Towards Automatic Generation and Continuous Improvement of Functional Test Cases: The Case of the Test-Duo Framework and Mutation Testing

CHIN-YUN HSIEH, CHEN-HSIN TSAI AND YU CHIN CHENG
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: {Hsieh; s3419011; ychcheng}@ntut.edu.tw

The Test-Duo framework is proposed for automating the task of generating functional test cases for execution on an automatic testing platform. With Test-Duo, the tester focuses on the tasks of refining and annotating use cases and preparing test data sets with tool support, leaving the tasks of generating the actual test cases and marshaling their execution to Test-Duo. The generated test cases are subjected to mutation testing, whose results are analyzed by the tester to identify and apply appropriate strategies for systematically and iteratively improving the quality of the generated test cases. The effectiveness of this improvement process is demonstrated with a case study.

Keywords: functional test case generation, use case, mutation testing, annotation, test case quality improvement, test automation

1. INTRODUCTION

Functional testing is a common practice for ensuring that the functional requirements of the system are implemented as intended [1]. Performed without tool support, functional testing can be both labor-intensive and error-prone. In the current trend in software development to encourage early and frequent testing at all levels, methods and tool support for test automation have contributed to improving the situation for functional testing [2]. In particular, tools such as FIT/FitNesse [3] and ROBOT framework [4], by allowing test cases to be written as scripts readable to non-programmers, have contributed greatly to reducing the effort on automating the tasks of running functional tests and collecting reports. However, the task of creating test cases to run on these tools remains largely a manual process.

In this paper, the Test-Duo framework is proposed to further automate the task of creating test cases for functional testing and to marshal the execution of the generated test cases on an automatic testing platform such as FIT/FitNesse and ROBOT framework. Using use cases as the functional specifications of the system under test (SUT) and employing the state space search approach [5], Test-Duo supports the tester in performing functional testing against the SUT as described in the following steps: (1) the tester makes the use case explicit by annotating the use case; (2) the tester prepares sets of input data; (3) until a tester-specified time of execution expires or the state space has been exhaustively searched, Test-Duo iteratively generates the test cases according to a spe-
specific search regime and sends them to the underlying automatic testing platform for execution; and (4) Test-Duo records any test cases that cause failures of the SUT. Note that the first two steps are manually performed by the tester with tool support from the Test-Duo framework while the last two steps are automatically performed by the Test-Duo framework.

As with any other method of testing, it is important that the set of test cases obtained be put under a measurement scheme for assessing and improving its quality. In this work, mutation testing [6-8] is adopted as the measurement scheme for assessing and improving the quality of test cases generated by Test-Duo. Mutation testing is chosen as the quality indicator because it is experimentally shown that test cases which kill the mutants will also identify the real faults [9]. Furthermore, mutation testing has been shown to be an indication of the fault detection ability of a test suite. Given two test sets, the test set that achieves a higher mutation score is deemed to have a better quality [10]. In the case of Test-Duo, since the test set generated depends on the test materials prepared by the tester in steps (1) and (2) above, and since Test-Duo is given a time budget for testing the SUT, an improvement is defined as follows:

A test set $S'$ generated with test materials $M'$ is an improvement of another test set $S$ generated with test materials $M$ if $M$ can be modified in some way into $M'$ so that the mutation score of $S'$ is not less than that of $S$ and the time for executing $S'$ is less than that of executing $S$.

Improvements – increasing the mutation score and/or reducing the time budget – are thus made through modifying test materials. To this end, applicable strategies are identified based on analysis performed on the result of mutation testing. In particular, three improvement strategies are closely related to the improvement of test materials: identifying missing test keywords for forcing the SUT to fail, refining data set for enabling a particular fault-exposing data combination to form, and fixing the missing parts of use case that deviate from the behaviors implemented in the SUT. The applicability of the three strategies will be further elaborated in section 3 of this paper.

To evaluate Test-Duo and the improvement made possible through mutation testing, we have implemented the Test-Duo framework and used ROBOT framework as the underlying automatic testing platform. We have designed a case study that applies Test-Duo in testing a point of sale (POS) application. Encouraging results are obtained in a case of three rounds of improvements. Specifically, in the first round, the test cases generated with the initial test materials receive 81.6% in mutation score in 72 hours of execution time. Mutation analysis leads to the refinement of data set in which a composite data object is added. Subsequently, in the second round, the generated test set receives 90.5% in mutation score in 24 hours of executing time. Again, mutation analysis is performed and shows that mutations are introduced to locations in code of the SUT that are not reachable by the test cases generated. Accordingly, the faulting use case is fixed with annotations added and data set revised. Thus, finally, a mutation score of 100% is reached with the execution time of 2.25 hours in the third round. It is interesting to note that a real bug is caught during the second round in a continuous effort to increase mutation score.

