Mutation Testing of PL/SQL Programs

Arzu Behiye Tarımcı

Turkcell Technology, Istanbul, Turkey

Hasan Sözer*

Ozyegin University, Istanbul, Turkey

Abstract

Mutation testing is a prominent technique for evaluating the effectiveness of a test suite. Existing tools developed for supporting this technique are applicable for mainstream programming languages like C and Java. Mutation testing tools and mutation operators used by these tools are inherently language-specific. Moreover, there is a lack of industrial case studies for evaluating mutation testing tools and techniques in practice. In this article, we introduce muPLSQL, a tool for applying mutation testing on PL/SQL programs, facilitating automation for both mutant generation and test execution. We utilized existing mutation operators that are applicable for PL/SQL. In addition, we introduced some operators specifically for this language. We conducted an industrial case study for evaluating the applicability and usefulness of our tool and mutation testing in general. We applied mutation testing on a business support software system. muPLSQL generated a total of 5,939 mutants. The number of live mutants was 680. Manual inspection of live mutants led to improvements of the existing test suite. In addition, we found 8 faults in source code during the inspection process. Test execution against the mutants required around 40 hours. The overall effort was almost one person month.

Keywords: software testing, mutation testing, mutation analysis, PL/SQL,

*Corresponding author

Email addresses: `arzu.tarimci@turkcell.com.tr` (Arzu Behiye Tarımcı), `hasan.sozer@ozyegin.edu.tr` (Hasan Sözer)

1. Introduction

Mutation Testing is a prominent technique for evaluating the adequacy of test suites and guiding the improvement of test cases [1]. There have been many tools developed for supporting this technique [2]; however, they are applicable for programs that are developed with mainstream programming languages like C [3, 4] and Java [5, 6]. A critical element of these tools is the list of mutation operators, which define the way that source code is modified and mutants are generated. These mutation operators are inherently language-specific. PL/SQL (Procedural Language/Structured Query Language) [7, 8] is a dynamic programming language adopted in the industry [9]. In state-of-the-practice, a significant portion of enterprise applications are implemented with PL/SQL that works on a Oracle[1] database management system [10, 11, 12]. However, this language has acquired limited attention in the research community. To the best of our knowledge, there have been no mutation operators and tools proposed for performing mutation testing and analysis on PL/SQL programs.

Mutation testing has known to be an expensive technique due to its scalability issues and extensive effort [13] required in analyzing live mutants, i.e., those mutants that are not detected (killed) by any of the test cases created for the System Under Test (SUT). The associated costs prohibit its applicability to real-life systems. Therefore, empirical evidence regarding the usefulness of mutation testing is rare [13]. There is a lack of industrial case studies for evaluating mutation testing tools and techniques in practice. Existing evaluations are mostly in the form of controlled experiments, which are out of an industrial context and based on small-scale programs that comprise a few hundred lines of code [2]. There are only a few industrial case studies [13, 14, 15] even on safety-critical systems, for which the reliability expectations are very high and

[1] https://www.oracle.com/

as such, high testing costs are acceptable.

In this work, we introduce muPLSQL, a tool for mutation testing of PL/SQL programs. The tool facilitates automation for both mutant generation and test execution. We reviewed mutation operators that were previously proposed for various programming languages. We implemented 44 operators that are applicable for PL/SQL. 17 of these are generic operators that were proposed for procedural languages [5, 16, 17]. 21 of the remaining ones were proposed for SQL [18]. In addition, we introduced 6 operators specifically for PL/SQL. These are all implemented as part of muPLSQL, which is an open source tool[2]. We designed this tool to be extensible for incorporating new mutation operators.

We conducted an industrial case study for evaluating the applicability and usefulness of our tool and mutation testing in general. We employed muPLSQL for mutation testing and analysis of a business support software system. We used 19 objects of this system in our study. They comprise 8,206 lines of PL/SQL code in total. muPLSQL generated a total of 5,939 mutants. The number of live mutants was 680. 320 mutants were generated by PL/SQL-specific mutation operators, 46 of which survived the test execution. Manual inspection of live mutants revealed over 112 missing test scenarios and data verification that should be incorporated to the existing test suite. In addition, we found 8 faults in source code during the inspection process. Test execution against the mutants required around 40 hours of computing time. The overall effort was almost one person month.

The contributions of this paper are threefold:

- We introduce new mutation operators specifically defined for PL/SQL programs based on the PL/SQL syntax;
- We introduce a new mutation testing tool that implements mutation operators defined for PL/SQL as well as previously proposed operators that are applicable for PL/SQL;

[2] `https://github.com/arzutr/MuPLSQL`

- We present an industrial case study for evaluating the effectiveness of mutation testing in the improvement of test cases and quality of a business support software system.

The remainder of this paper is organized as follows. In the following section, we provide background information on PL/SQL and mutation testing. In Section 3, we summarize related studies. In Section 4, we explain muPLSQL, the overall process and the implemented mutation operators. In Section 5, we present our industrial case study and the experimental setup. We present and discuss the results in Section 6. Finally, we discuss future work directions and conclude the paper in Section 7.

2. Background

In this section, we first provide background information on the PL/SQL language. Then, we briefly explain the mutation testing and analysis process. In the next section, we summarize related studies and practical applications of mutation testing as reported in the literature.

2.1. PL/SQL Programs

PL/SQL is a dynamic programming language that was first introduced with Oracle version 6 [19] to overcome some limitations of SQL and to incorporate procedural extensions [7, 8]. These extensions make it possible to intermix SQL statements with imperative code.

PL/SQL programs comprise elements such as datatypes, variables, functions and procedures. These are all deployed on an Oracle database. They can be referenced by any application connected to this database. Packages can be used for grouping logically related elements. They are defined as database schema objects. There can also be stand-alone procedures and functions that do not belong to any package. PL/SQL has evolved with the advent of full object-oriented programming capabilities delivered after Oracle 9i. So, it is now both a procedural and object-oriented programming language [20].

Procedures and functions are defined in the form of PL/SQL blocks [21]. Procedures are basically functions that do not return any value. A sample procedure block is provided in Listing 1, which consists of two main parts.

Listing 1: A sample PL/SQL procedure [10].

```
1  PROCEDURE P(id IN NUMBER) IS
2  sales NUMBER;
3  total NUMBER;
4  ratio NUMBER;
5  BEGIN
6  SELECT x,y INTO sales,total
7    FROM result WHERE result_id = id;
8  ratio := sales/total;
9  IF ratio > 10 THEN
10   INSERT INTO comp VALUES (id,ratio);
11 END IF;
12 COMMIT;
13
14 EXCEPTION
15 WHEN ZERO_DIVIDE THEN
16   INSERT INTO comp VALUES (id,0);
17   COMMIT;
18 WHEN OTHERS THEN
19   ROLLBACK;
20 END;
```

Hereby, the first part (Lines 1-4) is declarative. It defines the parameters and variables used by the procedure. The second part (Lines 5-20) is the executable part that starts and ends with `BEGIN` and `END` keywords, respectively. Optionally, this part can comprise exception handling (starting with the `EXCEPTION` keyword at Line 14) for handling error conditions. In this sample procedure, we can see that results of a `SELECT` query (Lines 6-7) are used in an expression (Line 8) and an `INSERT` query is possibly triggered within an `IF` statement block (Lines 9-11).

2.2. Mutation Testing

Mutation Testing is a technique for evaluating the adequacy of test suites and guiding the improvement of test cases by identifying weaknesses [1]. It is a fault-based testing technique, where artificial faults are injected to SUT for creating faulty versions of it. Each version is called a *mutant* and it contains a different fault from the other versions. A mutant is generated by transforming the original program to a faulty version with a small syntactic change. For instance, an arithmetic operator in an expression can be replaced with another operator. These syntactic changes are defined as transformation rules called *mutation operators*.

Some of the generated mutants might not be compiled due to syntax errors. These are called *stillborn* mutants. Each of the remaining mutants are executed with each of the existing test cases. A mutant is said to be *killed* if at least one of the test cases fails and as such the mutant is detected. Otherwise, if all the test cases pass, the mutant is categorized as a *live* mutant. Live mutants should be investigated to figure out why the existing test suite was unable to detect the injected fault. This investigation can reveal a need for improvement of the test suite.

