHEN`, `TG_LEVEL`, etc.) así como a las filas afectadas por las operaciones que lanzan la ejecución del disparador (`NEW` y `OLD`). Específicamente, las variables `NEW` y `OLD` se corresponden, en disparadores a nivel de fila, a la nueva fila para operaciones `INSERT` y `UPDATE` y la fila antigua para operaciones `UPDATE` y `DELETE`, respectivamente. El listado 2.3 muestra un ejemplo de la creación de un disparador denominado, `mytrigger`. En este ejemplo se puede observar que la configuración del disparador

indica que este se ejecutará antes de realizar cualquier operación INSERT o UPDATE en la tabla myrecords. Sin embargo, el disparador solo se ejecutará si la condición de la cláusula WHEN es satisfecha por los registros involucrados en la operación que lo dispara. La cláusula FOR EACH ROW establece que la función del disparador se invocará una vez para cada fila afectada por la instrucción que activó el mismo.

```
CREATE TRIGGER mytrigger
BEFORE INSERT OR UPDATE
ON myrecords
WHEN (NEW.mcol<NEW.pcol*10 AND NOT NEW.bcol)
FOR EACH ROW
EXECUTE PROCEDURE change_myrecords();
```

Listado 2.3: *Ejemplo de creación de un disparador.*

En los siguientes capítulos utilizaremos las expresiones “cabecera del disparador” para referirnos a la sentencia de creación del disparador, y “función del disparador” para indicar la función invocada por la cabecera del disparador.

Capítulo 3

Propuesta de pruebas de mutaciones para disparadores

En este capítulo se introduce la propuesta para aplicar la técnica de pruebas de mutación a disparadores diseñados en el RDBMS PostgreSQL. Este trabajo se centra en un sistema de gestión de bases de datos en particular para poder desarrollar una herramienta que implemente el marco y permita su evaluación experimental. Aunque ninguno de los lenguajes de programación de disparadores existentes sigue completamente el estándar ISO SQL/PSM, el marco propuesto puede ser generalizable a otros sistemas de bases de datos pues los componentes básicos de estos lenguajes son similares en todos los sistemas disponibles en la industria.

3.1. Definición de operadores de Mutación

En la técnica de pruebas de mutación, la generación de mutantes se realiza a partir de una serie de operadores de mutación bien definidos que, o bien simulan los errores habituales producidos por un *programador competente* (por ejemplo, confundir un operador por otro, o olvidar negar una expresión Booleana), o producen casos de prueba relevantes, como puede ser el caso de división por cero en expresiones aritméticas.

En esta sección se proporciona una descripción detallada del conjunto de operadores de mutación que proponemos en el marco de pruebas de mutación de disparadores. Los

operadores se han clasificado en 4 categorías según la naturaleza de los errores que inyectan en los disparadores. Estos operadores cubren los principales elementos a considerar cuando se diseñan los disparadores.

A su vez todas los operadores pueden ser divididos en dos grandes grupos. Por una parte los operadores de mutación que afectan a la sección de la cabecera del disparador, y por otra, los operadores de mutación que se aplican en el cuerpo de la función invocada desde la cabecera.

3.1.1. Operadores de cabecera del disparador

Se distinguen varios grupos de operadores de mutación en función de la cláusula de la cabecera del disparador a la que afectan.

Operadores de mutación de la cláusula WHEN

Este grupo incluye una colección de operadores de mutación que afectan a elementos de la cláusula `WHEN`. La cláusula `WHEN` permite definir en un disparador una expresión lógica que determina las filas modificadas de la tabla que van a provocar la ejecución del disparador. En esta cláusula se pueden definir diversos operadores de mutación.

En primer lugar se define un operador que corresponde a la eliminación completa de la cláusula `WHEN` en la sentencia de creación de un disparador. También se define una colección de operadores que modifican los operadores aritméticos, relacionales y lógicos incluidos en la expresión lógica asociada a dicha cláusula. Por último, se considera un operador que reemplaza las ocurrencias de las variables `NEW` y `OLD` en este componente de la instrucción de creación. Los operadores considerados son los siguientes:

- DELWC: este operador elimina la cláusula `WHEN` de la sentencia de creación.
- RLREPH: el operador de mutación reemplaza cada aparición de un operador relacional (=, <>, <, <=,>,>=) en la cláusula `WHEN` por cada uno de los restantes operadores relacionales.

- ARREPH: como el operador anterior, la mutación consiste en reemplazar las apariciones de los operadores aritméticos $(+, -, *, /)$ por cada uno de los restantes.

- LGREPH: en este caso, la mutación se aplica a las ocurrencias de los operadores lógicos `AND` y `OR` en la cláusula `WHEN`. Los operadores `AND` y OR se reemplazan entre sí.

- NTDELH: cada aparición del operador lógico `NOT` es eliminada.

- VAREPH: cada aparición de las variables `NEW` y `OLD` en la cláusula `WHEN` se reemplazan entre sí. Este operador de mutación solo se puede aplicar en el caso de que la operación que lanza el disparador sea `UPDATE`. En otro caso, el mutante generado será incorrecto debido al hecho de que las variables `NEW` y `OLD` solo están disponibles simultáneamente para las operaciones `UPDATE`.

Operadores de mutación de la cláusula de evento

El objetivo de estos operadores es detectar errores lógicos en la cláusula que indica las acciones que activan el disparador. Se distinguen 3 operadores de mutación en esta categoría:

- ADDEC: este operador se aplica a los disparadores cuya cláusula de evento contiene una o dos acciones. La mutación consiste en extender dicha cláusula con cada acción no asociada al disparador originalmente.

- DELEC: este operador se aplica a los disparadores cuya cláusula de evento contiene dos o tres acciones. La mutación consiste en eliminar de la cláusula de evento una de las acciones.

- REPEC: en este caso, el operador se aplica a los disparadores que presentan solo una o dos acciones en la cláusula de evento. El operador de mutación reemplaza cada acción por las que no están incluidas en la cláusula.

Operador de mutación de la cláusula de nivel de fila

Este grupo incluye un único operador de mutación, REPRLV, que reemplaza la cláusula `FOR EACH STATEMENT` por `FOR EACH ROW` en la sentencia de creación del disparador.

Operadores de mutación de la cláusula de temporización

El operador de mutación incluido en este bloque, REPTIM, modifica el momento en el que se lanza el disparador. Las cláusulas `BEFORE` y AFTER se reemplazan entre sí.

3.1.2. Operadores de función del disparador

Estos operadores se aplican al cuerpo de la función invocada por el disparador e introducen los mismos cambios que los definidos para la cláusula `WHEN` de la cabecera del disparador, a excepción del correspondiente a la eliminación de dicha cláusula. No obstante, las funcionalidades a las que afectan son diferentes, ya que en el caso de la cláusula `WHEN` afectan al filtrado de las filas que lanzan el disparador, mientras que en este caso modifican la funcionalidad asociada a la ejecución del mismo. Los identificadores de operador son distintos. Para distinguirlos de los operadores de cabecera, se elimina el sufijo "`H`" (header) que aparece en dichos operadores de cabecera. De este modo, los operadores considerados son los siguientes: RLREP, ARREP, LGREP, NTDEL y VARREP.

3.2. Generación de mutantes

A partir de la especificación de un disparador, compuesta por un cabecera de disparador y una función invocada por dicho disparador, se generan mutantes mediante la aplicación de todos o algunos de los operadores de mutación disponibles. A continuación vamos a mostrar varios ejemplos de la generación de algunos de los operadores. Los listados 3.1 y 3.2 incluyen el código de la cabecera del disparador y la función asociada a las que vamos a aplicar las mutaciones que aparecen en los ejemplos incluidos.

```
CREATE TRIGGER customer_unsubscribe AFTER UPDATE ON customer FOR EACH ROW
WHEN (OLD.active IS DISTINCT FROM NEW.active)
EXECUTE PROCEDURE customer_unsubscribe();
```

Listado 3.1: *Disparador.*

```
CREATE FUNCTION customer_unsubscribe()
RETURNS trigger
LANGUAGE "plpgsql" AS $BODY$
BEGIN
    INSERT INTO customers_audits(customer_id, entry_date)
    VALUES (new.ID, current_timestamp);
    RETURN NEW;
END;
 $BODY$;
```

Listado 3.2: *Función del disparador.*

El primero de los mutantes generados corresponde al operador de mutación VARREP. Este operador consiste en aplicar el reemplazo de `NEW` por `OLD` y viceversa en la función asociada al disparador. Al aplicar dicho operador, el mutante obtenido aparece en el listado 3.3 con el cambio realizado resaltado en color rojo.

```
CREATE FUNCTION customer_unsubscribe()
RETURNS trigger
LANGUAJE "plpgsql" as $BODY$
BEGIN
    INSERT INTO customers_audits(customer_id, entry_date)
    VALUES (old.ID, current_timestamp);
    RETURN NEW;
END;
$BODY$;
```

Listado 3.3: *Mutación VARREP.*

El segundo de los operadores de mutación aplicados es VAREPH. Este operador tiene el mismo funcionamiento que el VARREP definido anteriormente, pero éste se aplica sobre la cláusula `WHEN` del disparador, y no la función. Este operador genera dos mutantes. Esto se debe a que existen dos ocurrencias de `OLD` o `NEW` en la cabecera. El primero de ellos aparece en el listado 3.4 que ha realizado un cambio de `NEW` por `OLD`.

```
CREATE TRIGGER customer_unsubscribe AFTER UPDATE ON customer FOR EACH ROW
WHEN (old.active is distinct from old.active)
EXECUTE PROCEDURE customer_unsubscribe();
```

Listado 3.4: *Mutación VAREPH.*

El tercer mutante generado, de la aplicación del mismo operador, reemplaza la ocurrencia de `OLD` por `NEW`, tal como aparece en el listado 3.5.

```
CREATE TRIGGER customer_unsubscribe AFTER UPDATE ON customer FOR EACH ROW
WHEN (new.active is distinct from new.active)
EXECUTE PROCEDURE customer_unsubscribe();
```

Listado 3.5: *Mutación VAREPH.*

El siguiente mutante ha sido generado por el operador de mutación REPTIM. Este operador consiste en el reemplazo del modificador `AFTER` por `BEFORE`, o viceversa. En nuestro ejemplo ha sido sustituido la cláusula `AFTER` por `BEFORE` como vemos en el listado 3.6.

```
CREATE TRIGGER customer_unsubscribe BEFORE UPDATE ON customer FOR EACH
   ROW
WHEN (old.active is distinct from new.active)
EXECUTE PROCEDURE customer_unsubscribe();
```

Listado 3.6: *Mutación REPTIM.*

Por último, vamos a analizar el último de los operadores aplicados, DELWC, cuyo funcionamiento se basa en la eliminación de la cláusula `WHEN` del disparador.

```
CREATE TRIGGER customer_unsubscribe AFTER UPDATE ON customer FOR EACH ROW
EXECUTE PROCEDURE customer_unsubscribe();
```

Listado 3.7: *Mutación DELWC.*

Como podemos apreciar en el listado 3.7. ha desaparecido la condición `WHEN` de nuestro trigger.

3.3. Ejecución del proceso de pruebas

La ejecución de las pruebas con mutantes es un proceso complejo que implica dos aspectos importantes: la comparación de los resultados de la ejecución por un lado, y el mecanismo que permite la ejecución de múltiples pruebas de forma independiente en la base de datos de pruebas.

3.3.1. Comparación de los resultados de la ejecución de disparadores

Respecto a la comparación de resultados de la ejecución de disparadores, la aplicación de la técnica de pruebas de mutación en el contexto de una base de datos es diferente y más compleja que el caso general de pruebas de programas de propósito general o en el caso concreto de consultas SQL. En el caso de los programas de propósito general, la ejecución del proceso de pruebas consiste en comparar la salida del programa original con la del programa mutado para los casos de prueba que correspondan; esta salida puede ser textual o de otra forma, pero depende del caso concreto de pruebas para determinar la comprobación a realizar sobre los resultados del programa mutado. En el ámbito de las consultas SQL sobre bases de datos relacionales, la comprobación de mutantes se limita a comparar el resultado producido por la propia consulta SQL mutada con el resultado de la consulta original.

En el contexto de las pruebas sobre disparadores, la "salida" del disparador es más compleja, pues es el estado completo de las tablas de la base de datos de pruebas. Esto es debido a que el efecto de la ejecución de un disparador es, en la mayor parte de los casos, la modificación del contenido de alguna de las tablas de la base de datos, producida por la ejecución de sentencias DML `INSERT`, `DELETE` o `UPDATE` que forman parte del código del disparador. Por lo tanto, determinar si un mutante ha sido matado por algún test requiere comparar el estado de todas las tablas de la base de datos.

Para poder comparar el resultado de la ejecución de un test cuando está activado el

disparador original con el obtenido de la ejecución en el caso de que el disparador activado corresponda a un mutante debemos crear en el entorno de ejecución de las pruebas dos instancias de la base de datos de pruebas sobre las que se ejecutarán los tests utilizando ambos disparadores, respectivamente. Nos referiremos a estas bases de datos como base de datos del disparador original y base de datos del disparador mutante, respectivamente.

Para poder comparar el estado de ambas bases de datos, se recorren todas las tablas de la base de datos y se compara cada tabla de esta base de datos con la tabla de la base de datos del disparador original. Para comparar dos tablas se ha utilizado un procedimiento simplificado que funciona correctamente con bases de datos sencillas. En este procedimiento se realizan las siguientes dos comprobaciones:

1. Si el número de filas de la tabla obtenida tras ejecutar el test con el disparador original activado y la tabla obtenida tras ejecutar el test con el disparador mutado es diferente, los resultados son distintos y por consiguiente el test mata al mutante. Si por el contrario ambas tablas contienen el mismo número de filas, se realiza la siguiente comprobación.

2. Se produce la unión de la tabla obtenida tras ejecutar el test con el disparador original activado y la tabla obtenida tras ejecutar el test con el disparador mutado activado. Si la unión devuelve el mismo numero de filas que en el paso anterior, el mutante está vivo, en otro caso el test ha matado al mutante.

Este procedimiento necesita que la base de datos cumpla algunos requisitos: los datos contenidos en la tabla no deben depender del instante en el que se generan las filas sobre la tabla (por ejemplo, las columnas de tipo autoincremento o la utilización de la función de hora del sistema en la sentencia de modificación de datos, `NOW()` en el caso de PostgreSQL). Esto es así por dos motivos: en primer lugar, porque la ejecución de los casos de prueba se realiza en instantes distintos en cada una de las bases de datos. En segundo lugar, porque en las operaciones de modificación de datos de un RDBMS no es posible garantizar que

las filas se procesarán exactamente en el mismo orden. A pesar de estas limitaciones, se ha considerado que este procedimiento permite evaluar el marco de pruebas satisfactoriamente, aunque se prevé mejorarlo como parte del trabajo futuro.