The rest of this paper is organized as follows. Section 2 presents the architecture and the operations of Test-Duo. Section 3 details on how mutation testing is applied to
form a loop of improvement using the three strategies described in the previous paragraphs. Section 4 describes a case study and presents the findings. Section 5 discusses the consequences of applying Test-Duo and its limitations. Section 6 reviews the related work. Lastly, section 7 offers the concluding remarks and discusses directions of future work for the Test-Duo framework.

2. THE TEST-DUO FRAMEWORK

2.1 An Architectural Overview of the Test-Duo Framework

Although the main functional requirement of Test-Duo is to generate test cases from use cases, the functional requirement alone does not determine the architecture of Test-Duo; quality requirements need to be considered. There are three quality requirements that shape the architecture of Test-Duo:

- Test-Duo can generate a large number of test cases. Executing these test cases can consume a lot of computing resources. To shorten the turnaround time of obtaining test results, Test-Duo should support parallel execution of the test cases.
- As mentioned in introduction, the existing acceptance test platform such as FIT/FitNesse and ROBOT framework have contributed greatly to reducing the effort on automating the tasks of running functional tests and collecting reports. However, the task of creating test cases to run on these tools remains largely a manual process. Thus we design Test-Duo platform to generate functional test cases and integrate existing test platforms. To leverage existing acceptance testing frameworks, it must be easy for Test-Duo to interoperate with them.
- Test-Duo must allow a legacy system to be hooked up as an expected result generator. One of the most difficult challenges in test case generation is to generate the expected test results, but in some commonly occurring circumstances, it is possible to obtain expected result through another program. For example, in revamping a legacy system, by running the legacy system alongside its replacement test-step by test-step, a legacy system becomes a correct expected result generator.

Reflecting the three quality requirements, Test-Duo is architected as depicted in Fig. 1.
- To enable parallel execution of the test cases, Test-Duo is separated into two main components: the Test Director and the Test Driver. Specifically, the Test Director handles the tasks of automatically creating test cases for functional testing while the Test Driver marshals the execution of the generated test cases on an automatic testing platform. Parallel execution of test cases is accomplished by applying the Master-Worker work partitioning architecture pattern [11] of deploying multiple Test Drivers to work with the Test Director.
- To leverage the automatic testing platform, a set of scripts and custom keywords are created according to the target platform. Once the test starts, testing platform is launched to run our own scripts. Inside the scripts are the custom keywords that handle the testing information passed from Test Driver. These custom keywords are responsible for composing test cases according to testing information, launching SUT
keywords accompany with test data, and sending the testing result back.

- To get the expected result from a legacy system, the event sent to the SUT is also sent to the legacy system in order to keep both systems in the same state. A set of testing keywords for legacy system are created according to the SUT. When performing the test, the data is input to both systems. If the expected result is not specified when testing the SUT, Test Director will consult the legacy system by calling the corresponding keyword to get the expected result.

Note that the shaded rectangles in Fig. 1 represent external components or systems that interoperate with Test-Duo. The Test Director uses as input the required Test Materials which are obtained after analyzing the use cases and generates (1) functional tests and (2) directives for directing the Test Driver. The Test Director comprises two components: the State-Space Search Method which traces the possible execution path according to the structure of the given use cases and the Test Case Generator which creates the concrete test cases and handles interactions with the Test Driver. Collaborating with the Test Director, the Test Driver responds to the directives, feeding the functional tests to the underlying Automatic Testing Platform to test the SUT and returning the results to the Test Director. Note that the Implemented SUT Keywords are the implementation of the test keywords in test materials. These keywords are used to interact with the SUT to verify its correctness. In our approach, only the automatic testing platform possesses the capability to execute the functional test. The Test-Duo framework leverages this testing capability without implementing our own testing platform.

Fig. 1. Architecture of the Test-Duo framework.