Each mutant comprises an artificially introduced simple fault and as such, it is very close to the correct version of the program. Mutation testing is based on the assumption that these simple faults constitute a representative sample of real faults introduced by developers. This assumption is based on two hypotheses, namely the *Competent Programmer Hypothesis (CPH)* [22] and *Coupling Effect Hypothesis (CEH)* [23]. CPH states that developers tend to make simple mistakes and create programs that are almost correct. CEH states that complex faults emerge as a combination of simple faults. Hence, a test suite that is capable of detecting simple faults should also be able to detect complex faults [24]. There is also empirical evidence that validates this expectation [25]. The fault detection ability of a test suite can be measured with *mutation score*, which is basically the ratio between the number of killed mutants and the number of all mutants [26]. One can simulate any test adequacy criteria by carefully choosing the mutation operators and where they are applied in source code [1].

Each live mutant must be analyzed and a live mutant might not always suggest an improvement in the test suite. A mutation operator can lead to a so called *equivalent* mutant, of which the behavior is the same as the original program. For instance, the expression `(x * 2)` will produce the same output if we replace the arithmetic operator with `+` and if `x` has the constant value of 2. There might also be so-called *duplicate* mutants. These are the mutants that are equivalent to each other although they are different from the original program. For example, expressions `(x <= y)` and `(x < y + 1)`, which are transformed from the original expression `(x < y)`, evaluate to the same result. Duplicate mutants inflate the mutation score but do not contribute to the improvement of test cases [13]. It is an effort consuming task to review all the live mutants for filtering out the equivalent and duplicate ones [27]. Although there are some techniques proposed for providing tool support [28, 29], it still remains to be, by and large, a manual process. In general, equivalent mutant detection is an undecidable problem [30].

3. Related Work

Mutation testing has been studied for almost half a century [2]. The very first studies were based on the Fortran language [22, 31]. These were followed by applications to other programming languages. Since then, there have been many mutation testing tools and applications proposed with a particular concentration on those that focus on C [3] and Java [32] languages. More than half of all the studies published in the literature so far focus on Fortran, C and Java languages [2]. To the best of our knowledge, there have been no mutation operators and tools proposed for performing automated mutation testing on PL/SQL programs. PL/SQL combines features of SQL with features of imperative languages such as C. Therefore, some of the generic mutation operators for imperative languages [33, 34] and those proposed/implemented for SQL [35, 18, 36] are applicable for PL/SQL. We employ some of these in our work as-is. However, PL/SQL has some unique language features as well. In this work, we adjusted some of the operators for covering these features. We also introduce muPLSQL that comprises the necessary tools for automating the mutation testing process. Availability of effective tool support [37] and automated frameworks is an important factor for the successful application of mutation testing [2]. muPLSQL does not only automate the generation of mutants but also the deployment and testing of these mutants on an Oracle database. This process involves tight coupling and coordination with the database management system. As such, automated testing of PL/SQL mutant objects is more challenging than testing mutants that can work as stand-alone programs. There exist a few tools that facilitate mutation testing on database applications; however these tools focus on applications developed with Java [36] and C# [38] languages. These applications connect to external databases and execute SQL queries. They do not work as an integral part of the database management system as PL/SQL programs do. There exist a few tools for unit testing PL/SQL programs [39]; however, they do not support mutation testing.

Although there exists an extensive literature on mutation testing [2], the number of reported industrial case studies is not high. The overall cost of the technique constitutes a barrier for wide industrial adoption. Despite the tool support for mutant generation and test execution, manual effort is necessary for analyzing live mutants and pinpointing improvements for test cases. Moreover, the required computation time for automated tasks (i.e., mutant generation and test execution) can be extremely high as well. In a case study, the computation time was calculated as approximately half a year, if these tasks are to be completed on a single standard desktop computer [13]. This problem can be addressed by parallel execution of automated tasks and the utilization of cloud services [40]. Even then, the overall computation time was found to be hindering for agile development processes [40]. The initial investment for the adoption of mutation testing is also subject to extensive efforts. Despite full automation, tool configuration [41] and occasional manual intervention become necessary [13] when the generated test drivers or stubs do not compile together. In our case study, we needed to invest 58 hours of manual effort just for migrating existing unit tests to inject hooks and make them compatible with the requirements of our tool. This process also requires intense industry collaboration, expertise in the domain and SUT. These challenges make it hard to perform industrial case studies as summarized below.

Ramler et al. [13] investigated the applicability and usefulness of mutation testing with a case study on a safety-critical embedded software control system. This is a large-scale system that comprises 60,000 lines of C code. On one hand, mutation testing was proven to be useful for improving the quality of a test suite that already achieves 100% MC/DC coverage. Test cases were refactored and extended based on the feedback obtained by analyzing live mutants. On the other hand, the overall process was reported as extremely costly. They found that the computation time exceeds the computing resources commonly available for testing. They also noted that the number of mutation results is beyond the resource capacity of engineers for manual analysis. Particularly, 27,158 live mutants were reported. 200 of these were sampled and reviewed in the case study.

Previously, Baker and Habli [14] conducted another case study on two safety-critical airborne software systems developed with C and Ada languages. These systems have already satisfied the necessary coverage requirements for certification. Yet, several test inadequacies were identified as a result of the manual review of 831 live mutants. They analyzed 22 code samples with lines of code ranging between 3 and 46. The same set of authors contributed to another, recently published [15] case study in the context of safety-critical systems. The study was applied on 15 selected functions of a real nuclear software system with lines of code ranging between 10 and 63. They conclude that mutation testing can potentially improve fault detection. They also find mutation testing affordable in a nuclear industry context.

The majority of the published industrial case studies focus on safety-critical systems [13, 14, 15]. This is understandable since the reliability expectations are very high for these systems and high testing costs are acceptable. There exist applications in other domains as well. However, these are not necessarily based on large-scale systems. For instance, a case study on industrial Ruby projects were reported where the experimental objects comprise a couple of hundreds of lines of code [40]. To the best of our knowledge, there have been no industrial case studies conducted for evaluating the effectiveness of mutation testing in the context of enterprise applications in general and on PL/SQL programs in particular.

Petrović and Ivanković [42] proposed a scalable mutation analysis framework integrated with the code review process at Google. The framework supports 7 programming languages: Java, C++, Python, Go, JavaScript, TypeScript and Common Lisp. It implements 5 generic mutation operators like operator replacement and statement block removal. We implemented 44 operators as part of muPLSQL. 17 of these are generic mutation operators that were proposed for procedural languages [5, 16]. 21 operators were proposed for SQL [18]. 6 of them are inspired from existing operators but adapted for PL/SQL. One of them is newly introduced.

4. muPLSQL and the Overall Process

In this section, we first explain the overall process and the muPLSQL tool. Then, we discuss the proposed, implemented as well as excluded set of mutation operators in the following subsection.

Figure 1 depicts the overall process, which is divided into 3 steps: *i)* Mutant generation, *ii)* Test case execution, and *iii)* Result analysis. The first two steps are automated, whereas the last one is manual. In the first step, mutants are generated based on the original source code. In the second step, these mutants are deployed to the database and executed against all the test cases. An input configuration specifies where mutants are generated, stored and deployed. In the last step, live mutants are reviewed to eliminate duplicate and equivalent ones. The remaining mutants are further analyzed together with the test cases and the source code to identify points to improve.