3.3.2. Mecanismo de ejecución de pruebas independientes

El segundo aspecto relevante en el proceso de ejecución de pruebas consiste en el mecanismo que permita la ejecución de múltiples pruebas de forma independiente en las bases de datos de pruebas, tanto en la base de datos del disparador original, como en la base de datos del disparador mutado. Este proceso es más complejo que en el caso general de marcos de pruebas sobre programas de propósito general. En cada una de las ejecuciones de un caso de prueba con un mutante, la base de datos debe tener inicialmente un estado conocido y debe ser siempre el mismo para todas las pruebas con todos los mutantes y con el disparador original. Por lo tanto, se requiere restaurar una y otra vez el estado de la base de datos a la situación inmediatamente anterior a la ejecución de cada prueba. En el contexto de un RDBMS esto es un proceso generalmente costoso y que depende del contenido y complejidad de cada base de datos. En el siguiente capítulo se detalla este proceso de ejecución de pruebas en la implementación realizada sobre el RDBMS PostgreSQL.

Capítulo 4

Implementación de la propuesta

En este capítulo se describe la implementación del marco propuesto para su automatización y se describen de las diferentes herramientas tecnológicas utilizadas para su desarrollo. Asimismo, se introducen las funcionalidades que ofrece la herramienta.

4.1. Estructura del sistema

Con el objetivo de automatizar el marco de pruebas de mutación propuesto se plantea el desarrollo de una aplicación que proporcione las siguientes funcionalidades:

- Gestión de usuarios: Creación y gestión de usuarios para su acceso al sistema.
- Gestión de disparadores: Gestión de disparadores y funciones de disparador asociados a diferentes bases de datos existentes en el sistema gestor de bases de datos PostgreSQL local.
- Gestión de conjuntos de casos de pruebas: Gestión de conjuntos de casos de pruebas que serán aplicados en el ámbito de las pruebas de mutaciones.
- Automatización del proceso de mutación: Generación de mutantes para una selección de operadores de mutación, aplicación de conjuntos de casos de prueba a los mutantes generados y presentación de resultados.

La arquitectura del sistema 4.1 se ha basado en una aplicación web que reside en un servidor que almacena toda la información necesaria y donde se realizará la ejecución de las pruebas de mutación. Se ha conformado de manera que se tiene un servidor que utiliza diferentes bases de datos, una que es interna, y luego las externas que introduce el propio usuario para el uso de la herramienta. El sistema dispone de varios tests que se aplican contra los disparadores de las bases de datos. Y finalmente los clientes que se conectan para poder operar con el sistema.

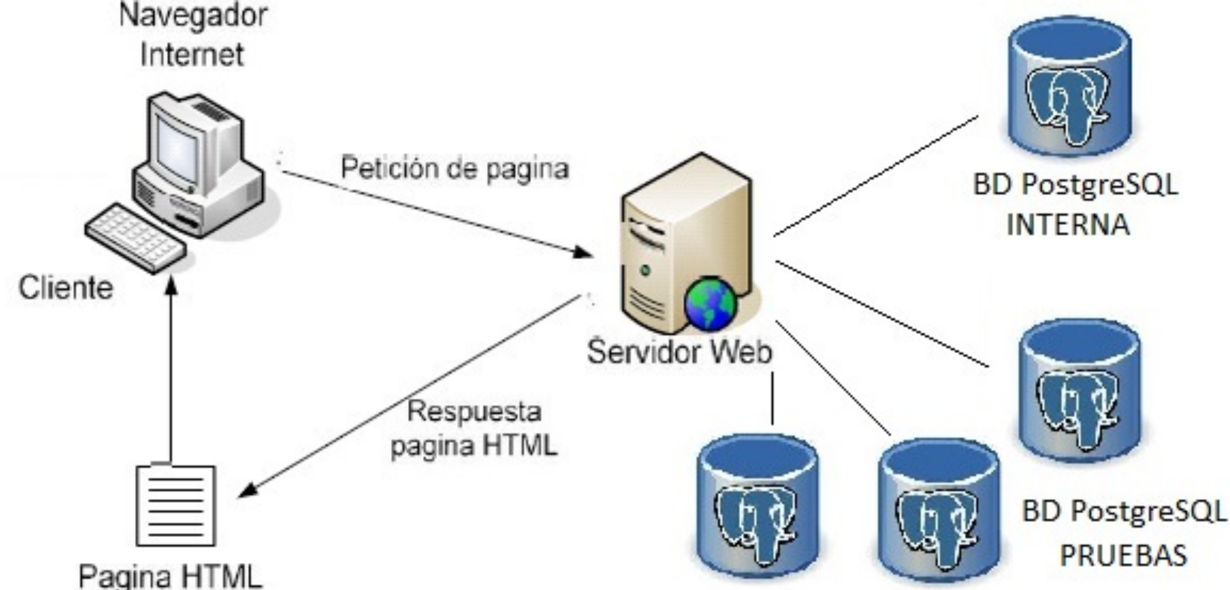


Figura 4.1: *Arquitectura de la aplicación.*

4.2. Herramientas y sistemas utilizados

En el desarrollo e implementación de la herramienta se utilizaron principalmente las tecnologías que se referencian a continuación.

4.2.1. PostgreSQL

PostgreSQL[16] es un sistema gestor de bases de datos objeto-relacional. Es de código abierto, gratuito, multiplataforma, y en el que se pueden manejar grandes cantidades de

datos. La aplicación implementada en este trabajo se ha basado en este sistema por su disponibilidad y el amplio conjunto de funcionalidades que proporciona.

En este trabajo se ha utilizado PostgreSQL tanto para desarrollar el marco pruebas como para almacenar los datos internos de la aplicación. Por este motivo, se han utilizado múltiples bases de datos dentro del RDBMS. El sistema implementado puede utilizar múltiples bases de datos de pruebas sobre las que se ejecutarán las pruebas de mutación de los disparadores definidos por los usuarios.

La aplicación está diseñada para que todas las bases de pruebas estén almacenadas en la instancia de PostgreSQL instalada en el servidor de la aplicación, pero el sistema se puede extender fácilmente para utilizar bases de datos instaladas en otros servidores.

4.2.2. Python

Python[8] es un lenguaje de programación de código de abierto que se utiliza para el desarrollo web y que en la mayoría de ocasiones se usa incrustado en páginas HTML. Este lenguaje se ha usado en este trabajo para realizar las conexiones que se han empleado entre la base de datos y la aplicación, así como la gestión de datos de entrada y salida dotando a su vez de formato a las páginas HTML. Además también se ha utilizado para la administración de las sesiones de usuarios.

4.2.3. HTML

HTML[3] es un lenguaje de marcado de hipertexto que se usa para construir el contenido de las páginas web. Se basa en etiquetas, a través de las cuales se definen los documentos.

El lenguaje HTML ha sido utilizado en el desarrollo de la aplicación para diseñar el interfaz. Gracias a este lenguaje de marcado, se ha distribuido los distintos elementos de la misma: en la parte izquierda un menú lateral, con el que se navega a todas los módulos de la aplicación, y en la parte central el contenido correspondiente a cada una de estos módulos.

4.2.4. PHP

PHP[6] es un lenguaje de programación de código abierto que se utiliza para el desarrollo web y que en la mayoría de ocasiones se usa incrustado en páginas HTML.

Este lenguaje se ha usado en este trabajo para realizar las conexiones entre la base de datos y la aplicación. También se ha utilizado en la gestión de datos de entrada y salida, dotando de formato a las páginas HTML, así como en la administración de las sesiones de usuarios.

4.2.5. Javascript

Javascript[4] es un lenguaje de programación interpretado. Generalmente se usa en la parte cliente, formando parte de un navegador web que proporciona mejoras en la interfaz del usuario y de las páginas web dinámicas.

En nuestra aplicación se ha usado para añadir funcionalidad a través de eventos en los formularios, lo que ha facilitado la implementación del proceso de mutación de los disparadores.

4.2.6. Cascading Style Sheets

Cascade Style Sheets, más conocido como CSS[1], es un lenguaje de reglas de estilo que se utiliza para definir la presentación del contenido HTML.

En este trabajo se ha utilizado para establecer los colores, tipos de letra y la maquetación de los elementos que componen los formularios.

4.2.7. pgAdmin4

pgAdmin4[5] se trata de la herramienta gráfica que gestiona y administra instalaciones de PostgreSQL. Esta utilidad proporciona una interfaz administrativa en la que se dispone de un editor de código procedural, además de un módulo de consulta SQL.

4.2.8. Entornos de desarrollo

Se han utilizado diversos entornos de desarrollo entre ellos Microsoft Visual Studio[9] y Eclipse[2]. Microsoft Visual Studio es un entorno de desarrollo compatible con diversos múltiples lenguajes de programación como PHP, Java, Python, C++ y C#. Permite crear aplicaciones que se conecten con páginas web, dispositivos móviles o embebidos. Eclipse es una plataforma de software que permite integrar diferentes aplicaciones en forma de plug-ins para construir un Integrated Develoment Environment.

4.3. Modelo de la base de datos interna

La aplicación desarrollada utiliza una base de datos interna para almacenar la información necesaria de usuarios, disparadores, mutantes y conjuntos de casos de prueba. El diagrama que se muestra en la figura 4.2 presenta el modelo de la base de datos diseñado para la aplicación. La base de datos almacena los principales datos manejados por el sistema: usuarios (*users*), operadores de mutación (*operators*), conjuntos de pruebas (*suites, test_suites, tests*), A su vez, almacena los resultados de la aplicación de los conjuntos de pruebas a los disparadores. Estos resultados nos permiten establecer la idoneidad de dichos conjuntos de pruebas para la detección de los errores simulados en los mutantes mediante la aplicación de los operadores de mutación.

La tabla `users` almacena los identificadores de usuarios que pueden acceder a la aplicación. Esta tabla almacena el login del usuario, que debe ser un correo electrónico, que debe ser único, y su correspondiente password. Además de guardar la información de si quiere compartir la información que tiene en la aplicación con el resto de los usuarios registrados en la misma a través del campo `compartirinfo`.

La tabla `triggers` contiene la información de los disparadores sobre los que se van a realizar pruebas de mutación. Contiene el código fuente de la cabecera (declaración del disparador) y la función asociada al disparador, así como el identificador de la base de datos de pruebas sobre la que está definido el disparador. Esta información es introducida por un

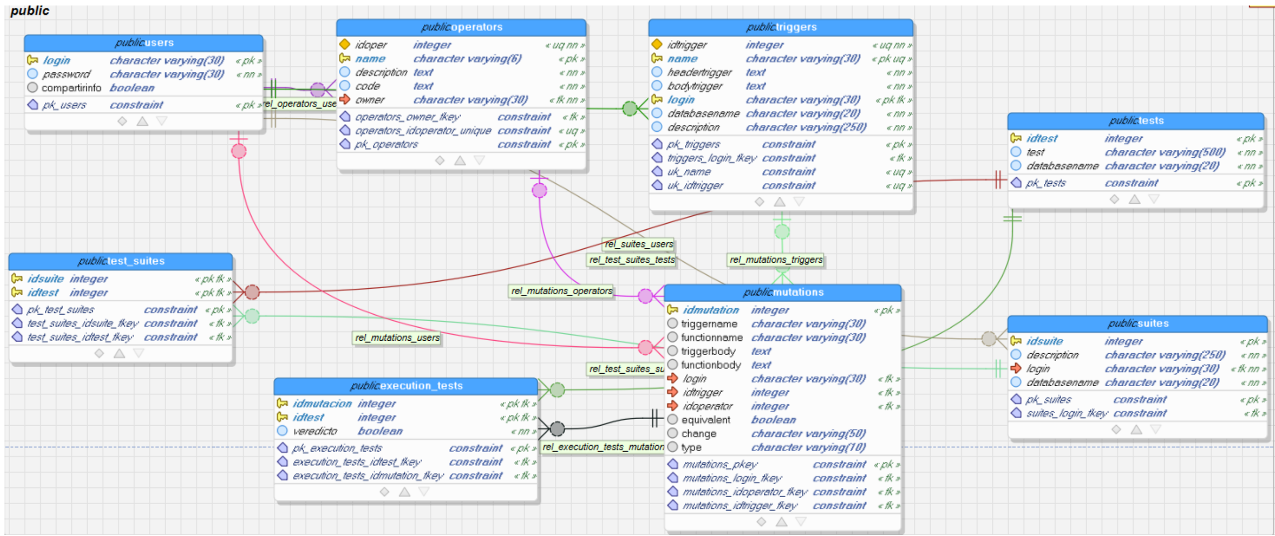


Figura 4.2: *Modelo Lógico de la Base de Datos.*

usuario en la aplicación.

La tabla `suites` contiene la información sobre los conjuntos de casos de prueba que los usuarios han introducido en la aplicación. Una suite está formada por uno o varios casos de prueba, almacenados en la tabla `tests`. Esta tabla almacena cada uno de los casos de prueba que se ejecutan en el proceso de pruebas mutación. Un caso de prueba es una sentencia SQL de modificación de datos (`INSERT`, `UPDATE` o `DELETE`). Cada caso de prueba debe indicar el identificador de la base de datos de prueba sobre la que está definido. La relación entre los casos de prueba y los conjuntos de casos de prueba está representada a través de una tabla intermedia, que es la tabla `test_suites` que se encarga de vincular los casos de prueba con la o las suites que los incluyen.

La tabla `operators` contiene la descripción de los operadores de mutación que ofrece la aplicación al usuario.

Las seis tablas descritas hasta el momento contienen la información necesaria para definir un proceso de mutación de disparadores. La información contenida en estas tablas es introducida por los usuarios del sistema. A partir de esta información, la aplicación realiza dos procesos fundamentales: genera un conjunto de mutantes para el disparador elegido y

los operadores de mutación seleccionados, y a continuación ejecuta estos mutantes respecto a un conjunto de casos de prueba seleccionado por el usuario, con el objetivo de verificar si los mutantes son matados por el conjunto de casos de prueba. El resultado de estos dos procesos se almacena en las tablas que se describen a continuación.

La tabla `mutations` contiene la información asociada a cada uno de los mutantes generados. En ella se almacena, entre otras cosas, el nombre y el código fuente mutado del disparador, tanto la cabecera como la función asociada. En la información almacenada de cada mutante tiene especial relevancia el campo `equivalent` que permite al usuario decir que un disparador mutante es equivalente a otro. La determinación de los mutantes equivalentes es un proceso manual que debe realizar el usuario.