The collaboration between the Test Director and the Test Driver in one test case is depicted in Fig. 2. First, the Test Director generates the skeleton of the test case to be executed and then fills in, step by step, the testing commands, which are either driver directives or test keywords with arguments, and sends them to the Test Driver. On the receiving end, if the command is a directive, the Test Driver sets up or tears down or terminates the test case. Otherwise, the command is a test keyword with the required arguments and is passed to the automatic testing platform for execution. The result is sent back to the Test Director to decide the next operation. If a result of failure is re-
turned, a Failed Test Case is created. The test case’s remaining steps are filled in and the test case is written to a file for further reviewing and regression testing. Otherwise, the Test Director continues until the test case is completed without failure.

- The flow of events in a use case is modeled with a structured program [12]: a use case step is a statement; branching in use case is a selection statement; and a loop of use case steps is an iteration statement. Thus, the legal moves among the steps in a use case can be obtained by parsing its structured program model.
- A test keyword can be an action that an actor performs against the SUT, an action that an actor gets information from the SUT, or an assertion for checking the state of the SUT. Test keywords from each of the steps and pre-/post-conditions in a use case are identified by the tester and named with readable description of their purpose. These keywords simulate the actions when an actor takes in interacting with the SUT. Keywords that reflect the actor’s actions should be fine-grained in order to be reused in use cases of the SUT.
- A data set is a collection of data that is used as arguments to the test keywords. The use case in Fig. 3 (which is adapted from [13]) is used as an example to describe how the Test Director generates test cases from test materials derived from it.
Use Case Name: Process Sale  
Primary Actor: Cashier  
Pre-conditions:  
- Cashier is identified and authenticated.  
Post-conditions:  
- Accounting and Inventory are updated.  
Main Success Scenario:  
1. Customer arrives at POS checkout with goods and/or services to purchase.  
2. Cashier starts a new sale.  
3. Cashier enters item identifier.  
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.  
Cashier repeats steps 3-4 until indicates done.  
5. System presents total with taxes calculated.  
6. Cashier tells Customer the total, and asks for payment.  
7. Customer pays and System handles payment.  
8. System logs completed sale and sends sale and payment information to the external system.  
10. Customer leaves with receipt and goods.  
Extensions:  
1a. Customer or Manager indicate to resume a suspended sale:  
1. Cashier performs resume operation, and enters the ID to retrieve the sale.  
2. System displays the state of the resumed sale, with subtotal.  
3. Cashier continues with sale.  
2a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples):  
1. Cashier verifies, and enters tax-exempt status code.  
2. System records status (which it will use during tax calculations).  
3a. Invalid item ID (not found in system):  
1. System signals error and rejects entry.  
2. Cashier responds to the error.  
3. Cashier performs Find Product Help to obtain true item ID and price. 

Fig. 3. Use case Process Sale of the POS program.

2.2 Test Case Generation

The Test Director uses the state space scheme [5] for generating the test steps [5]. Fig. 4 illustrates the state space graph and the legal moves of the use case (Fig. 3) of the SUT. The state space graph consists of nodes and directed edges. A node represents a state the SUT is in. A directed edge represents the execution of a use case step by the SUT. Thus, a directed edge $E$ from a node $N$ to a node $M$ means that the SUT changes from the state represented by $N$ to the state represented by $M$ by executing the use case step represented by $E$. The nodes are labeled as follows. The unique start node represents the state the SUT is in before the use case execution begins. A node $N$ is labeled with the sequence of use case steps that takes the SUT from the start node to $N$. A node that terminates the use case (i.e., any of the gray nodes in Fig. 4) represents the state of the SUT is in upon terminating a use case scenario. For such a node, which is referred to as a use-case terminating node in the rest of this paper, functional tests are generated by the Test Director. If a generated functional test of a use-case terminating node fails the SUT, the node is labeled as a goal node (i.e., the double-circled node in Fig. 4). The method of state space search can be any standard search algorithm such as depth-first search, breadth-first search, and so on. In this paper, depth-first iterative deepening [5] is used. With the algorithm, minimum-length test cases will be tested first.
A test case is generated by the Test Director as follows. In the above state space graph, when a use-case terminating node is reached, the node’s label – which is exactly the use case steps leading from the start node to the current node – is prefixed with a precondition check step and post-fixed with a post-condition check step to form a test case skeleton. For each step in the skeleton, test keywords are filled in for the actions that the actors or SUT performs and for the assertions that check the state of the SUT. Note that some of the steps are filled with more than one test keyword. Fig. 5 takes the right-most branch in Fig. 4 as an example, where the step e1 has four test keywords filled in and the steps m1, m2, and m10 have none.

Fig. 4. State space of the use case Process Sale.