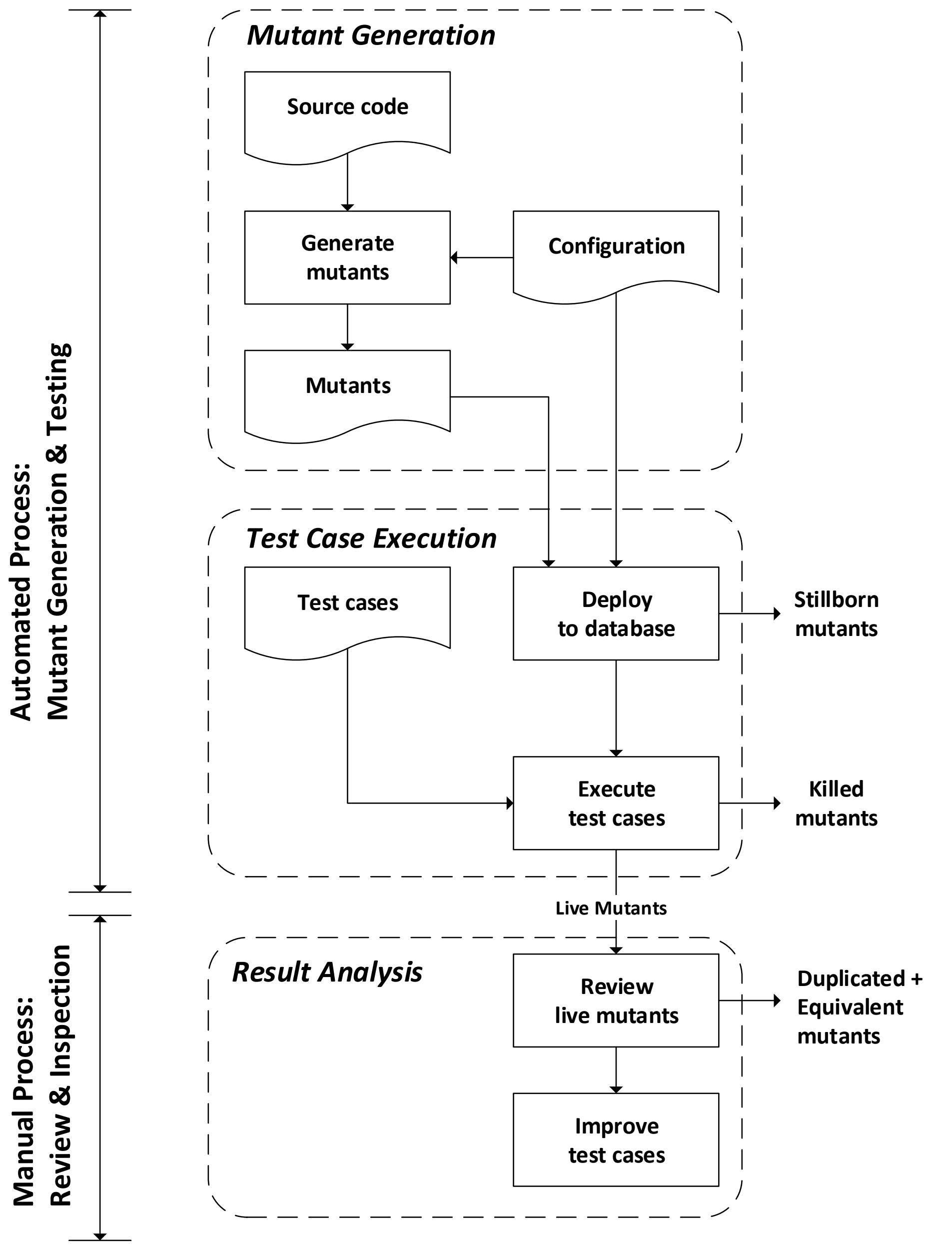

Figure 1: The overall process.

We developed the muPLSQL tool to automate the first two steps of the process. It is implemented with Oracle Java 8. muPLSQL is composed of two modules: PL/SQL Mutant Generator (PLSQL-MG) and PL/SQL Mutant Tester (PLSQL-MT). PLSQL-MG automates the first step and generates mutant PL/SQL programs (i.e., packages, package specifications, procedures and functions). It employs an open source parser[3] for parsing PL/SQL source code. Mutants are created for each procedure according to the defined mutation operators. Each mutant is saved as a separate source file. PLSQL-MG keeps information regarding each generated mutant on a local database to be able to track it throughout the process. It is designed to be extensible so that new mutation operators can be easily incorporated. It can also be used as a stand-alone Java library employable by other applications.

Mutants generated by PLSQL-MG are provided to PLSQL-MT as input. PLSQL-MT automates the second step of the process and it is developed with both Java and PL/SQL. PLSQL-MT reads mutants for testing from the database, deploys each mutant to the database and executes all the test cases on it. Test cases of PL/SQL objects are also kept in the database. PLSQL-MT reads and modifies these test cases dynamically and wraps them with additional PL/SQL code automatically to keep track of the test results for each mutant. Results are stored in the database for each mutant after test case execution.

Listing 2 shows a sample test case that is generated and executed by PLSQL-MT. The original test case that is stored in the database is listed between Line 6 and Line 19. This test case tests the procedure listed in Listing 1. It checks if any insert operation took place on table *comp* with a ratio having the value of 10. The other lines in Listing 2 are added by PLSQL-MT to keep the result per mutant. In our current implementation, test execution is time consuming particularly because mutant deployment and test execution tasks are performed sequentially, one mutant at a time, in a single thread. This is a deliberate design choice since the replication of database is subject to significant hardware costs.

[3] `https://github.com/raverkamp/plsql-parser`

Listing 2: A sample test case for the procedure listed in Listing 1 and additional PL/SQL code generated by PLSQL-MT to save the test results for each mutant.

```
DECLARE
rresult VARCHAR2(500);
pion_num NUMBER;
pios_result NUMBER:=0;
BEGIN
pion_num:=10;

p(pion_num);

SELECT ratio
INTO pios_result
FROM comp
where id :=pion_num;

IF pios_result > 0 THEN
 rresult := 'OK';
ELSE IF pios_result == 0 THEN
 rresult := 'NOK';
END IF;

saveResult(rresult, mutant_id);

COMMIT;

EXCEPTION
WHEN OTHERS THEN
rresult := SUBSTR(SQLERRMSG,1,4000);
saveResult(rresult, mutant_id);
COMMIT;
END;
```

The muPLSQL is open-source and available for experimental and educational use at a Github repository[4]. Further details regarding the features and the usage of the tool are explained as part of the documentation available at this repository. In the following subsection, we explain mutation operators that are currently implemented by the tool.

4.1. Mutation Operators

Currently, the total number of implemented mutation operators is 44. We grouped mutation operators for PL/SQL into 3 categories according to origin of mutation operator: *i)* Generic Mutation Operators (GMO) as listed in Table 1, *ii)* SQL Mutation Operators (SMO) as listed in Table 2, and *iii)* PL/SQL Mutation Operators (PMO) as listed in Table 3. PL/SQL is procedural language extension for SQL. PL/SQL allows to combine SQL statements with procedural structures. We reviewed existing mutation operators previously proposed for Fortran, Java, C and SQL [35, 34, 31, 5, 16, 18, 17]. We selected the ones that are applicable for PL/SQL. 17 of these are in the GMO category. These are the operators that were previously proposed for Fortran, Java and C languages.

21 operators are suitable for PL/SQL in the SMO category. These were previously proposed for SQL. We also adjusted some operators from existing ones or we propose new operators to cover some unique language features of PL/SQL. There are 6 such operators, taking place in the PMO category. The first and the second columns of Tables 1, 2 and 3 list the names and the codes (acronyms) of the mutation operators. The last column provides an example code snippet before and after the operator is applied.

AOR and LCR are two operators among GMO, which were used in many studies [5, 32, 16]. They also take place among the top 9 popular mutation operators that are known to be effective: RSR, GLR, SCR, CRP, LCR, ROR, ABS, SVR, AOR [43, 1]. We implemented all of these operators except UOI, SAN, DSA, DER and SDL. UOI [1] stands for *Unary Operator Insertion*. This

[4] `https://github.com/arzutr/muPLSQL`

Table 1: Generic Mutation Operators (GMO).

Operator Name	**Code**	**Example Application**
Absolute Value Insertion	ABS	i:= i+x; i:= i+ABS(x);
Arithmetic Operator Replacement (Rpl.)	AOR	i:= i+1; i:= i*1;
Constant Rpl.	CRP	i := 'Value'; i:= 'NO';
Scalar Constant Rpl.	SCR	i:=5; i:= 100000000000000000000;
GOTO Statement Rpl.	GLR	GOTO flagpoint; –GOTO flagpoint;
Logical Connector Rpl.	LCR	if v is not null and ... then if v is not null or ... then
Relational Operator Rpl.	ROR	if i = 5 then if i != 5 then
RETURN Statement Rpl.	RSR	return 'SUCCESS'; return null;
Type Change Rpl.	PCC	v NUMBER; v INTEGER;
Constant for Scalar Rpl.	CSR	i:= i+1; i:= 5 + 1;
Scalar Variable Rpl.	SVR	i := j; i:= null;
Array Reference for Array Reference Rpl.	AAR	a[1] :=4; a[2]:=4;
Constant for Array Reference Rpl.	CAR	x:=a[1]; x := v_constant_value;
Array Reference for Constant Rpl.	ACR	x:=v_constant; x :=a[1];
Array Reference for Scalar Variable Rpl.	ASR	x:=v_scalar_variable; x :=a[1];
Comparable Array Name Rpl.	CNR	x:=a[1]; x :=b[1];
Scalar Variable for Array Reference Rpl.	SAR	x:=a[i]; x :=v_scalar_variable;

operator could not be implemented in GMO category because unary arithmetic operators are not applicable for the procedural language characteristics of the PL/SQL language. However, it is relevant for SQL statements and as such, for the SMO category. *Bomb Statement Replacement* (BSR) [16] operator was ommitted since it generates an excessive amount of mutants that overwhelm resources. muPLSQL does not support SRC because CRP and CSR already cover this operator. GLR is included since mutation operators for GOTO statements are applicable for PL/SQL programs. PCC is also included. This operator was used in the study of Ma and Offutt [44]. muPLSQL implements it for available data types. There are also some operators implemented by muPLSQL although they did not generate any mutants in our case study. For instance, AAR, CAR, ACR, ASR, CNR and SAR are mutation operators that are related to the use of array constructs [34]. However, arrays are not used in our experimental objects. Likewise, GLR and CRP were not applicable for the tested PL/SQL objects in our study. Nevertheless, we included them both as part of muPLSQL and Table 1 for completeness and general applicability.