Para el proceso de mutación también es fundamental la tabla `execution_tests`. En esta tabla se almacena el resultado de la ejecución de los casos de prueba sobre los disparadores mutantes. Contiene en el campo `veredicto` el resultado final tras la ejecución de dicho test sobre la base de datos: si el mutante ha sido matado por alguno de los casos de prueba o si permanece vivo.

4.4. Funcionalidades de la Aplicación

En esta sección se describen las distintas funcionalidades de la aplicación implementada para evaluar esta propuesta de pruebas de mutación de disparadores.

4.4.1. Acceso Usuarios

El formulario de conexión a la aplicación mostrado en la figura 4.3 permite acceder al sistema mediante el uso del usuario y password así como realizar un nuevo registro si el usuario aún no lo ha hecho o recuperar la password si el usuario no la recuerda.

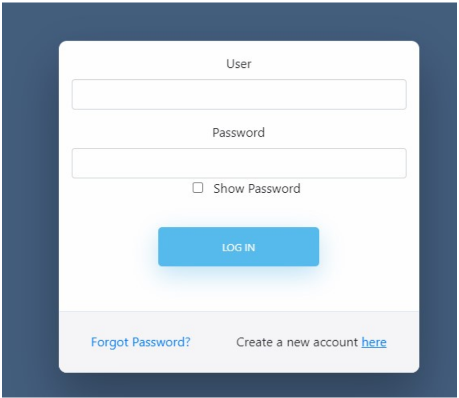


Figura 4.3: *Acceso usuarios.*

4.4.2. Menú principal

Una vez el usuario se ha conectado correctamente, se accede al formulario principal del sistema, que se muestra en la figura 4.4. A la izquierda aparece el menú desde el que el usuario puede acceder a cada uno de los módulos de la aplicación. Por una parte, aparecen las entradas que permiten gestionar los disparadores, los conjuntos de pruebas y los operadores de mutación. Por otra parte, permite acceder al módulo principal de generación de mutantes y ejecución del proceso de pruebas de mutaciones. Además, existe una opción para acceder al perfil de usuario, donde se permite actualizar la información del usuario.

4.4.3. Gestión de disparadores

Este formulario presenta un listado paginado de todos los disparadores registrados en la aplicación, como puede verse en la Figura 4.5. Cada uno de ellos tiene asociados diferentes iconos que permiten realizar diferentes acciones sobre cada uno de ellos. Las opciones de edición y eliminación solo están disponibles para los disparadores de los que sea propietario el usuario conectado. En otro caso el usuario solo podrá visualizar el disparador si el propietario ha dado permisos para compartir dicha información. Tampoco podrán modificarse o eliminarse los disparadores a los que se haya aplicado el proceso de mutación. Todos los usuarios podrán registrar nuevos disparadores a través del formulario correspondiente, mostrado en la figura 4.6. El usuario deberá elegir un nombre, una breve descripción del

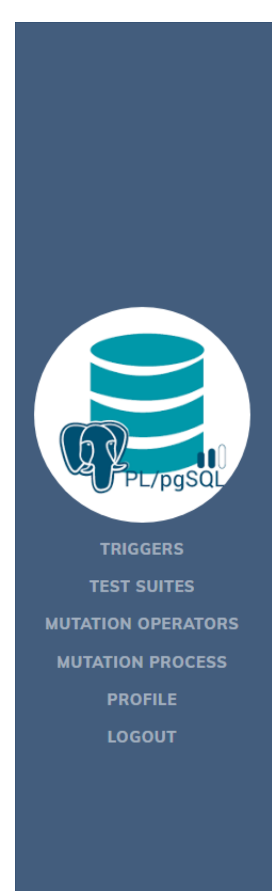


Figura 4.4: *Pantalla Principal.*

disparador, e introducir el código correspondiente tanto al disparador como a la función de disparador asociada. Además el usuario debe seleccionar la base de datos de pruebas en la que se ejecutará el disparador. Esta base de datos debe existir en la instancia de PostgreSQL donde está instalada la aplicación. Cuando se introduce un disparador en el sistema, se instala en la base de datos de pruebas para comprobar que no se producen errores sintácticos o de otro tipo. Si existe algún fallo, la aplicación mostrará el mensaje indicando el error, y el usuario tendrá que subsanarlo para que el disparador quede registrado en el sistema. Una vez comprobada la instalación del disparador, se desinstala y se almacena su código fuente en la base de datos interna de la aplicación. Además, se incluye en el listado global de disparadores registrados.

Figura 4.5: *Disparadores registrados.*

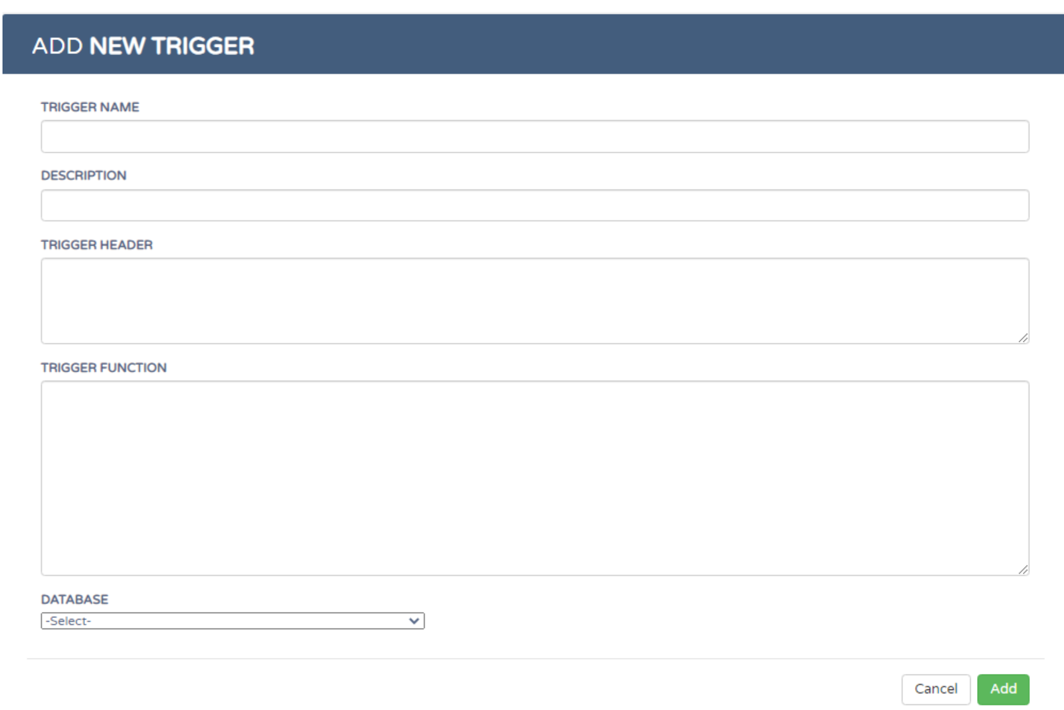


Figura 4.6: *Formulario nuevo disparador.*

4.4.4. Operadores de mutación

En esta pantalla, mostrada en la figura 4.7, se presentan los operadores de mutación disponibles en el sistema. En la versión actual de la aplicación esta lista no es editable, pues

el conjunto de operadores disponibles no es extensible. Sin embargo, se prevé la extensión futura del sistema para permitir añadir y modificar los operadores disponibles.

MUTATION OPERATORS

Acronym	Description
VARREP	Replace NEW - OLD
ADDEC	Add event clause
DELEC	Delete event clause
REPEC	Replace event clause
ARREP	Replace arithmetic operator (+,-,*,/)
LGREP	Replace logical operator (OR - AND)
REPTIM	Replace AFTER - BEFORE
REPRLV	Replace row operator
DELWC	Remove WHEN clause from statement
NTDEL	Remove logical operator NOT
RLREP	Replace relational operator (=;<>;<;<=;>;>=)
ARREPH	Replace arithmetic operator (+,-,*,/) in header
RLREPH	Replace relational operator (=;<>;<;<=;>;>=) in header
LGREPH	Replace logical operator (OR - AND) in header
VAREPH	Replace NEW - OLD in header
NTDELH	Remove logical operator NOT in header

Figura 4.7: *Operadores de mutación disponibles.*

4.4.5. Gestión conjuntos de pruebas

En este formulario se presenta un listado paginado similar al visto anteriormente para los disparadores, como se muestra en la figura 4.8. Los conjuntos de pruebas agrupan casos de prueba de forma que se pueden seleccionar para aplicar todos los casos del conjunto en un mismo proceso de ejecución de pruebas de mutación.

Al igual que ocurría con los disparadores, los conjuntos de casos de prueba solo pueden ser editados o eliminados por el usuario que los ha creado. Cualquier otro usuario del sistema puede utilizarlos pero no modificarlos.

Para incluir un nuevo conjunto de pruebas, mediante el formulario que aparece en la

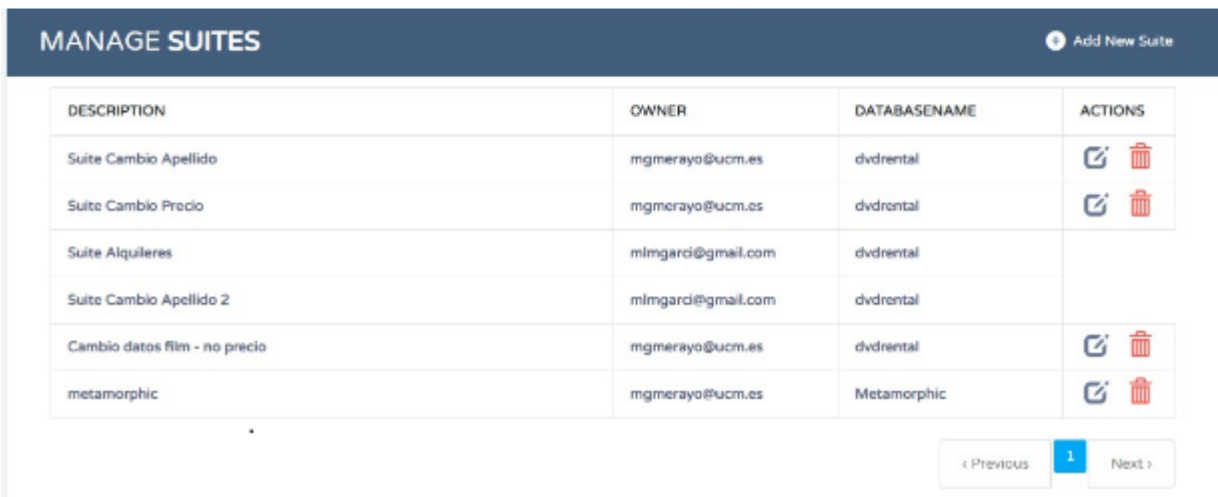


Figura 4.8: *Conjuntos de pruebas registrados.*

figura 4.9, el usuario debe completar el campo descripción y elegir la base de datos sobre la que se ejecutarán las instrucciones que constituyen los casos de prueba.

Cada caso de prueba estará formado por una única sentencia SQL de modificación de datos: `INSERT`, `UPDATE` o `DELETE`. La aplicación nos ofrece dos alternativas para indicar dichos casos de prueba. La primera opción, *Add new test*, permite indicar las instrucciones correspondientes a dichos casos de prueba en un campo de texto que aparece en el formulario correspondiente (figura 4.10). La segunda opción, *Load existing tests*, permite reutilizar casos de prueba incluidos en un conjunto de pruebas ya registrado. En el formulario que aparece en la figura 4.11, se seleccionará el conjunto de pruebas deseado, del cual el usuario puede seleccionar o bien todos los casos de prueba que incluye o un subconjunto de los mismos. Tan solo se podrán elegir conjuntos de casos de prueba correspondientes a la misma base de datos de pruebas a la que referencia el conjunto que se desea registrar.

Una vez indicados los casos de prueba, la aplicación los validará antes de registrarlos en la base de datos de la aplicación. La validación consiste en la ejecución de todos los casos de prueba uno por uno, revirtiendo los cambios realizados por cada uno de ellos. Este proceso es necesario porque los casos de prueba son independientes entre sí, por lo que se debe utilizar la base de datos elegida por el usuario.

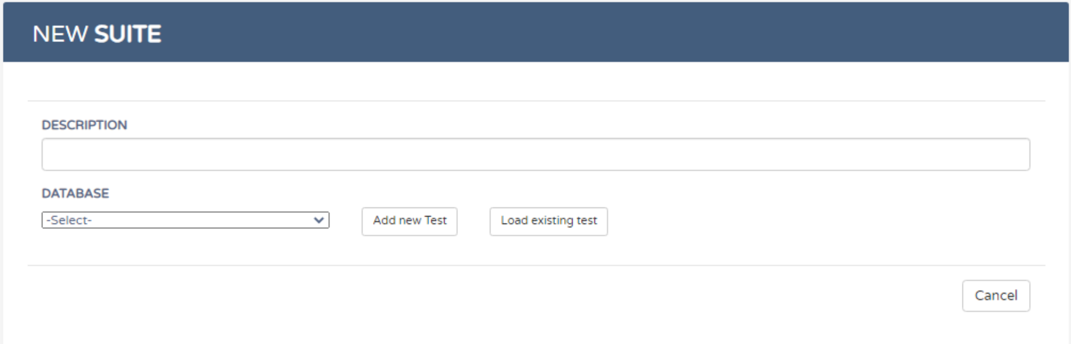


Figura 4.9: *Formulario nuevo conjunto de pruebas.*

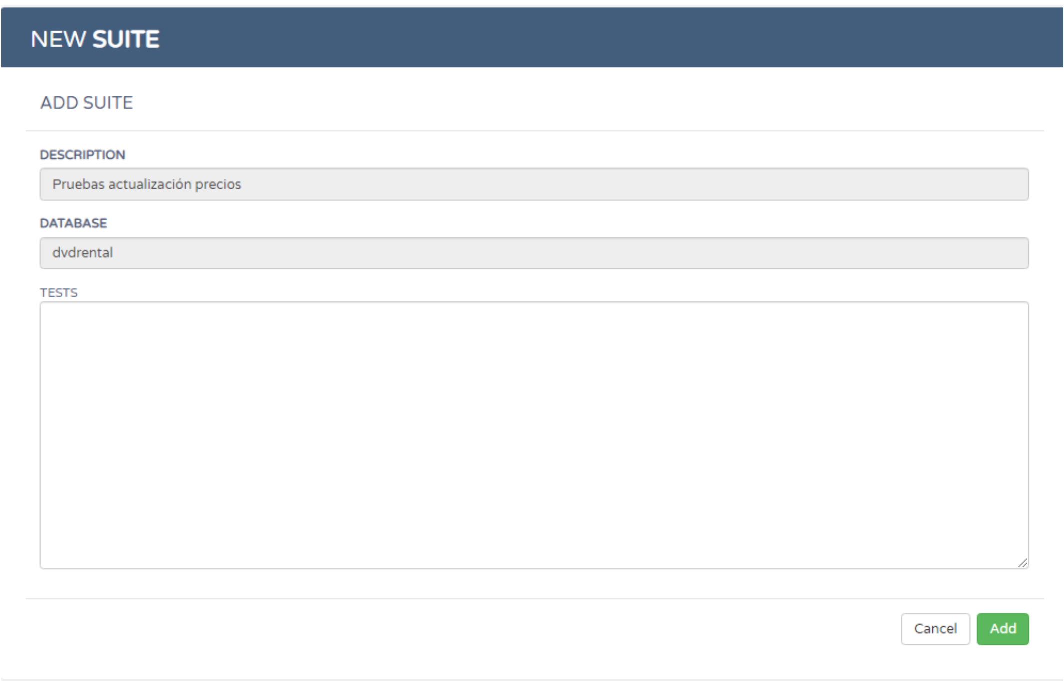


Figura 4.10: *Formulario inclusión nuevos conjuntos de pruebas.*

Si existe algún problema al validar el conjunto de casos de prueba, el sistema muestra un mensaje de error indicando que es lo que está ocurriendo, y por lo tanto se permite al usuario corregirlo. Una vez se han verificado que todos los casos de tests son correctos, se almacenan en la base de datos para hacer uso de ellos durante el proceso de mutación.