Legal Moves:
legalMove(m1, m2).
legalMove(m2, m3).
legalMove(m3, m4).
legalMove(m4, m5).
legalMove(m5, m6).
legalMove(m6, m7).
legalMove(m7, m8).
legalMove(m8, m9).
legalMove(m9, m10).
legalMove(m10, m1).
legalMove(m1, m2).
legalMove(m2, m3).
legalMove(m3, m4).
legalMove(m4, m5).
legalMove(m5, m6).
legalMove(m6, m7).
legalMove(m7, m8).
legalMove(m8, m9).
legalMove(m9, m10).
legalMove(m10, m1).

m: main success scenario
e: extension

Fig. 5. A functional test case skeleton created from a use-case terminating node.

cashier authenticated
resume sale
input data to the system
status should be
total should be
input data to the system
price should be
description should be
total after discount should be
accounting updated
inventory updated

Pre-C
m1

m2

m3

m4

m5

m10

Post-C
For the arguments required by each test keyword, the Test Director fills them in with two approaches:

- If the argument can be decided before the test is executed, it is selected from a data set prepared by the tester as described in the next section. As an example test case in Fig. 5, argument of test keyword “input data to the system” in the step m3 and test keyword “price should be” in the step m4 are filled in with the identifier and the expected price, respectively, of the item being processed.
- Otherwise, the argument can be determined only after the test case is executed. For example, in Fig. 5, test keyword “total after discount should be” in the step m5 cannot be determined in advance because it depends on the items selected in the previous steps which include items contained in the resumed sale (the step e1) and the additional item purchased (the step m3). In this case, the Test Director calls the expected result generator in Fig. 1 to get the needed arguments for the given test keyword. Test-Duo provides interface for tester to hook up a suitable expected result generator. In section 4, an expected result generator, which is implemented by applying the concept of program slicing [14], is hooked up to provide excepted results for testing the POS use case.

2.3 Test Data Preparation and Usage

Fig. 6 illustrates how the test data is prepared and used for each test keyword in five steps. The first four steps are manually performed by the tester. As will be seen below, these four steps are very similar to the steps performed during object-oriented analysis. The result obtained by these four steps enables the automation of the fifth step, where the Test Director of the Test-Duo framework selects arguments passed to a test keyword from various data sets prepared by the tester.

A. Identify use case vocabularies

Use case vocabularies are the terms used to describe the information exchanged
between the actors and the system in the use case. For example, “item identifier”, “item
description” and “item price” are captured from the step 3 and 4 of Fig. 3. The noun
phrase identification method [15] can be used for the use case vocabularies identification.
These noun phrases in the use cases are the candidates to be the arguments of the test
keywords.

B. Aggregate use case vocabularies into a number of conceptual classes

Conceptual classes are the aggregations of closely related use case vocabularies
which form real world concepts. For example, the class “Item for sale” is created by ag-
gregating the three use case vocabularies “item identifier”, “item price” and “item de-
scription” because they refer to the attributes of the item processed in the step 3 and 4 of
the use case of Fig. 3. As another example, the class “Non-existing item” aggregates only
the use case vocabulary “item identifier” as suggested by the step 3 and 3a of the use
case of Fig. 3. The domain modeling guidelines [13] is used to perform the high level
analysis of the SUT and create the conceptual classes to stand for the real world objects
in the use case domain. These conceptual classes should be created from the view of use
case actor that focuses on the external interface of the SUT. The use case vocabularies
and conceptual classes reflect to the real world items and concepts while using the SUT.
They are similar to the conceptual models obtained during object-oriented analysis. Thus,
when such conceptual models are available, they can be used as the use case vocabular-
ies and conceptual classes in the proposed method.

C. Instantiate concrete objects for each conceptual class

Concrete objects are the instances of a conceptual class and serve as the concrete
arguments when performing the testing. For example, a “Juice” object with three attrib-
te-value pairs, which include “item identifier” with value “3004”, “item price” with
value “$28”, and “item description” with value “Orange Juice”, is an instance of the
conceptual class “Item for sale”. Concrete objects of the class “Items for sale” form a
data set and objects of the class “Non-existing item” form another.

D. Specify argument type for each test keyword

A test keyword may need one or more arguments to completely represent the action.
For each argument in a test keyword, a type of the form “conceptual class”. “use case
vocabulary” is specified for it. For example, in the test keyword “input data to the sys-
tem” in the step m3 of Fig. 5, the argument has the type of “item for sale”. “item identi-
fier”. Note that for test keywords that use arguments obtained from the expected result
generator, the type is left unspecified because it is entirely determined by the expected
result generator.