Mutation operators in the SMO category were previously applied to SQL statements in the study of Tuya et al. [35, 18]. Hereby, ROR, AGR, ABS, AOR and LCR are actually the same as ROR, AGR, ABS, AOR and LCR in GMO category but just for SQL clauses, respectively. In PL/SQL programs, the only difference is that they are applied just only within WHERE clauses of SQL SELECT statements. AGR, NLF, UNI, SEL, JOI, SUB, LIKE, NLS, NLI, NLO and ORD are all the mutation operators that were previously proposed for SQL [35, 18, 36]. muPLSQL supports these operators. The only excluded operator is GRU [35, 18, 45] that lead to stillborn mutants for PL/SQL. Oracle database compiles deployed PL/SQL objects to prepare to be ready to be run. It raises a deployment compile error in case of the use of incompatible types and SQL syntax errors. This makes all the generated mutants left stillborn. Therefore, we excluded all operators that change function specifications and variable types, making PL/SQL objects invalid.

Finally, mutation operators in the PMO category are defined and/or implemented specifically for PL/SQL programs. We defined the PMO operators by reviewing existing mutation operators together with the PL/SQL language reference [46]. Those operators that are directly applicable for PL/SQL are adopted as GMO and SMO operators. PMO operators are mainly defined by

Table 2: SQL Mutation Operators (SMO).

Operator Name	**Code**	**Example Application**
SELECT Clause	SEL	select DISTINCT OBJECT_ID from all_objects select OBJECT_ID from all_objects
JOIN Clause	JOI	RIGHT JOIN LEFT JOIN
Subquery Predicates	SUB	EXISTS NOT EXISTS
Aggregate Function Repl.	AGR	select MAX(OBJECT_ID) from all_objects select AVG(OBJECT_ID) from all_objects
Union Repl.	UNI	select ... where object_name is not null union select ... where object_name like ‘%SYS%’
Ordering Repl.	ORD	select * from table name order by asc select * from table name order by desc
Relational Operator Repl.	ROR	select ... where column_name = ‘’ select ... where column_name != ‘’
Logical Connector Repl.	LCR	select ... where ... and object_id =20 select ... where ... or object_id =20
Unary Operation Insertion	UOI	select ... where ... and object_id =20 select ... where ... and object_id =20-5
Absolute Value Insertion	ABS	select ... where ... and object_id =20 select ... where ... and object_id =-20
Arithmetic Operator Repl.	AOR	select ... where ... and o_id <i+5 and o_id >i-5 select ... where ... and o_id <i-5 and o_id >i-5
Between Predicate	BTW	select ... where ... and o_id between 5 and 10 select ... where ... and o_id not between 5 and 10
Like Predicate	LIKE	select ... where object_name like ’%SYS%’ select ... where object_name NOT like ’%SYS%’
Null Predicate Repl.	NLF	select ... where object_name is null select ... where object_name is not null
Null in Select List	NLS	select null as status,object_name from all_objects select ” as status,object_name from all_objects
Include Nulls Repl.	NLI	select ... where object_name = ’SYS’ select ... where object_name is null
Other Nulls Repl.	NLO	select ... where ... and call_count = null select ... where ... and call_count = 1
Column Repl.	IRC	select OBJECT_ID from all_objects where ... select COLUMN_SIZE from all_objects where ...
Constant Repl.	IRT	select ... where object_name = ’SYS’ select ... where object_name = ’SYS2’
Parameter Repl.	IRP	select record_date from all_objects select sysdate - 5 as record_date from all_objects
Hidden Column Repl.	IRH	select OBJECT_ID from all_objects where ... select ROW_ID from all_objects where ...

Table 3: PL/SQL Mutation Operators (PMO).

Operator Name	Code	Example Application
Rollback Removal	RBC	ROLLBACK; –ROLLBACK;
Oracle Function Repl.	OFR	NVL(column_name,‘’) SUBSTR(column_name,1)
Commit Removal	CMR	COMMIT; –COMMIT;
Exception Insertion	EXI	WHEN OTHERS THEN WHEN TOO_MANY_ROWS THEN
Query Error Insertion	QER	exec. immediate ‘truncate table t_name’ exec. immediate ‘and truncate table t_name’
Oracle Sequence Nextval Repl.	OSR	select m_sequence.nextval into i from dual select m_sequence.currval into i from dual

modifying the other existing operators to make them applicable for PL/SQL. They are defined to work on PL/SQL-specific functions and statements. In addition, we checked the coverage of the language reference and introduced a new operator to covers a syntactic feature that does not exist in other languages. *Oracle Function Replacement* (OFR) operator replaces a call to a function with a call to another function. The mutant program can be successfully compiled but the called function behaves differently at runtime. OFR uses a list of compatible PL/SQL built-in functions to perform the replacement. This list is extensible. RBC and CMR remove statements from the source code. A mutation operator that removes statements arbitrarily can generate an excessive amount of mutants, many of which are stillborn due to compilation errors [15]. Therefore, RBC and CMR implement such an operator for specific statements only; ROLLBACK and COMMIT statements in PL/SQL, respectively. Removal of these statements can significantly change transaction management and the behavior of programs that employ distributed databases. A double hyphen (–) transform the rest of the line into a comment. *Exception Insertion* (EXI) forces the program to throw a particular type of exception. *Query Error Insertion* (QER) alters SQL queries that are used as part of the EXECUTE IMMEDIATE statements. These statements are used for executing dynamic SQL statements or anonymous PL/SQL blocks. The alteration of the query does not lead to a compile error but the execution of the altered query leads to an error. All the previous operators in the PMO category are inspired from existing operators and they are defined by adjusting these for PL/SQL. In addition, we introduced a new operator named *Oracle Sequence Nextval Replacement* (OSR) specifically for PL/SQL programs. This operator replaces NEXTVAL pseudocolumn with CURRVAL. These iteration operators are used for reading a sequence of values from the database. NEXTVAL returns the item that proceeds the current one, whereas CURRVAL returns the current[5].

In the next section, we present an evaluation of these mutation operators and muPLSQL.

5. Industrial Case Study

We conducted a case study for mutation testing and analysis of a business support software system implemented with the PL/SQL language. In the following subsection, we first introduce our research questions. Then we explain our SUT and the experimental setup.

5.1. Research Questions

We aim at answering the following three research questions (RQs) in our case study:

- **RQ1:** How effective is mutation testing with muPLSQL in revealing test inadequacies for PL/SQL programs?
- **RQ2:** What are the costs of applying mutation testing for PL/SQL programs in terms of both computing resources and manual effort?
- **RQ3:** How does the effectiveness of mutants generated by PLSQL-specific operators compare to those that are generated with other (generic and SQL-specific) mutation operators?

[5]https://docs.oracle.com/cd/A84870_01/doc/server.816/a76989/ch26.htm

Our first research question, *RQ1* is on the usefulness of mutation testing and our tool for improving test suites of PL/SQL programs. Usefulness is measured in terms of the number of points to improve identified in tests as a result of the overall process. The second research question, *RQ2* aims at investigating the costs that have to be paid for possible benefits investigated with *RQ1*. Hereby, there are two types of costs that have to be measured. The first one is about the computational resources required for mutant generation and testing. We need to worry about this aspect as it was previously found that the computation time can exceed the commonly available resources [13]. The second one is about the manual effort that is required for using muPLSQL (adaptation of test cases and configuration) and analyzing live mutants one by one to identify test inadequacies. The last research question, *RQ3* is about the effectiveness of mutation operators implemented by muPLSQL. In particular, we are interested in the effectiveness of PLSQL-specific operators that we introduced in this study. We would like to compare their effectiveness with respect to the other (generic and SQL-specific) mutation operators that we employed.

In the following subsection, we explain the SUT and our experimental setup used in our case stuy.