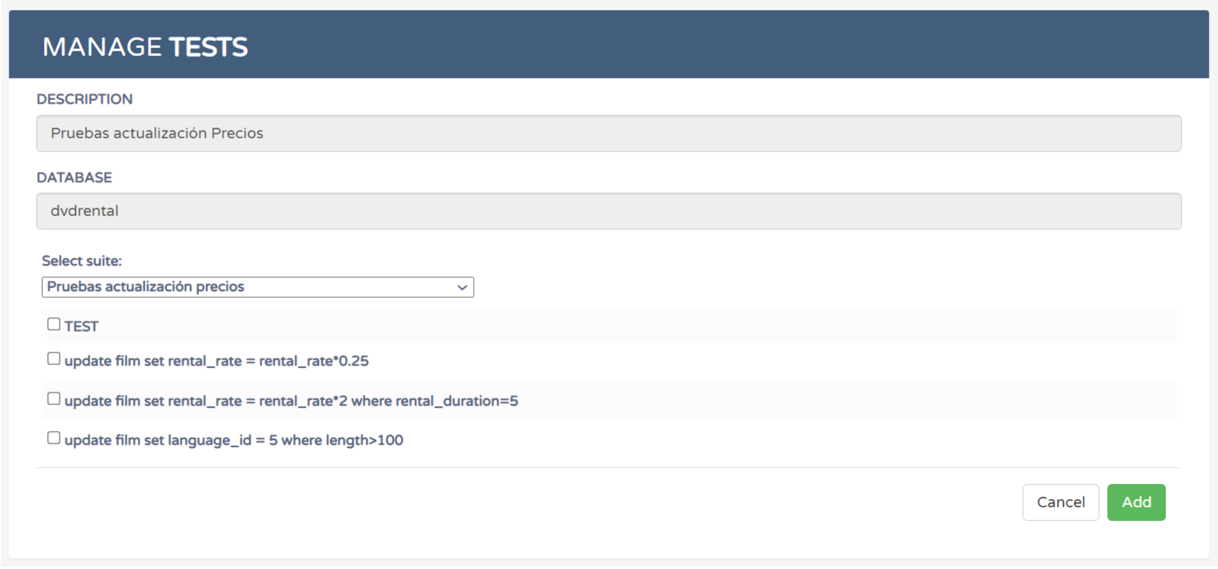


Figura 4.11: *Formulario uso conjuntos de pruebas existentes.*

4.4.6. Proceso de mutación

Esta opción permite al usuario aplicar la técnica de pruebas de mutación a los disparadores y conjuntos de pruebas registrados en el sistema.

En el formulario inicial que se muestra en la figura 4.12, el usuario debe elegir el disparador al que desea aplicar el proceso de pruebas de mutación, así como los operadores de mutación que se aplicarán a dicho disparador.

Una vez realizada la selección, el sistema aplicará al disparador, en caso de que sea posible, cada uno de los operadores, dando lugar a los mutantes correspondientes. En caso de no poder generar ningún mutante, el sistema informará al usuario. Si el proceso de generación de mutantes ha funcionado correctamente, se mostrará un nuevo formulario que presenta una relación de los mutantes generados, mostrado en la figura 4.13. Para cada mutante, se indica su identificador, si la mutación afecta al disparador o a la función asociada, el operador de mutación que ha generado el mutante y una breve descripción del mismo. También es posible, seleccionando un mutante, visualizar el código correspondiente a ese mutante, como se puede observar en el ejemplo mostrado en la figura 4.14.

Una vez generados los mutantes, el usuario deberá elegir un conjunto de pruebas para

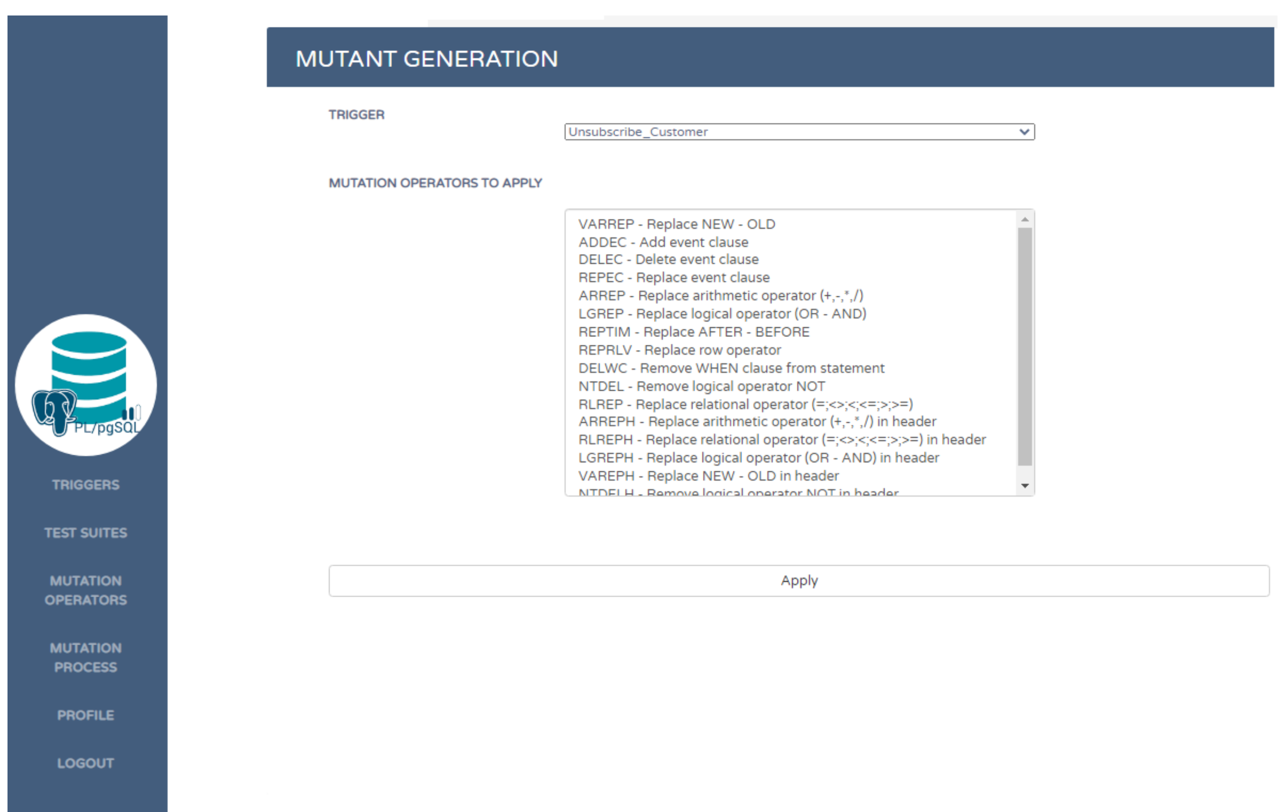


Figura 4.12: *Pruebas de mutación: selección disparador y operadores de mutación.*

ser evaluado respecto a dichos mutantes. Para ello deberá elegir una test suite de las disponibles en el combo mostrado en la parte inferior de la figura 4.13. En este combo solamente aparecerán las test suites de la base de datos de pruebas asociada al disparador elegido por el usuario.

En la figura 4.15 se muestra el resultado del proceso de mutación de un disparador después de aplicar los operadores de mutación. En la parte superior de la figura se observa el número de mutantes no equivalentes obtenidos, además del porcentaje de disparadores vivos y muertos resultantes. En la parte inferior de la pantalla de resultados, se puede visualizar una tabla donde aparece el listado de los disparadores mutantes vivos y los matados en la pestaña correspondiente. Además se permite al usuario visualizar los disparadores que aparecen en dicho listado.

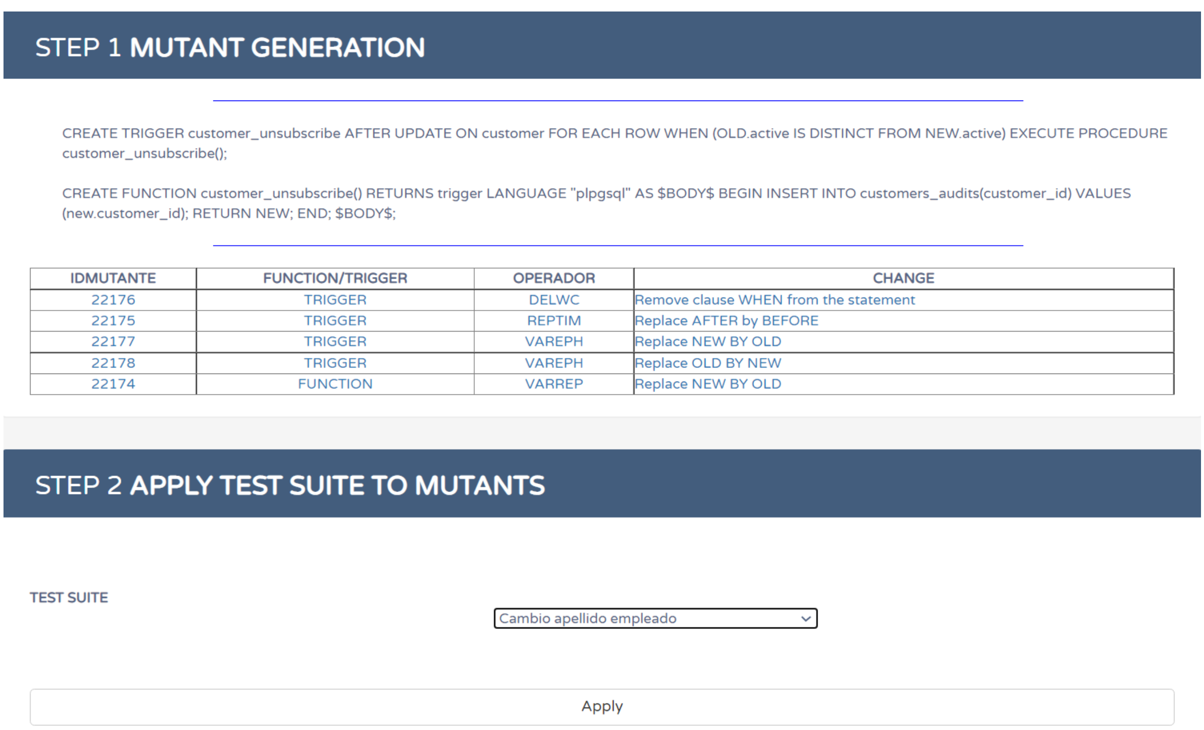


Figura 4.13: *Pruebas de mutación: mutantes generados.*

4.4.7. Detalles de implementación del proceso de ejecución de pruebas de mutación

Como se ha indicado en el capítulo anterior, la ejecución de pruebas de mutación es un proceso complejo porque es necesario ejecutar los casos de prueba de forma independiente entre sí y sobre conjuntos de tablas diferentes, que corresponden a las bases de datos del disparador original y de cada uno de los mutantes. Con este fin, es necesario realizar una serie de tareas que se describen a continuación.

1. En primer lugar se realiza una copia del estado inicial de la base de datos de pruebas sobre la que se ejecuta el disparador original y los mutantes. Esta copia será utilizada durante el proceso de pruebas para recuperar el estado inicial de la base de datos de pruebas. Para ello, se consulta el diccionario de datos de PostgreSQL para obtener el nombre de todas las tablas de la base de datos de pruebas y se copia cada una de ellas sobre un nuevo conjunto de tablas renombradas añadiendo el prefijo `backup_` a cada

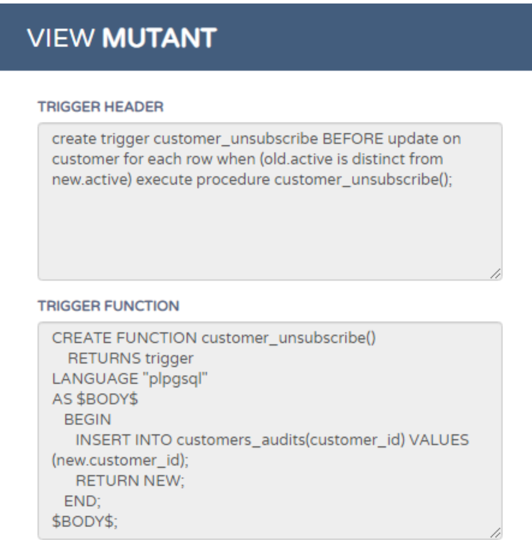
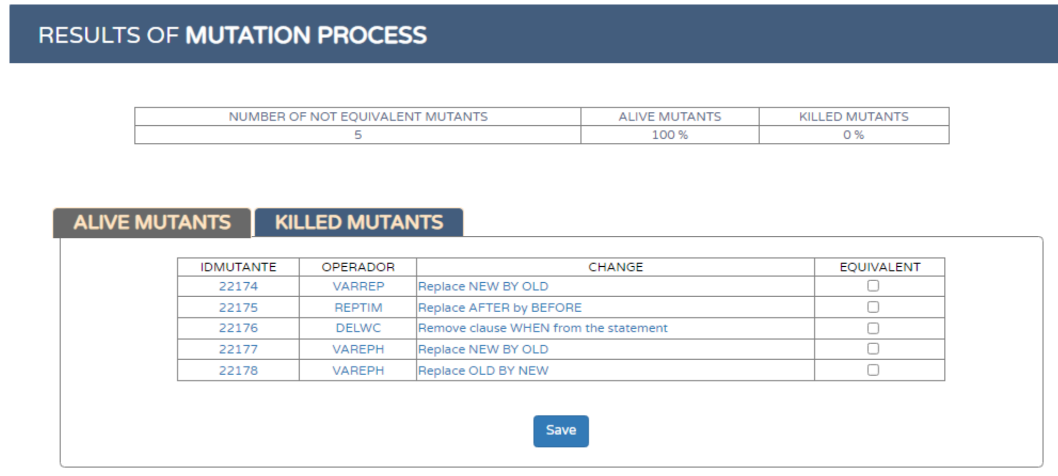


Figura 4.14: *Código generado de un mutante.*



NUMBER OF NOT EQUIVALENT MUTANTS	ALIVE MUTANTS	KILLED MUTANTS
5	100 %	0 %

IDMUTANTE	OPERADOR	CHANGE	EQUIVALENT
22174	VARREP	Replace NEW BY OLD	☐
22175	REPTIM	Replace AFTER by BEFORE	☐
22176	DELWC	Remove clause WHEN from the statement	☐
22177	VAREPH	Replace NEW BY OLD	☐
22178	VAREPH	Replace OLD BY NEW	☐

Figura 4.15: *Tabla de resultados.*

una de las tablas originales.