5.2. Experimental Object and Setup

SUT is a real-world industrial OSS/BSS software system [47] that supports more than a hundred thousand transactions per day. OSS (Operations Support System) supports daily operations of a service provider. BSS (Business Support System) implements billing and customer management, network management as well as service operations like service provisioning and management [47]. The system has been maintained for 20 years by Turkcell[6], which is the largest mobile operator in Turkey. It comprises 56,323 lines of PL/SQL code in total. It is also tightly coupled with Oracle database objects, including 50 tables, 121 data types and 11 views. However, part of this legacy system is now obsolete, for

[6] http://www.turkcell.com.tr

which unit tests are not developed or no longer maintained. In our study, we focused on the actively maintained part, which includes 19 PL/SQL objects. 13 of these are stand-alone functions and procedures. The remaining 6 objects are packages. There are 125 test cases developed for these 19 objects, which contain 8,206 lines of code.

SUT is not a safety-critical system, i.e., its failure is not expected to cause a physical hazard for human life or the environment. Hence, it is not subject to safety standards, formal regulations and rigorous certification tests [48]. However, it is a business support system [47] from the telecommunications domain. That is, its failure might significantly interrupt business operations. Therefore, the testing process is audited by an independent quality assurance team in the company. Each PL/SQL object has a specification regarding the set of scenarios and input parameters supported by the object. The set of test cases are reviewed for ensuring the coverage of this specification after each introduction of a new object or modification of an existing one.

In our study, we used a desktop computer with 3.0 GHz CPU and 16 GB RAM for mutant generation. This computer has Windows 10 64-bit operating system, Java Development Kit (JDK 1.8) and Oracle 11g client programs installed on it. We used the Oracle Database version 11g Release 2 (11.2) for deploying and testing the generated mutants. The database management system runs on an IBM AIX Power Systems operating system. The version of the operating system is IBM AIX 7.1.4.2 (Unix), which runs on an IBM Power 780 virtual server. We prepared the configuration file of muPLSQL according to our setup. We also needed to perform test case migration. Hereby, we needed to edit the source code of all the test cases manually to inject hooks such that results can be communicated to muPLSQL. We present and discuss the results in the next section.

6. Results and Discussions

Table 4 summarizes the overall setup and results of our case study. Migration of test cases took 58 hours of a senior software developer. 5,939 mutants were generated within 48 minutes. 1,048 of these were stillborn. The remaining 4,891 mutants were deployed to the database successfully. Test execution process on these mutants took 40 hours and 17 minutes. The overall computation time depends on the SUT, the test suite and the configuration of the computer used for mutant generation and test execution. In our case, the overall duration turned out to be less than 2 days. This means that mutant generation and test execution can be completed over the weekend. This is acceptable because mutation testing is not supposed to be employed on a daily basis. Even the regression testing is performed once a month before each release of the SUT that is used in our case study. Mutation testing is supposed to be applied considerably less often than regression testing. It should be applied when the system is subject to a major evolution. Only then, one might consider to repeat the mutation testing process to evaluate the adequacy of the existing test suite.

The last part of Table 4 lists the results regarding the mutation testing process. 4,211 mutants were killed as a result of 135,875 test executions. 680 mutants remained to be live at the end of the process. We analyzed all these mutants together with the corresponding objects and their test cases. The analysis process took 51 hours. Table 5 lists the detailed results for each of the 19 objects of our study. These objects are enumerated in the first column. The next two columns list the lines of code and the number of test cases for each object. This is followed by the number of applicable operators and the number of generated mutants. Hereby, stillborn mutants are not counted as part of the generated mutants. The last three columns list the number of killed mutants, the number of live mutants and the mutation score obtained for each object.

We can see that there are some objects with 0% mutation score or very close to this score but these are mainly small objects for which a low number of mutants are generated. For instance, only two mutation operators were appli-

Table 4: Summary of the mutation testing and analysis study.

# of lines of code	8,206
# of objects	19
# of test cases	125
# of GMO per Mutant	8
# of SMO per Mutant	14
# of PMO per Mutant	5
Total # of mutation operators per mutant	27
Time for test migration	58 hours
Time for mutant generation	48 minutes
Time for test execution	40 hours 17 minutes
Overall computation time	41 hours 5 minutes
# of stillborn mutants	1,048
# of killed mutants	4,211
# of live mutants	680
Total # of generated mutants	5,939
Time for live mutant analysis	51 hours

cable for generating the executable mutants of *OBJ2*. The rest of the generated mutants were stillborn. We can see that the largest 2 objects (*OBJ14*, *OBJ17*) listed in Table 5 achieved 100% mutation score. These objects count for half of the SUT (5,422 lines of code in total). Figure 2 depicts a bar chart regarding the number of live and killed mutants for the 19 objects. Note that the y-axis of the chart is in logarithmic scale. We can notice high variance among the objects regarding the total number of mutants and the ratio of live mutants. Especially, some of the objects stand out like *OBJ12* with 262 live mutants (see Table 5). We elaborate on the results and specific cases in alignment with our research questions in the following subsections. We conclude the section with a discussion of threats to validity.

6.1. Effectiveness of muPLSQL *(RQ1)*

We manually reviewed all the 680 live mutants. Results are summarized in Table 6, where live mutants are categorized into 3 categories. The first category comprises *equivalent mutants*, which behave the same as the original code. We

Table 5: Properties and the mutation testing results regarding the tested PL/SQL objects (Note: Stillborn mutants are excluded from the count of *Total Mutants*).

Object Properties			**Mutant Generation**		**Test Results**		
Object	**# LOC**	**Test Case #**	**Operators Applied**	**Total # Mutants**	**# Killed**	**# Live**	**Score**
OBJ1	26	2	9	19	4	15	21.05
OBJ2	22	2	2	10	0	10	0
OBJ3	25	3	3	9	8	1	88.88
OBJ4	26	2	5	8	1	7	12.5
OBJ5	26	2	4	7	3	4	42.85
OBJ6	30	3	6	19	15	4	78.94
OBJ7	32	3	6	11	8	3	72.72
OBJ8	26	2	4	7	3	4	42.85
OBJ9	32	4	8	23	5	18	21.73
OBJ10	92	2	10	64	16	48	25
OBJ11	53	5	11	42	16	26	38.09
OBJ12	600	14	22	477	215	262	45.07
OBJ13	53	5	9	26	21	5	80.76
OBJ14	3500	14	11	1655	1655	0	100
OBJ15	250	5	19	200	106	94	53
OBJ16	351	12	11	227	142	85	62.55
OBJ17	2,922	39	20	1,950	1,950	0	100
OBJ18	75	3	13	71	25	46	35.21
OBJ19	65	3	13	66	18	48	27.27
Total	8,206	125	-	4,891	4,211	680	-

investigated the reasons of survival for the other mutants and we identified two main reasons, which correspond to the second and third categories. The second category includes mutants that survive due to *missing scenario verification*. Although there exist test cases for each function or procedure to test the happy flow as well as a number of exceptional scenarios, we found out that some of the exceptional scenarios were not verified by the test cases. The third category includes mutants that survive due to *missing data verification*. PL/SQL programs are tightly coupled with databases. The content of the processed and stored data is as important as the control flow and functional behavior. Therefore, test cases do not only have to verify the output behavior of functions and procedures, but also persistent results of the executed queries. We found out

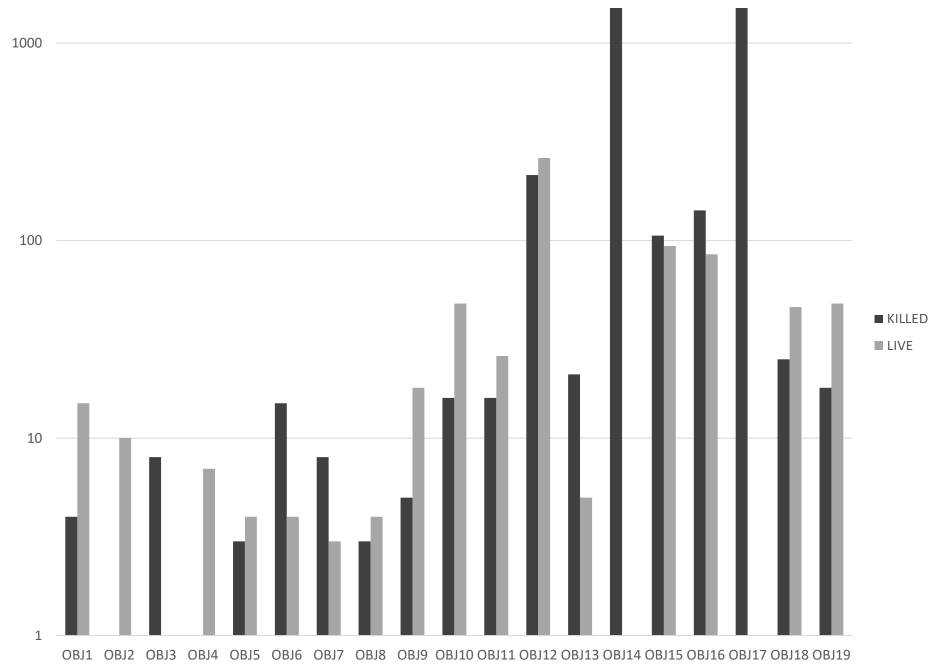

Figure 2: Number of live and killed mutants per PL/SQL object.

that the data stored in database were not verified in some cases. The numbers of live mutants for each of these 3 categories are listed in the last 3 columns of Table 6. The first column is used for categorising these mutants also according to the type of mutation operators used for their generation. We can see that the number of live mutants generated by PMO is relatively low; however, the ratio of equivalent mutants is also low (35%). 30 mutants out of 46 pointed at test inadequacies. On the other hand, 134 mutants out of 391 live mutants generated by GMO (i.e., 64%) turned out to be equivalent. We compare the effectiveness of various mutation operator types in further detail in Section 6.3.