2. A continuación se recorre la lista de casos de prueba del conjunto seleccionado por el usuario. Por cada caso de prueba se realizan los siguientes pasos:

   2.1. Se instala el disparador original y la función asociada.

   2.2. Se ejecuta el caso de prueba sobre la base de datos del disparador original. Se guarda el estado de la base de datos en un conjunto de tablas con prefijo `bdtrigger_`.

2.3. Se desinstala el disparador original y se restaura el estado de la base de datos de pruebas con el conjunto de tablas con prefijo `backup_`.

2.4. Se inicia una transacción en la base de datos.

2.5. Se recorre la lista de mutantes generados que permanecen vivos. Por cada mutante se realizan los siguientes pasos:

2.5.1. Se instala el disparador mutante y se ejecuta el caso de prueba. Si se produce un error en la ejecución, se marca el mutante como *matado*, pues la mutación produce un estado distinto de la base de datos de pruebas.

2.5.2. Se compara el estado de la base de datos de pruebas con la copia almacenada con prefijo `bdtrigger_`. Si son distintas, se marca el mutante como *matado*.

2.5.3. Se revierte la ejecución del caso de prueba deshaciendo la transacción con `ROLLBACK`.

2.5.4. Se actualiza el estado de la ejecución del mutante en la base de datos interna con el veredicto alcanzado (en la tabla `execution_test`) y se confirma la transacción.

Finalmente, aparecerá una pantalla en la aplicación en la que podremos ver cuales de los test lanzados contra los disparadores mutantes generados han sido matados y cuales han quedado vivos. La comparación de las bases de datos de pruebas realizada en el paso 2.5.2. se realiza comparando las tablas una a una con el procedimiento descrito en el apartado 3.3.1. A modo de ejemplo, en las figuras 4.16 y 4.17 se pueden ver dos fragmentos de código en el que se muestran el copiado y el restaurado de la base de datos de pruebas inicial, respectivamente.

```
//HACEMOS EL BACKUP ANTES DE EJECUTAR LOS TRIGGERS

$conexionTablas =   pg_connect($confConexion);

$listaTablasOriginal = pg_query("select tablename as table from pg_tables  where schemaname = 'public';");

while ($listaTablas = pg_fetch_array($listaTablasOriginal, null, PGSQL_ASSOC)) {
    $nombreTabla = $listaTablas['table'];
    pg_query("drop table if exists backup_".$nombreTabla.";");
}
pg_close($conexionTablas);

$conexionBackup = pg_connect($confConexion);

$tablas = pg_query($conexionBackup,"select tablename as table from pg_tables  where schemaname = 'public';");

while ($listaTablas = pg_fetch_array($tablas, null, PGSQL_ASSOC)) {

        $nombreTabla = $listaTablas['table'];
        pg_query("create table if not exists backup_".$nombreTabla." as (select * from ".$nombreTabla.");");
}

pg_close($conexionBackup);
```

Figura 4.16: *Fragmento de código fuente que copia la base de datos de pruebas inicial.*

```
//Restauración de la copia original

$conexionBackup = pg_connect($confConexion);

$tablas = pg_query($conexionBackup,"select tablename as table from pg_tables where schemaname = 'public' and tablename not like 'backup_%' and tablename not like 'bdtrigger_%';");

while ($listaTablas = pg_fetch_array($tablas, null, PGSQL_ASSOC)) {
    $nombreTabla = $listaTablas['table'];
    pg_query($conexionBackup,"truncate ".$nombreTabla." cascade;");
}
$tablas = pg_query($conexionBackup,"select tablename as table from pg_tables where schemaname = 'public' and tablename not like 'backup_%' and tablename not like 'bdtrigger_%';");

while ($listaTablas = pg_fetch_array($tablas, null, PGSQL_ASSOC)) {
    $nombreTabla = $listaTablas['table'];
    pg_query($conexionBackup,"insert into ".$nombreTabla."(select * from backup_".$nombreTabla.");");
}
pg_query($conexionBackup,"COMMIT;");
pg_close($conexionBackup);
```

Figura 4.17: *Fragmento de código fuente que restaura la base de datos de pruebas inicial.*

Capítulo 5

Experimentos y pruebas realizadas

En este capítulo se estudian algunos experimentos realizados para ilustrar el funcionamiento del sistema que se ha descrito en el capítulo anterior.

Como se ha podido comprobar en capítulos anteriores, para un mismo disparador se pueden aplicar diversos operadores de mutación y cada uno de ellos puede generar uno o varios mutantes, por lo que el resultado obtenido varía en función de diversos parámetros.

Para dar un resultado al usuario, nos basamos en la cantidad de mutantes vivos y muertos que hemos obtenido tras aplicar los operadores de mutación y nuestros casos de test. Una vez que se introduce el disparador a analizar en el sistema y se seleccionan los operadores de mutación a aplicar, se generan los mutantes correspondientes a estos operadores y se verifica que los mutantes sean correctos. Si durante este proceso se generan mutantes incorrectos que producen errores de compilación, automáticamente son descartados. Tambien son descartados aquellos mutantes que provocan un error en el proceso de creación en la base de datos. Los mutantes restantes son los que se utilizarán para realizar el proceso de pruebas de mutación. Para ello, se elige un conjunto de casos de prueba que se aplicará a cada uno de los mutantes. Con estos casos de prueba se obtiene un veredicto `TRUE` o `FALSE` en función de los resultados obtenidos.

Este veredicto se consigue una vez se han lanzado los test del conjunto de pruebas contra el disparador original y el mutante. Si al lanzar alguno de los tests contra el mutante, la base de datos sufre modificaciones diferentes a las producidas por el disparador original, el

veredicto sobre ese mutante es que ha sido "matado" por el caso de prueba. Si el estado de la base de datos después de ejecutar el test con el disparador mutante es el mismo al producido con el disparador original, el veredicto del sistema es que el mutante permanece "vivo". Para que un mutante sea matado es suficiente con que un solo caso de test lo mate; para que un mutante permanezca vivo, es necesario que todos los tests produzcan el mismo estado de la base de datos que el producido por el disparador original.

Para ilustrar este proceso se van a exponer algunos ejemplos de prueba realizados en los siguientes apartados.

5.1. Experimento 1

El primer ejemplo es un caso sencillo de disparador que inserta una fila en una tabla si se cumple una condición sencilla en la tabla `employees`. El código del disparador se muestra a continuación:

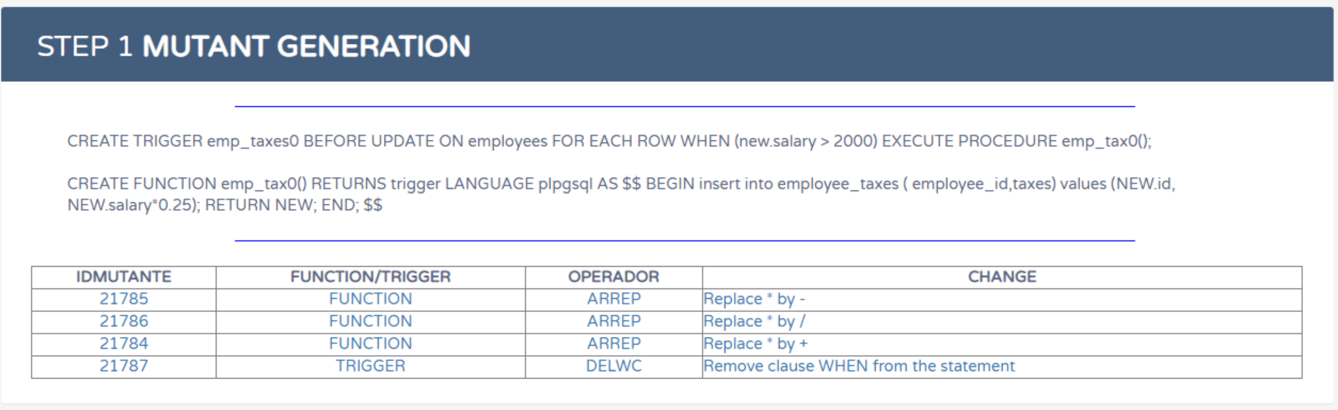
```
CREATE FUNCTION emp_tax0() RETURNS trigger LANGUAGE plpgsql AS
$$
BEGIN
    insert into employee_taxes ( employee_id,taxes)
    values (NEW.id, NEW.salary*0.25);
    RETURN NEW;
END;
$$;

CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
WHEN (new.salary > 2000)
EXECUTE PROCEDURE emp_tax0();
```

Para realizar este experimento se ha considerado que la tabla `employees` contiene los siguientes datos:

id	first_name	last_name	Salary
44	'Pepe'	'Lopez'	2500

Se seleccionan dos de los operadores de mutación disponibles en la aplicación: ARREP (reemplazamiento de operador aritmético) Y DELWC (eliminación de la cláusula WHEN).

Con ellos, se han generado los 4 mutantes que se pueden ver en la figura 5.1 .



STEP 1 **MUTANT GENERATION**

CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW WHEN (new.salary > 2000) EXECUTE PROCEDURE emp_tax0();

CREATE FUNCTION emp_tax0() RETURNS trigger LANGUAGE plpgsql AS $$ BEGIN insert into employee_taxes (employee_id,taxes) values (NEW.id, NEW.salary*0.25); RETURN NEW; END; $$

IDMUTANTE	FUNCTION/TRIGGER	OPERADOR	CHANGE
21785	FUNCTION	ARREP	Replace * by -
21786	FUNCTION	ARREP	Replace * by /
21784	FUNCTION	ARREP	Replace * by +
21787	TRIGGER	DELWC	Remove clause WHEN from the statement

Figura 5.1: *Experimento 1. Listado de mutantes generados.*

Una vez se han generado los mutantes, el siguiente paso es elegir y aplicar una test suite. En este experimento se ha elegido una que está formada por un único test:

```
UPDATE EMPLOYEES SET SALARY = 2600 WHERE ID = 44;
```

Al aplicar dicho test sobre cada uno de los mutantes, se ha obtenido el resultado mostrado en la figura 5.2.

RESULTS OF **MUTATION PROCESS**

NUMBER OF NOT EQUIVALENT MUTANTS	ALIVE MUTANTS	KILLED MUTANTS
4	25 %	75 %

Figura 5.2: *Experimento 1. Resultado del proceso de mutación.*

El resultado es de 1 mutante vivo, que es el que corresponde al operador DELWC, y 3 mutantes matados, que son los generados por el operador ARREP. En la página de resultados se pueden visualizar tanto los mutantes que han quedado vivos (figura 5.2) como los mutantes matados, mostrados en la figura 5.3.

Para entender mejor este ejemplo, se mostrarán cada uno de los mutantes y los motivos por los cuales han sido matados o permanecen vivos.

ALIVE MUTANTS | KILLED MUTANTS

IDMUTANTE	OPERADOR	CHANGE
22110	ARREP	Replace * by +
22111	ARREP	Replace * by -
22112	ARREP	Replace * by /

Figura 5.3: *Experimento 1. Listado de mutantes matados.*

Primero se estudia el mutante con ID 21787, que corresponde a la aplicación del operador DELWC, eliminación de la clausula `WHEN` en la sentencia de declaración del disparador. La sentencia que resulta de la aplicación de este operador es la siguiente:

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
EXECUTE PROCEDURE emp_tax0();
```

Si se ejecuta el test con el disparador original, al ser el salario del empleado 44 mayor de 2000 después de ejecutar el test, se cumple la condición de la cláusula `WHEN`, por lo que se lanzará el disparador y por lo tanto se modifica el estado de la base de datos. Por otra parte, el disparador mutante en el que se ha eliminado la condición, el disparador se ejecuta en cualquier caso, con el código de la función original. Por lo tanto, el resultado de ejecutar el disparador original y el disparador mutante va ser el mismo, con lo que el sistema produce como veredicto que dicho mutante ha quedado vivo.

Los otros tres mutantes generados, con ID 21784, 21785 Y 21786, son los correspondientes al operador de mutación ARREP. Este operador sustituye uno a uno los operadores aritméticos contenidos en el disparador por el resto de los operadores aritméticos disponibles. En este caso, se sustituye el operador aritmético * por los operadores +,-,/, generando tres mutantes distintos. A continuación se muestra el código del primero de los mutantes generados:

```
CREATE FUNCTION emp_tax0() RETURNS trigger LANGUAGE plpgsql AS
$$
    BEGIN
        insert into employee_taxes ( employee_id,taxes)
        values (NEW.id, NEW.salary+0.25);
```