The main goal of mutation testing is to find inadequacies in test cases. Indeed, muPLSQL was very effective, detecting over 370 inadequacies in test cases. In addition, the analysis process exposed faults in the source code. As a side benefit, we discovered 8 faults in the source code during our study. All these faults were revealed during the investigation of equivalent mutants, trying to

Table 6: Investigation of results regarding the reasons of survival for live mutants per mutation operator type.

Operator Type	**Equivalent Mutants**	**Missing Scenario Verification**	**Missing Data Verification**
GMO	134 (19.70%)	55 (8.09%)	102 (15.00%)
SMO	155 (22.79%)	44 (6.47%)	144 (21.18%)
PMO	16 (2.35%)	13 (1.91%)	17 (2.57%)
Total	305 (44.85%)	112 (16.47%)	263 (38.67%)

figure out why they behave the same as the original source code. For instance, there might be variables declared but not used at all within in a function. Therefore, any mutation targeting at these variables do not have any effect on the behavior of the function. Investigation of the corresponding equivalent mutants reveals such unused variables. The detected faults are listed below:

- A procedure was executing always the same scenario among a number of alternatives. This was due to a logical error in conditional statements.
- In a procedure, wrong range values were specified while calling the `SUBSTRING` function that returns a sub-string for the given range of indices within an input string.
- The variable that is returned by a function was initialized with a value that represents successful termination. It was supposed to be initialized with an error code instead and then updated after the successful completion of the transaction is ensured.
- The cursor value of a `SELECT` query was not verified. An error was supposed to be raised in case the transaction is not completed as expected.
- The value of a variable was modified with an `EXCEPTION` block. This was not allowed; however, its effects were not visible as long as the execution does not lead to the corresponding exceptional case.

- One of the input parameters of a function was not being used at all within the function.
- A function was returning 0 as an error code instead of raising an exception. As a result, the function behaves the same when there is an error and when the computed result is actually 0.
- A function was raising the same (wrong) type of exception (e.g., *Null Pointer Exception*) for various errors (e.g., *IO Exception*, *SQL Exception*, etc.).

As a result, we observed that mutation testing is not only effective for improving test cases. Investigation of live mutants also revealed faults in source code. Analysis of live mutants created by OFR (PMO), EXI (PMO) and CRP (GMO) operators led to the detection of two faults each. The other two faults were revealed, while analyzing live mutants created by the LCR (GMO) and AGR (SMO) operators. Note that half of the detected faults were revealed by the analysis of mutants created with PMO operators. In the following subsection, we focus on the cost of mutation testing.

6.2. Cost of Mutation Testing (RQ2)

In our study, mutation testing was effective in identifying test inadequacies and improving the quality of both the system and its test cases. However, it is also subject to costs, which have been a major obstacle for its practical adoption [49]. These costs are caused by several factors [49] including the generation and execution of a large number of mutants as well as the analysis of live mutants to determine whether they are equivalent or not. There also exist an initial setup cost. Test tools need to be integrated and configured to resolve platform dependencies and run the tests on the mutated code [13]. Such efforts were minimal in our study since we used muPLSQL rather than proprietary tools. However, we needed to refactor some of the legacy test cases to be processed by muPLSQL. Migration of test cases took 58 hours of a senior software developer. Mutant generation and test execution process took 41 hours and 5 minutes in total. Reviewing 680 live mutants took 51 hours. So, it took around 5.0 minutes of analysis time per mutant on average. The overall process took almost one person month.

We employed some strategies [49] for reducing costs. We developed muPLSQL to automate the mutant generation and execution tasks. We also defined PL/SQL-specific mutation operators for avoiding the creation of superfluous mutants and for reducing the total number of mutants. For instance, we implemented `CMR` and `RBC` that remove `COMMIT` and `ROLLBACK` statements, respectively. These statements are critical in altering the behavior of distributed database applications like our SUT. A generic statement removal operator would lead to an excessive amount of mutants, many of which would possibly be stillborn.

The only manual processes of our study involve the migration of test cases and the review of live mutants. Therefore, these tasks took the most time and effort. However, test migration task was a one-time effort. This effort is not necessary for the newly developed test cases. On the other hand, review of live mutants was very useful in improving test cases. Besides, investigation of equivalent mutants enabled us to detect faults in source code. Overall, we conclude that costs of mutation testing is affordable and worthed considering its benefits. We evaluate the effectiveness of mutation operators in the following subsection.

6.3. Effective Mutation Operators (RQ3)

Table 7 lists mutation analysis results for each mutation operator. Hereby, we can see the number of *killed*, *live* and *stillborn* mutants as well as their ratio with respect to the total number of mutants (5,939). The last column lists the mutation scores for each mutation operator. We can see that more than half of the mutants (3,271) were generated with SMO. However, only 343 of them survived the test cases. Moreover, 155 mutants out of 343 turned out to be *equivalent* (see Table 6). As a result, only 188 mutants out of 3,271 were usefull for identifying test inadequacies. The total number of mutants and the number of live mutants seem to be large for operators CRP, SVR (GMO) and ROR

Table 7: Mutation analysis results per mutation operator (Op.).

Op. Type	Op. Code	Killed Mutants	Live Mutants	Stillborn Mutants	Total	Score
GMO	ABS	599 (10.1%)	24 (0.4%)	144 (2.4%)	767 (12.9%)	96.1
	AOR	78 (1.3%)	36 (0.6%)	51(0.9%)	165 (2.8%)	68.4
	ASR	277 (4.7%)	6 (0.1%)	59 (1%)	342 (5.8%)	97.9
	CRP	178 (3%)	56 (0.9%)	121 (2%)	355 (6%)	76.1
	LCR	0 (0%)	8 (0.1%)	4 (0.1%)	12 (0.2%)	0
	ROR	221 (3.7%)	52 (0.9%)	47 (0.8%)	320 (5.4%)	81
	RSR	56 (0.9%)	38 (0.6%)	0 (0%)	94 (1.6%)	59.6
	SVR	114 (2%)	71 (1.2%)	108 (1.8%)	293(5%)	61.6
	Total	1523 (25,6%)	291 (4,9%)	534 (9%)	2348 (39.5%)	-
SMO	ABS	190 (57.4%)	50 (15.1%)	91 (27.5%)	331 (5.6%)	79.2
	AGR	22 (62.9%)	11 (31.4%)	2 (5.7%)	35 (0.6%)	66.7
	AOR	72 (58.1%)	12 (9.7%)	40 (32.3%)	124 (2.1%)	85.7
	BTW	52 (92.9%)	4 (7.1%)	0 (0%)	56 (0.9%)	92.9
	IRC	37 (24.2%)	13 (8.5%)	103 (67.3%)	153 (2.6%)	74
	IRH	81 (49.7%)	2 (1.2%)	80 (49.1%)	163 (2.7%)	97.6
	IRP	9 (50%)	0 (%)	9 (50%)	18 (0.30%)	100
	LCR	268 (94.7%)	15 (5.3%)	0 (0%)	283 (4.8%)	94.7
	LIKE	98 (87.5%)	0 (0%)	14 (12.5%)	112 (1.9%)	100
	NLI	6 (8%)	7 (9.3%)	62 (82.7%)	75 (1.3%)	46.2
	NLS	2 (100%)	0 (0%)	0 (0%)	2 (0.03%)	100
	ORD	11 (57.9%)	3 (15.8%)	5 (26.3%)	19 (0.3%)	78.6
	ROR	1563 (86.1%)	216 (11.9%)	36 (2%)	1815 (30.6%)	87.9
	UOI	24 (28.2%)	10 (11.8%)	51 (6%)	85 (1.4%)	70.6
	Total	2435 (74.4%)	343 (10.5%)	493 (15.1%)	3271 (55.1%)	-
PMO	CMR	1 (33.3%)	2 (66.7%)	0 (0%)	3 (0.05%)	33.3
	EXI	43 (52.4%)	18 (22%)	21 (25.6%)	82 (1.4%)	70.5
	OFR	173 (86.9%)	26 (13.1%)	0 (0%)	199 (3.4%)	86.9
	OSR	35 (100%)	0 (0%)	0 (0%)	35 (0.6%)	100
	QER	1 (100%)	0 (0%)	0 (0%)	1 (0.01%)	100
	Total	253 (79.1%)	46 (14.4%)	21 (6.6%)	320 (5.4%)	-
Total		4211 (70.9%)	680 (11.5%)	1048 (17.7%)	5939	86.1

(SMO) in particular. This is due to the high number of arithmetic operators and scalar values used in the source code. We noticed that transaction results are encoded as numbers and business workflow is controlled with integer flags in many cases.

The number of mutants generated by GMO (2,348) is relatively less than those generated by SMO. However, their ratio among all the mutants is still very high, counting for almost 40% of all. Moreover, the number of live mutants (291) is higher, whereas the number of stillborn mutants (534) and the number of equivalent mutants (134) are lower compared to SMO (see Table 6). 188 mutants out of 2,348 were useful for identifying test inadequacies. In particular, ROR (SMO) stands out in the SMO category. Almost 60% of all the mutants and more than half of all the live mutants are generated by this operator.

The total number of mutants (320) and the number of live mutants mutants (46) generated by PMO are very low compared to those generated by GMO and SMO. However, the ratio of stillborn mutants is also very low and only two of the live mutants were identified to be equivalent (see Table 6). 30 mutants were helpful for pinpointing inadequacies in test cases. In this category, OSR, the operator that we introduced in this study specifically for PL/SQL, turns out to be the most successful one. The OFR operator generated no stillborn mutants and 26 out of 46 live mutants were generated by this operator. However, the QER operator introduced only 1 mutant, which is killed.

In general, we observe that the number of generated mutants decreases as more language-specific operators are adopted. On the other hand, the ratio of effective mutants increases. Therefore, language-specificity of mutation operators can be increased for cost-effectiveness, just like trading off recall for precision [50]. Mutation operators can be defined/selected to be even more specific, not only based on the language used but also the type of application. For instance, QER applies on EXECUTE IMMEDIATE instructions, which are more commonly used for batch operations rather than accomplishing interactive user tasks. Therefore, the number of applicable source code elements differs among various types of PL/SQL objects. We discuss validity threats for our study in the following subsection.

We applied an additional evlauation method previoulsy used [51] for assessing the effectiveness and contribution PMO mutation operators. First, we improved the existing test cases such that they can kill all the mutants that are generated by GMO and SMO operators. Then, we executed these improved test cases on mutants that are generated by PMO operators. There were 320 mutants generated by PMO operators. We identified the procedures, on which PMO operators were applied to generate these mutants. We applied all the GMO and SMO operators on these procedures to create mutants. 2,898 mutants were created in total. Then, we improved the set of existing test cases such that they can kill all these mutants. Finally, we executed these refined set of test cases on the 320 mutants created by PMO operators. 35 mutants remained alive. These results prove that PMO operators contribute to the mutation testing process.

6.4. Threats to Validity

Our evaluation is subject to *external validity threats* [52] since all the tested objects are part of a single SUT from a particular application domain. However, it is a legacy system with various features that were subject to a long-term maintenance. Therefore, tested objects represent a diverse set of functionalities in the domain, being developed and tested by different teams over time. High variance in mutation scores of these objects can be interpreted as an indication of this case. Furthermore, our study complements other prior studies [13, 14, 15], which mainly focus on safety-critcal systems.

There exist *internal validity threats* [52] regarding our cost measurements. The study was carried out by the first author together with the support of engineers in the company. She was the same person who developed muPLSQL. Learning, configuring and using this tool might take longer time for others despite its open-source repository that also provides usage and configuration instructions.

Our results regarding the effectiveness of mutation testing and muPLSQL can be subject to *construct and conclusion validity threats* [52]. We employed mutation score as an evaluation metric, which assumes that every mutant is of equal value. This assumption is subject to validty threats, especially when there exist a large number of *subsumed* mutants. A mutant is subsumed if it is killed by every test case that kills another mutant that is already killed by one of the test cases [53]. Subsumed mutants constitute a threat to validity since they inflate the mutation score [54]. We did not rely only on this metric and we also identified many points to improve in test cases as well as several faults in the source code. On the other hand, our SUT is not a safety-critical system and its test suite was not enforced to comply with a certain coverage criteria like MC/DC. Therefore, one can argue that the observed benefits might be simply caused by improper testing of the SUT rather than the effectiveness of mutation testing. However, test effectiveness is not always correlated with structural coverage criteria [55]. Moreover, we were informed that test cases were reviewed by an independent quality assurance team in the company. Of course, this is a manual and error-prone process but maintenance of the SUT has been subject to rigorous development practices at the least. This was confirmed by the fact that the largest 2 objects, counting for half of the source code of experimental objects (5,422 lines of code), achieved 100% mutation score in our study. PL/SQL is mainly used for developing data intensive, business applications like enterprise resource planning applications. These applications are not safety-critical, but they are mission-critical applications [56], which have high reliability and availability requirements. The impact and cost of defects can be very high for these systems. Therefore, our results suggest that the cost of mutation testing is acceptable for reducing the risks involved for PL/SQL programs. Our tool can be used for any PL/SQL program, for which test cases are also developed with PL/SQL. There is no need for the use of an external test automation tool or scripting language.

7. Conclusion and Future Work

We introduced muPLSQL, an open-source tool for applying mutation testing on PL/SQL programs. It facilitates automation for both mutant generation and test execution. We implemented 44 mutation operators, 6 of which are specific for the PL/SQL language.

We conducted an industrial case study with a real business and operation support software system from the telecommunications domain. Mutation testing and muPLSQL turned out to be very usefull in improving both test cases and the source code. Manual inspection of live mutants revealed total 375 test inadequacies in the existing test suite. In addition, 8 new faults were found in the source code. The overall process was subject to costs in terms of both manual effort and computation time. However, we conclude that these costs are affordable especially considering the benefits gained from the process. The most number of mutants were generated by mutation operators that are generically applicable to procedural languages such as Java. However, a large portion of these were either killed or eliminated as equivalent mutants. The number of mutants that were generated by PL/SQL-specific operators were low; however, these mutants were effective in improving test cases. In general, the number of generated mutants decreases as more language-specific operators are adopted. On the other hand, the ratio of effective mutants increases. Therefore, language-specificity of mutation operators can be increased for cost-effectiveness.

We have witnessed that mutation testing is subject to high costs and considerable benefits at the same time. We would recommend its use for any safety-critical, mission-critical, or business-critical system, where the incurred costs would be paid off with increased reliability. However, resources might be insufficient for its adoption during the development of cost-sensitive systems. An initial investment is necessary for tool development and/or integration. In our case, we had to develop a mutation testing tool from scratch dedicated to PL/SQL programs. We even had to implement mutation operators specific for the PL/SQL language. These efforts might not be necessary for applications that are developed with mainstream programming languages since there are many mature tools available for these languages. Still, the cost of adopting these tools for establishing an automation infrastructure should not be underestimated. This infrastructure should support the generation of mutants, execution of these mutants against a test suite and logging results for inspection. The cost of developing and maintaining such an infrastructure is high even if an existing mutation testing tool is adopted. We had to make changes to the existing test platform and adapt our test cases although we integrated our own tool. Therefore, the initial investment for cost-sensitive systems can only be amortized in the context of software product line engineering, where a family of products share a common software base. Computation time for test execution and the manual effort required for reviewing live mutants can also be considered as additional costs. However, these are not significant compared to the cost of development and maintenance of the automation infrastructure according to our experience. This cost is amplified for PL/SQL programs due their coupling with a database management system. Deployment of these programs take considerable time and their mean recovery time is high in case of failures. Hence, the whole platform must be replicated in a test environment not to impact the production environment.

Possible future work directions include enhancing the level of automation for several tasks. Detection and elimination of stillborn mutants can be automated. Analysis of live mutants can also be partially automated to eliminate some of the equivalent mutants. Automated test data generation might be possible for completing the missing test cases that lead to live mutants. muPLSQL is also designed to be extendable for incorporating new mutation operators.