```
        RETURN NEW;
    END;
$$;
```

Si se ejecuta el test con el disparador original, en la tabla `employee_taxes` se introduce una nueva fila con el importe resultante de multiplicar el salario de los empleados por el valor 0.25. Si se aplica el test elegido con el mutante generado, el valor introducido será el resultado de sumar 0.25 al salario. Por tanto, al realizar la comparación de ambos estados de la base de datos, el sistema determinará que son distintos en el caso particular del caso de prueba utilizado. Esto implica que dicho mutante sea matado. Esto mismo ocurre con los otros dos mutantes generados con este operador ARREP, correspondientes a los operadores aritméticos de resta y división.

5.1.1. Experimento 1B

La suite de pruebas utilizada en este experimento se puede mejorar añadiendo un test que consiga matar el mutante que ha quedado vivo en el experimento. Con este fin, se ha añadido un nuevo caso de test, por lo que la suite ahora es la siguiente:

- `UPDATE EMPLOYEES SET SALARY = 2600 WHERE ID = 44;`
- `UPDATE EMPLOYEES SET SALARY = 1900 WHERE ID = 44;`

Al aplicar la test suite anterior se obtienen los resultados mostrados en la figura 5.4. Ahora se puede observar que todos los mutantes ha resultado matados, pues el nuevo test introducido no cumple la condición de la cláusula `WHEN`, por lo que el disparador original no se ejecuta con este test, mientras que el disparador mutado generado por el operador DELWC sí. De esta forma, se detecta un estado de la base de datos distinto que mata el mutante.

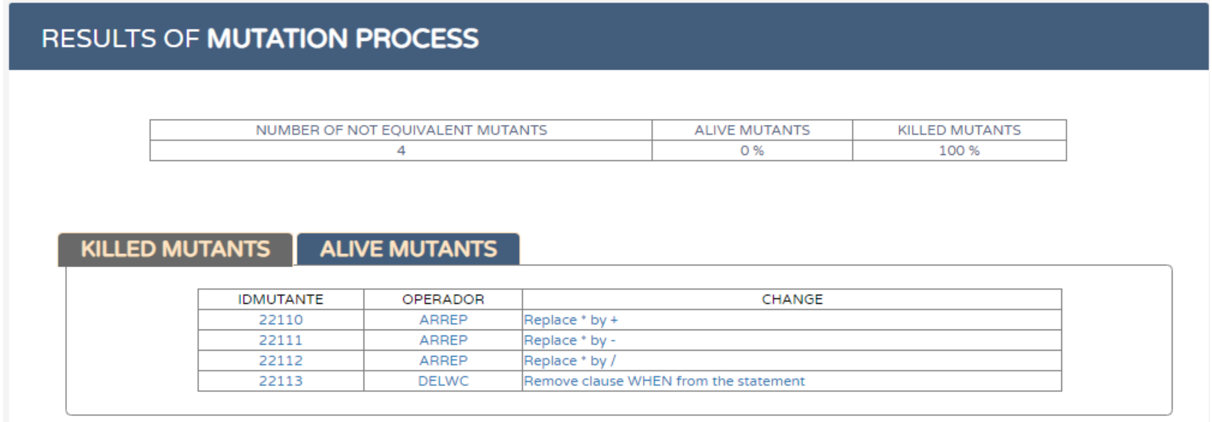


Figura 5.4: *Experimento 1. Nuevo resultado del proceso de mutación.*

5.2. Experimento 2

El siguiente experimento realizado para el estudio corresponde a un disparador algo más complejo que se muestra a continuación:

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
WHEN (new.salary > 2000 and new.salary < 2500)
EXECUTE PROCEDURE emp_tax0();

CREATE FUNCTION emp_tax0()
RETURNS trigger LANGUAGE plpgsql AS
$$
BEGIN
    insert into employee_taxes ( employee_id,taxes)
    values (NEW.id, NEW.salary*0.25);
    RETURN NEW;
END;
$$;
```

Tabla `employees`:

id	first_name	last_name	Salary
1	'Marta'	'Lopez'	1200
2	'Pepe'	'García'	2300

En este experimento se ha elegido un conjunto más amplio de operadores de mutación. Los operadores seleccionados se muestran en la figura 5.5. En la figura 5.6 se muestra la

MUTANT GENERATION

TRIGGER

emp_tax0_v2

MUTATION OPERATORS TO APPLY

VARREP - Replace NEW - OLD
ADDEC - Add event clause
DELEC - Delete event clause
REPEC - Replace event clause
ARREP - Replace arithmetic operator (+,-,*,/)
LGREP - Replace logical operator (OR - AND)
REPTIM - Replace AFTER - BEFORE
REPRLV - Replace row operator
DELWC - Remove WHEN clause from statement
NTDEL - Remove logical operator NOT
RLREP - Replace relational operator (=;<>;<;<=;>;>=)
ARREPH - Replace arithmetic operator (+,-,*,/) in header
RLREPH - Replace relational operator (=;<>;<;<=;>;>=) in header
LGREPH - Replace logical operator (OR - AND) in header
VAREPH - Replace NEW - OLD in header
NTDELH - Remove logical operator NOT in header

Figura 5.5: *Experimento 2. Selección de operadores de mutación.*

lista de mutantes generados por los operadores seleccionados.

Por último, la test suite utilizada para este experimento contiene dos test:

- `update employees set salary = salary * 1.2 where id = 1;`
- `update employees set salary = salary * 2 where id = 2;`

En la figura 5.7 se muestra el resultado de ejecutar estos tests a los mutantes generados por los operadores seleccionados.

Como se puede observar en la tabla de resultados de la figura 5.7, el número total de mutantes no equivalentes es 13, y los porcentajes de mutantes vivos y matados son 62 % y 38 %, respectivamente .

En la figura 5.8 se observan los mutantes que han quedado vivos después de realizar el proceso de mutación. La figura 5.9 muestra la tabla con los mutantes generados que han sido matados.

STEP 1 **MUTANT GENERATION**

CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW WHEN (new.salary > 2000 and new.salary < 2500) EXECUTE PROCEDURE emp_tax0();

CREATE FUNCTION emp_tax0() RETURNS trigger LANGUAGE plpgsql AS $$ BEGIN insert into employee_taxes (employee_id,taxes)values (NEW.id, NEW.salary*0.25); RETURN NEW; END; $$;

IDMUTANTE	FUNCTION/TRIGGER	OPERADOR	CHANGE
21981	FUNCTION	ARREP	Replace * by -
21982	FUNCTION	ARREP	Replace * by /
21980	FUNCTION	ARREP	Replace * by +
21993	TRIGGER	LGREPH	Replace AND BY OR
21989	TRIGGER	RLREPH	Replace < by <=
21990	TRIGGER	RLREPH	Replace < by <>
21992	TRIGGER	RLREPH	Replace < by =
21991	TRIGGER	RLREPH	Replace < by >
21988	TRIGGER	RLREPH	Replace < by >=
21994	TRIGGER	VAREPH	Replace NEW BY OLD
21995	TRIGGER	VAREPH	Replace NEW BY OLD
21978	FUNCTION	VARREP	Replace NEW BY OLD
21979	FUNCTION	VARREP	Replace NEW BY OLD

Figura 5.6: *Experimento 2. Tabla de mutantes generados.*

RESULTS OF **MUTATION PROCESS**

NUMBER OF NOT EQUIVALENT MUTANTS	ALIVE MUTANTS	KILLED MUTANTS
13	62 %	38 %

Figura 5.7: *Experimento 2. Tabla de resultados.*

Para comprender mejor el resultado de este experimento, a continuación se estudian los mutantes generados y su comportamiento con los casos de prueba. Primero se analizan los mutantes que han sido matados. En la figura 5.9 se puede ver que han sido cinco los mutantes que se han quedado matados. Estos mutantes corresponden a tres operadores distintos, todos de cabecera: RLREPH, LGREPH, VAREPH.

El primero de los mutantes matados es el mutante con ID 21988, en el que se ha aplicado el reemplazo del operador menor (<) por el operador mayor o igual (>=):

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
WHEN (new.salary > 2000 and new.salary >= 2500)
EXECUTE PROCEDURE emp_tax0();
```

ALIVE MUTANTS | KILLED MUTANTS

IDMUTANTE	OPERADOR	CHANGE	EQUIVALENT
21978	VARREP	Replace NEW BY OLD	☐
21979	VARREP	Replace NEW BY OLD	☐
21980	ARREP	Replace * by +	☐
21981	ARREP	Replace * by -	☐
21982	ARREP	Replace * by /	☐
21989	RLREPH	Replace < by <=	☐
21992	RLREPH	Replace < by =	☐
21994	VAREPH	Replace NEW BY OLD	☐

Save

Figura 5.8: *Experimento 2. Tabla de resultados de mutantes vivos.*

ALIVE MUTANTS | KILLED MUTANTS

IDMUTANTE	OPERADOR	CHANGE
21988	RLREPH	Replace < by >=
21990	RLREPH	Replace < by <>
21991	RLREPH	Replace < by >
21993	LGREPH	Replace AND BY OR
21995	VAREPH	Replace NEW BY OLD

Figura 5.9: *Experimento 2. Tabla de resultados de mutantes matados.*

Este cambio en la condición de la cláusula `WHEN` de la declaración del disparador es fundamental, pues el disparador original no llega a ejecutarse con ninguno de los tests utilizados en el experimento, pero con el cambio producido por esta mutación, la condición pasa a ser cierta en el segundo test, pues el salario del empleado 2 después de ejecutar el test es mayor a 2500. Por lo tanto, en este caso el estado de la base de datos con el disparador mutado es diferente del estado obtenido con el disparador original, por lo que el mutante es matado. Lo mismo ocurre con el mutante con ID 21990, en el que se reemplaza el operador menor (<) por el operador distinto (<>), como se muestra a continuación:

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees for each row
WHEN (new.salary > 2000 and new.salary <> 2500)
EXECUTE PROCEDURE emp_tax0();
```

El comportamiento del mutante 21991 es idéntico a los anteriores, pues en ese mutante

se realiza la sustitución del operador menor (<) por el operador (>) en la segunda condición de la cláusula WHEN.

El cuarto de los mutantes matados es el que tiene ID 21993, que corresponde a la aplicación del operador LGREPH, que reemplaza el operador logico AND por OR en la cláusula WHEN:

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
WHEN (new.salary > 2000 OR new.salary < 2500)
EXECUTE PROCEDURE emp_tax0();
```

La nueva condición del mutante hace que se cumpla con el segundo test, por lo que el disparador es lanzado y el estado de la base de datos obtenido después de ejecutar el test es distinto al del disparador original. Por lo tanto, este mutante también resulta matado.

El resultado es el mismo con el mutante 21995, generado con el operador de mutación VAREPH, que realiza la sustitución de NEW por OLD generando el siguiente disparador:

```
CREATE TRIGGER emp_taxes0 BEFORE UPDATE ON employees FOR EACH ROW
WHEN (new.salary > 2000 and old.salary < 2500)
EXECUTE PROCEDURE emp_tax0();
```

Una vez se han analizado los mutantes matados, es el momento de revisar el listado de los mutantes vivos.

En este experimento lo que ocurre en casi todos los casos de mutantes vivos es que no se llegan a ejecutar ni el disparador original ni el disparador mutado, porque no se cumplen las condiciones de la cláusula WHEN en ninguno de los dos casos. Todos los mutantes menos uno (21978 a 21992) se pueden explicar por este motivo. El último, 21994, es más interesante, pues aunque se modifica la condición de la cabecera del disparador, no se llega a disparar el disparador porque sigue sin cumplirse dicha condición con los casos de prueba utilizados. La conclusión que se puede llegar con este experimento 2 es que el conjunto de tests no permite probar correctamente el disparador y habría que modificar el contenido de las tablas y los casos de prueba para que se puedan cubrir todos los casos considerados por los operadores de mutación.

5.2.1. Experimento 2B

En el experimento anterior se ha podido comprobar que la mala calidad de los tests aplicados resulta en un gran número de mutantes vivos. Por lo tanto, se debe mejorar el conjunto de tests aplicados para conseguir matar estos mutantes. Para ello, se actualiza la test suite añadiendo un nuevo caso de test, que queda de la siguiente forma:

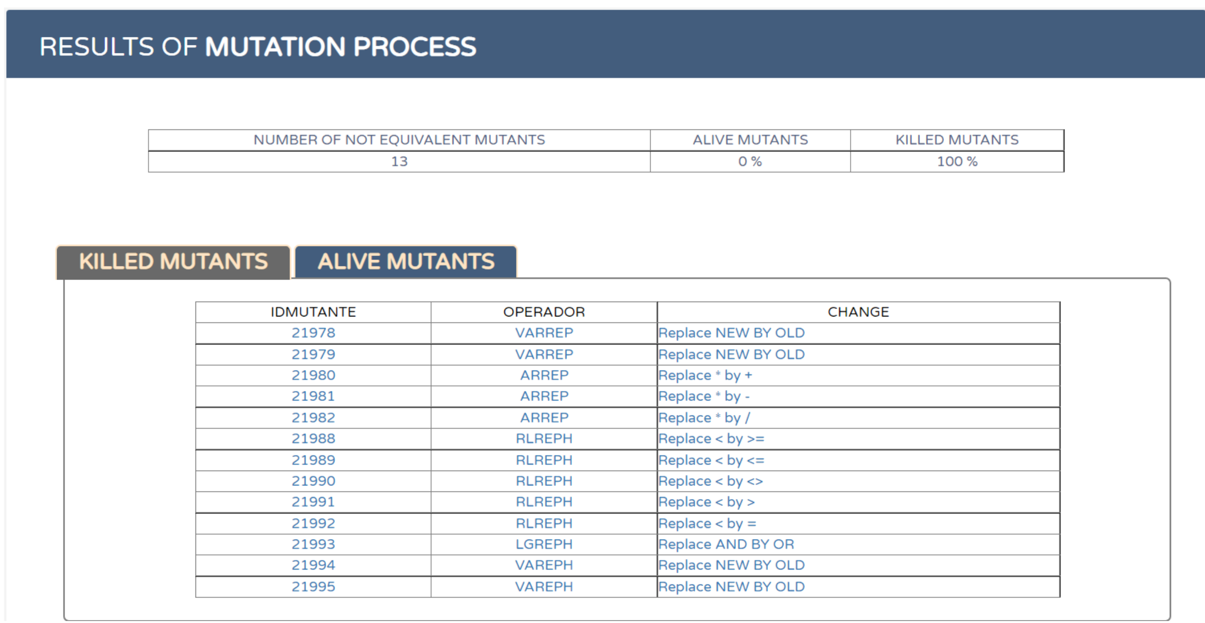
- `update employees set salary = salary * 1.2 where id = 1;`
- `update employees set salary = salary * 2 where id = 2;`
- `update employees set salary = salary * 2;`

Si se aplica esta suite al disparador del experimento anterior, se obtiene el resultado mostrado en la figura 5.10. Se puede observar que, añadiendo un nuevo caso de test, el resultado del proceso de mutación ha variado bastante. En esta ocasión se aplica una operación que duplica el salario de los empleados y además cumple con la condición que sea entre 2000 y 2500, pues el test añadido, a diferencia de los dos anteriores, modifica todas las filas de la tabla `employees`. De este modo, se ejecutan los disparadores de los mutantes y al hacer la comparativa entre el estado de la base de datos con el disparador original y el estado de la base de datos de estos mutantes, se puede determinar que son distintos, por lo que quedan matados todos los mutantes.

5.3. Experimento 3

El tercero de los experimentos que se va describir en este apartado es el siguiente:

```
CREATE FUNCTION film_actor_audit() RETURNS TRIGGER AS
$film_actor_audit$;
BEGIN
    UPDATE actor SET films=films-1 WHERE actor_id= OLD.actor_id;
    RETURN OLD;
END;
$film_actor_audit$;
LANGUAGE plpgsql;
```



RESULTS OF **MUTATION PROCESS**

NUMBER OF NOT EQUIVALENT MUTANTS	ALIVE MUTANTS	KILLED MUTANTS
13	0 %	100 %

KILLED MUTANTS | ALIVE MUTANTS

IDMUTANTE	OPERADOR	CHANGE
21978	VARREP	Replace NEW BY OLD
21979	VARREP	Replace NEW BY OLD
21980	ARREP	Replace * by +
21981	ARREP	Replace * by -
21982	ARREP	Replace * by /
21988	RLREPH	Replace < by >=
21989	RLREPH	Replace < by <=
21990	RLREPH	Replace < by <>
21991	RLREPH	Replace < by >
21992	RLREPH	Replace < by =
21993	LGREPH	Replace AND BY OR
21994	VAREPH	Replace NEW BY OLD
21995	VAREPH	Replace NEW BY OLD

Figura 5.10: *Experimento 2B. Tabla de resultados.*

```
CREATE TRIGGER film_actor_audit
AFTER DELETE ON film_actor
    FOR EACH ROW EXECUTE PROCEDURE film_actor_audit();
```

La base de datos tiene información de las tablas sobre las que afecta este trigger

Tabla `actor`:

actor_id	**first_name**	**last_name**	**films**
1	'Penelope'	'Guines'	3
2	'Nick'	'Wahlberg'	3
3	'Ed'	'Chase'	3
4	'Jennifer'	'Davis'	3

Tabla `film` (se han excluido algunas columnas de la tabla y se han abreviado los nombres de las columnas):

id	**title**	**year**	**lang.**	**rate**	**length**	**repl.cost**
1	'Academy Dinosaur'	2006	1	'0.99'	86	'20.99'
2	'Ace Goldfinger'	2006	1	'4.99'	48	'12.99'
3	'Adaptation Holes'	2006	1	'2.99'	50	'18.99'
4	'Affair Prejudice'	2006	1	'2.99'	117	'26.99'

Para este experimento, a diferencia de los anteriores, se seleccionan todos los operadores disponibles que nos ofrece la aplicación, como se muestra en la figura 5.11.

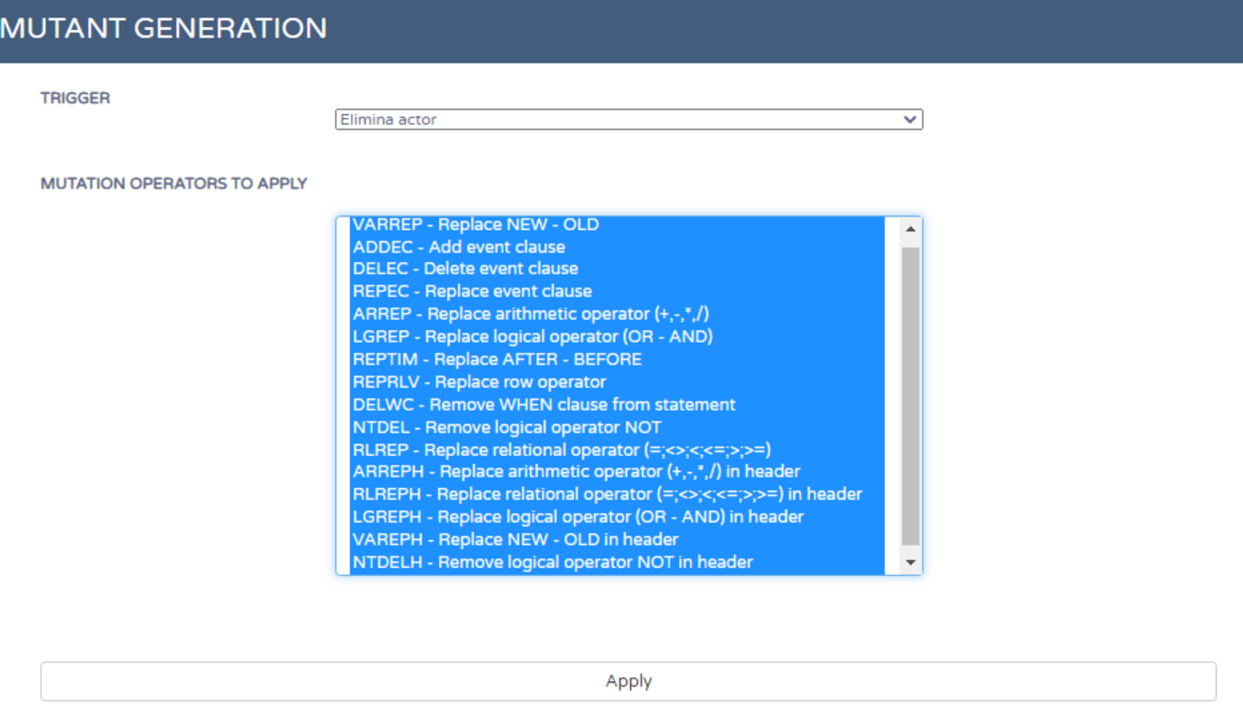


Figura 5.11: *Experimento 3. Selección de operadores de mutación.*

Una vez se eligen los operadores, con este experimento se pone a prueba la aplicación, ya que debe verificar el funcionamiento de todos los operadores disponibles en la aplicación.

La suite elegida para este tercer experimento está formada por los siguientes casos de tests:

- `insert into film_actor values (1,2);`
- `insert into film_actor values (1,20);`
- `insert into film_actor values (1,23);`
- `insert into film_actor values (2,2);`
- `insert into film_actor values (2,7);`

STEP 1 **MUTANT GENERATION**

CREATE TRIGGER film_actor_audit AFTER DELETE ON film_actor FOR EACH ROW EXECUTE PROCEDURE film_actor_audit();

CREATE FUNCTION film_actor_audit() RETURNS TRIGGER AS $film_actor_audit$ BEGIN UPDATE actor SET films= films -1 WHERE actor_id= OLD.actor_id; RETURN OLD; END; $film_actor_audit$ LANGUAGE plpgsql;

IDMUTANTE	FUNCTION/TRIGGER	OPERADOR	CHANGE
22183	TRIGGER	ADDEC	ADD INSERT clause
22184	TRIGGER	ADDEC	ADD UPDATE clause
22188	FUNCTION	ARREP	Replace - by *
22187	FUNCTION	ARREP	Replace - by +
22189	FUNCTION	ARREP	Replace - by /
22186	TRIGGER	REPEC	REPLACE clause
22185	TRIGGER	REPEC	REPLACE clause
22190	TRIGGER	REPTIM	Replace AFTER by BEFORE
22200	FUNCTION	RLREP	Replace = by <
22197	FUNCTION	RLREP	Replace = by <=
22198	FUNCTION	RLREP	Replace = by <>
22199	FUNCTION	RLREP	Replace = by >
22196	FUNCTION	RLREP	Replace = by >=

Figura 5.12: *Experimento 3. Tabla de mutantes generados.*

- `delete from film_actor where film_id=2;`

Al aplicar la suite, la herramienta procesa los casos de tests con los mutantes generados obtniendo el resultado mostrado en la figura 5.13.

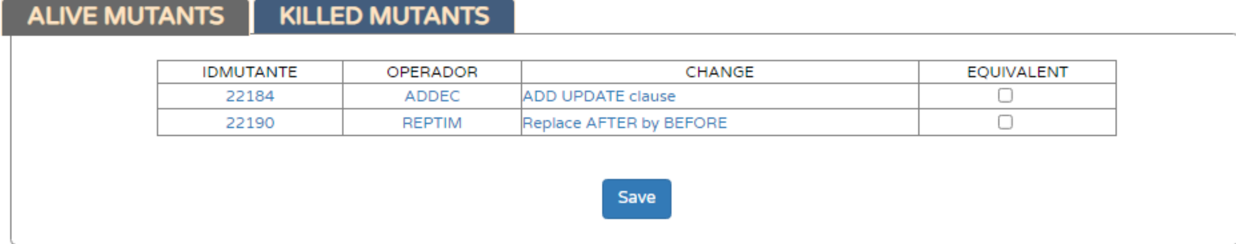
RESULTS OF **MUTATION PROCESS**

NUMBER OF NOT EQUIVALENT MUTANTS	ALIVE MUTANTS	KILLED MUTANTS
13	15 %	85 %

Figura 5.13: *Experimento 3. Tabla de resultados.*

Como se puede apreciar en la tabla de resultados, de un total de 13 mutantes hay 15 por ciento de mutantes vivos (figura 5.14), que equivalen numéricamente a 2 mutantes, y por consiguiente, el resto, los 11 restantes han quedado matados (figura 5.15).

El primero de los mutantes que han quedado vivos es el que tiene ID 22184, que ha sido generado al aplicar el operador ADDEC. Este nuevo mutante ha añadido la cláusula `UPDATE` resultando de la siguiente manera.



ALIVE MUTANTS | KILLED MUTANTS

IDMUTANTE	OPERADOR	CHANGE	EQUIVALENT
22184	ADDEC	ADD UPDATE clause	☐
22190	REPTIM	Replace AFTER by BEFORE	☐

Save

Figura 5.14: *Experimento 3. Tabla de resultados de mutantes vivos.*

ALIVE MUTANTS | KILLED MUTANTS

IDMUTANTE	OPERADOR	CHANGE
22183	ADDEC	ADD INSERT clause
22185	REPEC	REPLACE clause
22186	REPEC	REPLACE clause
22187	ARREP	Replace - by +
22188	ARREP	Replace - by *
22189	ARREP	Replace - by /
22196	RLREP	Replace = by >=
22197	RLREP	Replace = by <=
22198	RLREP	Replace = by <>
22199	RLREP	Replace = by >
22200	RLREP	Replace = by <

Figura 5.15: *Experimento 3. Tabla de resultados de mutantes matados.*

```
CREATE TRIGGER film_actor_audit
  AFTER DELETE OR UPDATE on film_actor
  FOR EACH ROW EXECUTE PROCEDURE film_actor_audit();
```

Este disparador ha quedado vivo, ya que la nueva cláusula incluida no afecta con la test suite elegida, ya que no tiene ningún caso de test con esa acción. Por lo que el nuevo mutante, al ser lanzado contra la test suite, no tendrá ninguna diferencia a cuando es lanzado nuestro disparador original con la test suite. Al no encontrar ninguna diferencia en los estados resultantes de la base de datos, el mutante queda vivo.

El otro de lo mutantes vivos que ha quedado vivo es el que tiene ID 22190. Este mutante se ha obtenido al aplicar el operador REPTIM. Este operador realiza el reemplazo de la cláusula `AFTER` por `BEFORE` dando lugar al siguiente mutante.