Acknowledgment

We would like to thank the database administration team at Turkcell for their technical support throughout the case study. In particular, we would like to thank Ahmet Karaduman for his help in performing memory optimization, which is necessary to manage the memory overhead caused by the deployment and execution of an extensive number of PL/SQL objects on an Oracle database.

References

[1] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in Computers, Vol. 112, Elsevier, 2019, pp. 275–378.

[2] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) (2010) 649–678.

[3] Y. Jia, M. Harman, MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language., in: Proceedings of Testing: Academic Industrial Conference Practice and Research Techniques, 2008, pp. 94–98.

[4] T. Chekam, M. Papadakis, Y. L. Traon, Mart: A mutant generation tool for LLVM, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, p. 1080–1084.

[5] Y.-S. Ma, J. Offutt, Y.-R. Kwon, Mujava: a mutation system for java, in: Proceedings of the 28th International Conference on Software engineering, 2006, pp. 827–830.

[6] H. Coles, T. Laurent, C. Henard, M. Papadakis, A. Ventresque, PIT: a practical mutation testing tool for java (demo), in: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, p. 449–452.

[7] B. Pribyl, S. Feuerstein, Oracle PL/SQL Programming , 6th Edition, O'Reilly Media, 9781449324452, 2014.

[8] J. Harper, M. McLaughlin, Oracle Database 11g PL/SQL Programming Workbook, Oracle Press, 2010.

[9] L. D. Paulson, Developers shift to dynamic programming languages, IEEE Computer 40 (2) (2007) 12–15.

[10] M. Altιnιşιk, H. Sözer, Automated procedure clustering for reverse engineering pl/sql programs, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016, pp. 1440–1445.

[11] M. Altinisik, E. Ersoy, H. Sözer, Evaluating software architecture erosion for PL/SQL programs, in: Proceedings of the 11th European Conference on Software Architecture, 2017, pp. 159–165.

[12] E. Ersoy, K. Kaya, M. Altinisik, H. Sözer, Using hypergraph clustering for software architecture reconstruction of data-tier software, in: Proceedings of the 10th European Conference on Software Architecture, 2016, pp. 326–333.

[13] R. Ramler, T. Wetzlmaier, C. Klammer, An empirical study on the application of mutation testing for a safety-critical industrial software system, in: Proceedings of the Symposium on Applied Computing, 2017, pp. 1401–1408.

[14] R. Baker, I. Habli, An empirical evaluation of mutation testing for improving the test quality of safety-critical software, IEEE Transactions on Software Engineering 39 (6) (2012) 787–805.

[15] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, I. Medina-Bulo, Evaluation of mutation testing in a nuclear industry case study, IEEE Transactions on Reliability 67 (4) (2018) 1406–1419.

[16] J. Offutt, P. Ammann, L. Liu, Mutation testing implements grammar-based testing, in: Proceedings of the 2nd Workshop on Mutation Analysis, 2006, pp. 12–12.

[17] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of mujava, in: Proceedings of the 2006 international workshop on Automation of software test, 2006, pp. 78–84.

[18] J. Tuya, M. J. Suárez-Cabal, C. De La Riva, Mutating database queries, Information and Software Technology 49 (4) (2007) 398–417.

[19] D. Litchfield, The Oracle® Hacker's Handbook: Hacking and Defending Oracle, Wiley, 2007.

[20] M. McLaughlin, Oracle Database 11g PL/SQL Programming, Oracle Press, 9780071643566, 2008.

[21] S. Gupta, Oracle Advanced PL/SQL Developer Professional Guide, Packt Publishing, 2012.

[22] A. Acree, T. Budd, R. DeMillo, R. Lipton, F. Sayward, Mutation analysis, Tech. Rep. GIT-ICS-79/08, Georgia Institute of Technology (1979).

[23] R. DeMillo, R. Lipton, F. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (4) (1978) 34–41.

[24] A. J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering and Methodology 1 (1) (1992) 5–20.

[25] T. Chekam, M. Papadakis, Y. L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering, 2017, pp. 597–608.

[26] H. Zhu, P. Hall, J. May, Software unit test cov-erage and adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.

[27] D. Schuler, A. Zeller, (un-)covering equivalent mutants, in: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, 2010, pp. 45–54.

[28] R. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262.

[29] A. J. Offutt, Jie Pan, Detecting equivalent mutants and the feasible path problem, in: Proceedings of the 11th Annual Conference on Computer Assurance, 1996, pp. 224–236.

[30] M. Papadakis, M. Delamaro, Y. Le Traon, Mitigating the effects of equivalent mutants with mutant classification strategies, Science of Computer Programming 95 (2014) 298–319.

[31] K. N. King, A. J. Offutt, A fortran language system for mutation-based software testing, Software: Practice and Experience 21 (7) (1991) 685–718.

[32] Y.-S. Ma, J. Offutt, Y. R. Kwon, Mujava: an automated class mutation system, Software Testing, Verification and Reliability 15 (2) (2005) 97–133.

[33] H. Agrawal, R. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, E. Spafford, Design of mutant operators for the C programming language, Tech. Rep. SERC-TR-41-P, Purdue University, West Lafayette, Indiana (1989).

[34] Y.-S. Ma, J. Offutt, Description of muJava's Method-level Mutation Operators, available online, accessed on February 3, 2020 (2018).
URL `https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`

[35] J. Tuya, M. J. Suarez-Cabal, C. De La Riva, Sqlmutation: A tool to generate mutants of sql database queries, in: Proceedings of the 2nd Workshop on Mutation Analysis, 2006, pp. 1–1.

[36] C. Zhou, P. Frankl, Mutation testing for java database applications, in: Proceedings of the IEEE International Conference on Software Testing Verification and Validation, 2009, pp. 396–405.

[37] A. J. Offutt, R. H. Untch, Mutation 2000: Uniting the orthogonal, in: Mutation testing for the new century, Springer, 2001, pp. 34–44.

[38] K. Pan, X. Wu, T. Xie, Automatic test generation for mutation testing on database applications, in: Proceedings of the 8th International Workshop on Automation of Software Test, 2013, pp. 111—-117.

[39] S. Harper, PL/SQL Unit Testing, Apress, Berkeley, CA, 2011, pp. 97–120.

[40] N. Li, M. West, A. Escalona, V. H. Durelli, Mutation testing in practice using ruby, in: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops, 2015, pp. 1–6.

[41] S. Nica, R. Ramler, F. Wotawa, Is mutation testing scalable for real-world software projects, in: Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle, 2011.

[42] G. Petrović, M. Ivanković, State of mutation testing at Google, in: Proceedings of the 40th international conference on software engineering: Software engineering in practice, 2018, pp. 163–171.

[43] A. J. Offutt, G. R. A. Lee, R. H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering Methodology 5 (1996) 99–118.

[44] Y.-S. Ma, J. Offutt, Description of class mutation mutation operators for Java, available online, accessed on February 3, 2020 (2014).
URL `https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`

[45] A. Derezińska, An experimental case study to applying mutation analysis for sql queries, in: Proceedings of the International Multiconference on Computer Science and Information Technology, 2009, pp. 559–566.

[46] Oracle Corporation, PL/SQL Language Reference, available online, accessed on March 3, 2022 (2013).
URL `https://docs.oracle.com/database/121/LNPLS/toc.htm`

[47] J. Sathyan, Fundamentals of EMS, NMS and OSS/BSS, Auerbach Publications, 2016.

[48] K. Fowler, Mission-critical and safety-critical development, IEEE Instrumentation Measurement Magazine 7 (4) (2004) 52–59.

[49] A. Pizzoleto, F. Ferrari, J. Offutt, L. Fernandes, M. Ribeiro, A systematic literature review of techniques and metrics to reduce the cost of mutation testing, Journal of Systems and Software 157 (2019) 110388.

[50] M. Buckland, F. Gey, The relationship between recall and precision, The Journal of American Society for Information Science 45 (1) (1994) 12–19.

[51] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, Y. L. Traon, How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults, Empirical Software Engineering 23 (2018) 2426–2463.

[52] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2012.

[53] P. Ammann, M. Delamaro, J. Offutt, Establishing theoretical minimal sets of mutants, in: Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation, 2014, pp. 21–30.

[54] M. Papadakis, C. Henard, M. Harman, Y. Jia, Y. L. Traon, Threats to the validity of mutation-based test assessment, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, p. 354–365.

[55] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435—-445.

[56] D. Sprott, Enterprise resource planning: componentizing the enterprise application packages, Communications of the ACM 43 (4) (2000) 63–69.