```
CREATE TRIGGER film_actor_audit
  BEFORE delete on film_actor
  FOR EACH ROW EXECUTE PROCEDURE film_actor_audit();
```

El motivo por el que dicho mutante ha quedado vivo es que al cambiar el momento en el que se realiza la acción de la función, en este caso de actualizar la tabla `ACTOR`, no influye si es antes o después del borrado sobre la tabla `FILM_ACTOR`. La acción se va realizar de la misma forma modificando los datos de igual manera, por lo que al realizar la comparación entre el estado de la base de datos ejecutando el disparador original y el estado de la base de datos del mutante, la información final va ser idéntica. De esta manera corroboramos que dicho mutante está vivo.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se ha definido un marco utilizando la técnica de pruebas de mutación para evaluar la calidad de los conjuntos de pruebas con los que se verifica el funcionamiento de un disparador. Se ha definido un conjunto de operadores de mutación específicos para este contexto, se ha implementado una aplicación que permite evaluar el funcionamiento de este marco y se ha comprobado su validez con diversos experimentos. La aplicación se ha implementado para el sistema de gestión de bases de datos PostgreSQL con disparadores programados utilizando el lenguaje procedural PL/PgSQL. La herramienta desarrollada permite generar automáticamente los mutantes de un disparador de acuerdo a los operadores definidos y determinar el porcentaje de mutantes matados por un conjunto de pruebas. No se ha encontrado en la literatura científica de pruebas de mutación ningún marco que permita verificar la calidad de un conjunto de pruebas para disparadores de bases de datos relacionales.

Una vez visto el alcance del trabajo, se puede decir que se han cumplido las expectativas iniciales detalladas como objetivos específicos del trabajo.

En primer lugar se ha realizado un exhaustivo estudio previo del uso existente de las pruebas de mutación aplicadas en el campo de las bases de datos. Se ha revisado la documentación existente y no se ha encontrado ningún sistema que haga pruebas de mutación

sobre disparadores de bases de datos.

El segundo de los objetivos que se ha cumplido es el de definir un conjunto de operadores de mutación para un lenguaje de programación de disparadores. En esta herramienta se han añadido 16 operadores de mutación, divididos en dos grandes grupos. Por una parte se han definido operadores de cabecera, en los que se realizan cambios en la sentencia de creación del disparador. Son los operadores ARREPH, RLREPH, LGREPH, VAREPH, NTDELH. Y en segundo lugar, se han definido los operadores de cuerpo o de función, los cuales aplican los cambios en la función asociada al disparador. Estos operadores son VARREP, ADDEC, DELEC, REPEC, ARREP, LGREP, REPTIM, REPRLV, DELWC NTDEL y RLREP. Se ha conseguido que gracias a estos operadores, se ofrezcan diferentes opciones para aplicar los cambios y conseguir variedad de disparadores mutantes que permiten probar los aspectos específicos de un lenguaje de programación de disparadores.

El tercero y uno de los grandes objetivos marcados al principio del trabajo es la creación de una herramienta que genere los mutantes de un disparador y ejecute conjuntos de pruebas sobre ellos de forma automática.

El cuarto y último de los objetivos fijados consiste en la evaluación de los resultados obtenidos con el marco desarrollado. Se ha verificado su funcionamiento en diversos disparadores con diferentes conjuntos de casos de prueba y se ha comprobado su funcionamiento razonablemente eficiente con bases de datos de ejemplo. En resumen, se ha constatado la viabilidad de este tipo de técnicas de pruebas de mutación en un contexto complejo como es el de disparadores de bases de datos.

Esta aplicación con sus ficheros fuentes están publicados bajo una licencia de código abierto y libre, a través del siguiente enlace:

- https://github.com/Milani90/mutationTesting

6.2. Trabajo futuro

Una vez que se han descrito los objetivos conseguidos en el trabajo, se ha corroborado que se pueden realizar ciertas mejoras y extensiones como trabajo futuro.

Por una parte, se puede mejorar el proceso de comparación de las bases de datos de pruebas del disparador original y del disparador mutado. En el capítulo 3 se plantearon algunas de las limitaciones del procedimiento utilizado. Se podría mejorar analizando los tipos de datos de las columnas de cada tabla y determinando aquellas columnas que pudieran depender del instante de ejecución o el orden de actualización de las filas en la tabla. Este procedimiento puede ser muy complejo, pues la base de datos de pruebas puede tener instalados otros disparadores que pueden interferir en el proceso de comparación y generación de copias de la base de datos inicial.

Otra extensión que se puede realizar sobre el sistema consiste en añadir un módulo para la gestión de operadores de mutación por parte del usuario. De esta manera, el propio usuario de la aplicación podría modificar los operadores de mutación predefinidos y añadir nuevos operadores a los existentes, en lugar de basarse únicamente en el listado facilitado en la aplicación. Esta extensión requiere que el usuario conozca en profundidad el funcionamiento de la aplicación, pues el usuario debería introducir o modificar con este módulo el código fuente necesario para implementar el nuevo operador, de forma que la aplicación pueda ejecutarlo para generar mutantes. De esta manera podría actualizarse el listado predefinido de mutantes que existe actualmente en la aplicación. Además se podrían habilitar permisos para el uso autorizado o no de estos operadores para los demás usuarios del sistema.

Otra posible mejora de la aplicación sería su adaptación como una aplicación de escritorio. Esto ayudaría a mejorar el rendimiento de la aplicación y la eficiencia de la misma con sus tiempos de respuesta. Las aplicaciones web conllevan una serie de condicionantes, por lo que una aplicación de escritorio ayudaría a mejorar el producto obtenido. A esto se le suma que las aplicaciones de escritorio por lo general son mas seguras que las aplicaciones web, y de esta manera evitaríamos en lo posible cualquier mal uso de la herramienta.

Por último, este marco de pruebas de mutación podría extenderse para tratar disparadores definidos para otros sistemas de gestión de bases de datos diferentes de PostgreSQL, como por ejemplo Oracle, MYSQL ó SQLite. Aunque los componentes principales de los lenguajes de programación de disparadores son similares en todos los sistemas de bases de datos relacionales, y de hecho existe un estándar denominado SQL/PSM, todas las implementaciones comerciales incluyen diferencias sintácticas y semánticas que los diferencian del estándar. Por ejemplo, los disparadores en Oracle no necesitan ninguna función asociada. Para implementar esta extensión se deberían estudiar las características específicas de cada RDBMS que se vaya a considerar para intentar reutilizar en lo posible el código existente para PostgreSQL.

Capítulo 7

Introduction

This chapter introduces the fundamental motivations to do this work, in addition to its objectives. It also describes the work plan that has been carried out during the development of the final Master Project and the structure of this report.

7.1. Motivation

Mutation testing (*mutation testing*[14,19]) is a white box testing technique for the design and quality assessment of software system test suites. The main objective of this technique is to determine the capacity of a set of tests for the detection of small errors that a *competent* programmer could introduce during the development or maintenance of a software system. The basic idea for evaluating test cases by mutation testing is to insert small syntactic changes into the original system, generating a large number of *erroneous* versions called mutants. Mutants are generated automatically by applying mutation operators which precisely define the changes introduced in the system. Typical examples of mutation operators are the substitution of an arithmetic operator for another or the removal of a syntactic element from a statement in the program which is being evaluated. Once a set of mutants is available, the test cases to be evaluated is applied to both the original system and each of the mutants. If the system's test suite is well designed, it should detect errors introduced into these mutants, producing different outputs with respect to the original system. In this case the mutant is said to have been *killed*. If the application of each test case to the mutant

generates exactly the same output as the original system, then it can be concluded that the test suite is not capable of detecting the error introduced in the mutant, and in this case it is said that the mutant remains *alive*. The detection capacity of the test cases is given by a metric that corresponds to the percentage of mutants that have been killed. If this values is very low, the test cases should be extended to be able to detect the errors reflected in the alive mutants. The hypothesis of this technique is that, if test cases are able to distinguish a system from its mutants, it will be suitable for detecting faults inadvertently introduced into the system.

7.2. Objectives

The mutation testing technique has been used in the last decades in many different areas of software development. Particularly, there are multiple applications of this technique in database systems, mainly aimed at detecting errors in SQL data query statements. In addition, some authors have applied this technique on data modification SQL statements. These types of applications are especially sophisticated, since determining whether a mutant has been killed requires comparing the state of the entire database. However, in the literature in this area, no application of this technique to the triggers of a database system has been found. The main objective of this work is to study the application of this technique to database triggers, establishing a mutation testing framework for triggers integrated into databases and developing a tool to evaluate its performance.

The specific objectives of this work are the following:

- Study the literature related to the use of mutation tests in the field of databases.
- Establish a set of mutation operators for a trigger design language.
- Develop a tool that implements the application of mutation operators for the generation of mutants.
- Evaluate the results obtained with the developed framework.

7.3. Workplan

The tasks that are needed to develop this work have been divided into the following phases:

- Phase 1: Analysis and review of previous works.
- Phase 2: Selection of the database management system that will be considered for the development of the work.
- Phase 3: Design of specific mutation operators.
- Phase 4: Design and implementation of the modules that make up the tool that automates the mutation testing framework.
- Phase 5: Evaluation of the proposed framework.

The supervision and monitoring of this work is carried out through weekly meetings that allow us to review the progress made, to identify problems during development, to determine the scope of the objectives and to plan guidelines to continue with the development.

7.4. Structure of the Document

This working memory is structured in six chapters.

- Chapter 1 - Introduction: Exposes the motivation for the work carried out together with the objectives set.
- Chapter 2 - Preliminaries: Exposes the fundamental concepts on which the proposed framework is based and explores the related literature. It includes the basic elements used to understand the work done.

- Chapter 3 - Proposal for Mutation Tests for Triggers. This chapter presents the proposed mutation operators for application in the proposed mutation testing framework and describes the processes for generating mutants and running tests.
- Chapter 4 - Implementation of the proposal: Describes the development process of the tool, as well as the different functionalities that it integrates.
- Chapter 5 - Evaluation: Some experiments performed for the evaluation of the proposal and analysis of the results obtained are presented.
- Chapter 6 - Conclusions and Future Work: Exposes the conclusions that can be extracted from the work carried out. Additionally, possible modifications to the tool that could improve its operation are indicated.

Capítulo 8

Conclusions and future work

8.1. Conclusions

In this work, a framework has been defined using the mutation testing technique to evaluate the quality of the test cases with which the performance of a trigger is verified. A set of specific mutation operators has been defined for this context, an application has been implemented that allows evaluating the functioning of this framework, and its validity has been verified with various experiments. The application has been implemented for the PostgreSQL database management system with programmed triggers using the PL / PgSQL procedural language. The developed tool allows to automatically generate the mutants of a trigger according to the defined operators and determine the percentage of mutants killed by a set of tests. No framework has been found in the scientific mutation testing literature to verify the quality of a set of test cases for relational database triggers.

Having seen the scope of the work, it can be said that the initial expectations detailed as specific objectives of the work have been met.

In the first place, an exhaustive previous study of the existing use of applied mutation tests in the field of databases has been carried out. Existing documentation has been reviewed and no system has been found that does mutation testing on database triggers.

The second objective that has been achieved is to define a set of mutation operators for a trigger programming language. In this tool 16 mutation operators have been added, divided

into two large groups. On the one hand, header operators have been defined, in which changes are made in the trigger creation statement. They are the operators ARREPH, RLREPH, LGREPH, VAREPH, NTDELH. And secondly, the body or function operators have been defined, which apply the changes to the function associated with the trigger. These operators are VARREP, ADDEC, DELEC, REPEC, ARREP, LGREP, REPTIM, REPRLV, DELWC NTDEL, and RLREP. Thanks to these operators, it has been possible to offer different options to apply the changes and achieve a variety of mutant triggers that allow testing the specific aspects of a trigger programming language.

The third and one of the most important objectives set at the beginning of the work is the creation of a tool that generates the mutants of a trigger and executes test cases on them automatically.

The fourth and last objective consists of evaluating the results obtained with the developed framework. It has been verified to work on various triggers with different test cases and it has been checked that it is reasonably efficient when working with sample databases. In summary, the viability of this type of mutation testing techniques has been confirmed in a complex context such as database triggers.

8.2. Future work

Once the objectives achieved in the work have been described, it has been confirmed that certain improvements and extensions can be made as future work.

On the one hand, the process of comparing the test databases of the original trigger and the mutated trigger can be improved. In Chapter 3 some of the limitations of the procedure used were highlighted. The procedure used in this work could be improved by analyzing the data types of the columns of each table and determining those columns that could depend on the moment of execution or the order of updating the rows in the table. This procedure can be very complex, as the test database may have other triggers installed that can interfere with the process of comparing and generating copies of the initial database.

Another extension that can be carried out on the system consists of adding a module for the management of mutation operators by the user. In this way, the application user himself could modify the predefined mutation operators and add new operators to the existing ones, instead of relying only on the list provided in the application. This extension requires the user to have a deep knowledge of how the application works, since the user should introduce or modify with this module the source code necessary to implement the new operator, so that the application can execute it to generate mutants. In this way, the predefined list of mutants that currently exists in the application could be updated. In addition, permissions for the authorized or unauthorized use of these operators could be enabled for other users of the system.

Another possible improvement of the application would be its adaptation as a desktop application. This would help improve the performance of the application and the efficiency of the application with its response times. Web applications involve a series of constraints, so a desktop application would help to improve the product obtained. In addition, desktop applications are generally safer than web applications, and in this way we would avoid any misuse of the tool as much as possible.

Finally, this mutation testing framework could be extended to deal with triggers defined for other database management systems other than PostgreSQL, such as Oracle, MYSQL or SQLite. Although the main components of trigger programming languages are similar in all relational database systems, and in fact there is a standard called SQL / PSM, all commercial implementations include syntactic and semantic differences that differentiate them from the standard. For example, triggers in Oracle do not need any associated functions. To implement this extension, the specific characteristics of each RDBMS to be considered should be studied to try to reuse as much as possible the existing code for PostgreSQL.

Bibliografía

[1] Css. Disponible en: https://lenguajecss.com/.

[2] Eclipse. Disponible en: https://www.eclipse.org/.

[3] Html. Disponible en: https://html.com/.

[4] Javascript. Disponible en: https://www.javascript.com/.

[5] Pgadmin. Disponible en: https://www.pgadmin.org/.

[6] Php. Disponible en: https://www.php.net/.

[7] Postgresql: The world's most advanced open source relational database.

[8] Python. Disponible en: https://www.python.org/.

[9] Visualstudio. Disponible en: https://visualstudio.microsoft.com/.

[10] Iso/iec 9075-1:1999. information technology. database languages, sql, 1999.

[11] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269. Association for Computing Machinery, July 2014.

[12] W. K. Chan, S. C. Cheung, and T. H. Tse. Fault-based testing of database application programs with conceptual data model. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 187–196, 2005.

[13] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *The VLDB Journal*, 24(6):731–755, December 2015.

[14] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[15] Anna Derezinska. An experimental case study to applying mutation analysis for sql queries. volume 4, pages 559 – 566, 11 2009.

[16] The PostgreSQL Global Development Group. PostgreSQL. Disponible en: https://www.postgresql.org/.

[17] B. P. Gupta, D. Vira, and S. Sudarshan. X-data: Generating test data for killing sql mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 876–879, 2010.

[18] Lorena Gutiérrez-Madroñal, Inmaculada Medina-Bulo, and Mercedes G. Merayo. Mutation Operators for Google Query Language. In *Intelligent Information and Database Systems*, pages 354–365. Springer Singapore, 2020.

[19] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, 1977.

[20] Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems. In *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, pages 31–40, March 2013.

[21] Ana Claudia B. Loureiro Moncao, Celso G. Camilo, Leonardo T. Queiroz, Cassio L. Rodrigues, Plinio de Sa Leitao, and Auri M.R. Vincenzi. Shrinking a database to

perform SQL mutation tests using an evolutionary algorithm. In *2013 IEEE Congress on Evolutionary Computation*, pages 2533–2539, June 2013. ISSN: 1941-0026.

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

[23] K. Pan, X. Wu, and T. Xie. Automatic test generation for mutation testing on database applications. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 111–117, 2013.

[24] Tanmoy Sarkar, Samik Basu, and Johnny S. Wong. ConSMutate: SQL Mutants for Guiding Concolic Testing of Database Applications. In *Formal Methods and Software Engineering*, pages 462–477. Springer Berlin Heidelberg, 2012.

[25] Hossain Shahriar and Sheetal Batchu. Towards mutation-based testing of column-oriented database queries. In *Proceedings of the 2014 ACM Southeast Regional Conference*, pages 1–6. Association for Computing Machinery, 2014.

[26] Hossain Shahriar and Mohammad Zulkernine. MUSIC: Mutation-based SQL Injection Vulnerability Checking. In *2008 The Eighth International Conference on Quality Software*, pages 77–86, August 2008.

[27] Javier Tuya, Ma Jose Suarez-Cabal, and Claudio de la Riva. SQLMutation: A tool to generate mutants of SQL database queries. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pages 1–5, 2006.

[28] Javier Tuya, Ma José Suárez-Cabal, and Claudio De La Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007. Publisher: Elsevier BV.

[29] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using

mutant schemata. *ACM SIGSOFT Software Engineering Notes*, 18(3):139–148, July 1993.

[30] Chris J. Wright, Gregory M. Kapfhammer, and Phil McMinn. Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 63–72, March 2013.

[31] Chixiang Zhou and Phyllis Frankl. Mutation Testing for Java Database Applications. In *2009 International Conference on Software Testing Verification and Validation*, pages 396–405, 2009. ISSN: 2159-4848.

[32] Chixiang Zhou and Phyllis Frankl. Inferential Checking for Mutants Modifying Database States. In *Verification and Validation 2011 Fourth IEEE International Conference on Software Testing*, pages 259–268, March 2011.

[33] Chixiang Zhou and Phyllis Frankl. JDAMA: Java database application mutation analyser. *Software Testing, Verification and Reliability*, 21(3):241–263, 2011.