E. Select argument from data sets

The Test Director uses argument type information to randomly select appropriate
attributes of concrete objects from the data sets to serve as arguments to test keywords in
a test case skeleton. Also, in a step that has multiple different subsequent steps, the ar-
gument type information guides the Test Director in the selection of concrete object for
each subsequent step. For example, a keyword “Input data to the system” is annotated to
the step m3 in the use case of Fig. 3 to represent that an item is purchased by the cus-
tomer. The keyword “Input data to the system” takes an argument. According to the structure of the use case, either the step m4 in the normal flow or the step e3 in the extension flow can be executed after the step m3. The former requires that the test keyword “Input data to the system” in the step m3 be assigned an “item identifier” argument from concrete object of the class “Item for sale” while the latter requires an argument from concrete object of the class “Non-existing item”. As represented in Fig. 7, by looking ahead to the next step, an appropriate argument to the test keyword in the step m3 can be set accordingly.

![Fig. 7. Applying different argument according to the execution path.](image)

3. MEASURING AND IMPROVING THE QUALITY OF GENERATED TEST CASES

In this paper, mutation testing is applied to measure the quality of the test cases generated by Test-Duo. Furthermore, results of mutation testing are analyzed to help identifying potential enhancements in the test materials.

Mutation testing is a fault-based testing technique for measuring the quality of the test cases in finding bugs inside the SUT [7, 8, 10, 16, 17]. It can be applied not only on source code [6, 18], but also on specifications [19, 20]. For other applications, mutation testing can also be applied to generate test data [21] and to reduce the number of bug candidates when debugging [22]. In this work, we have focused on using program mutation [8], which is done by systematically seeding a syntactic change inside the original program with the intention to cause a SUT failure. Each mutated program is called a mutant which contains a different syntactic change to simulate a fault that the programmers inadvertently introduce into the program during development or maintenance work.

Program mutation testing can be applied at unit level [23, 24], integration level [25-27] and system level [28] to measure the test case quality. The vast majority of existing work has focused on the unit level or the integration level. In our work, mutation testing is applied for measuring and improving the quality of the functional test cases generated by Test-Duo. Strong mutation is used since the generated tests detect difference between system’s specification and its externally observable behaviors [8]. Under
strong mutation, a mutant is either an equivalent mutant that produces the same output as the original program or a non-equivalent mutant that can cause an externally observable failure to occur. If a failure caused by a mutant is detected by a test case, the mutant is said to be killed, otherwise, the mutant remains alive. Only non-equivalent mutants can be killed; equivalent mutants must be identified by the tester. Mutation score of a test set is defined as the number of mutants killed by a test set divided by the total number of non-equivalent mutants.

3.1 A Mutation-Improvement Loop

Test-Duo’s capability of finding failures, which is measured with the mutation score of the generated test cases, is iteratively improved by the loop consisting of a mutation stage and an improvement stage as shown in Fig. 8. By improving, we mean not only killing as many mutants as possible, but also killing the mutants as early as possible. In our approach, test cases are generated by Test-Duo based on the test materials and are executed within a given time. Through the process of improving the generated test cases, the test materials are improved. Furthermore, by improving Test-Duo’s capability of finding failures, potential bugs behind the failures in the SUT will be identified and fixed using the improved test materials.

The loop of improvement is adapted from the generic mutation analysis process [16]. Before entering the loop, the SUT should be well tested by Test-Duo using existing test materials. The loop is then entered if Test-Duo finds no failures using the existing test materials within the given time constraint. In other words, the capability of finding failures within the given time has reached its limit. To extend the capability of test materials beyond its current limit, the loop enters the mutation stage and the mutation score is
measured. Iteration of the loop ends if the mutation score has reached a targeted level. Otherwise, the loop enters the improvement stage to refine test materials and the loop then continues iterating. Thus, upon terminating the loop, Test-Duo’s capability of finding failures using the test materials has reached a satisfactory level.

Next, we describe the mutation stage and the improvement stage in detail.

3.2 Mutation Stage

In the mutation stage, the mutants of the SUT are created exactly once the first time mutation stage is entered. Next, Test-Duo is employed to kill the mutants that are still alive within a given time constraint. When this time expires, testing is stopped and the mutants are classified into two groups: those that are killed and those that are still alive as summarized in Table 1.

Table 1. Result-based mutant classification and corresponding improving strategies.

<table>
<thead>
<tr>
<th>Mutant status</th>
<th>Category</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Killed</td>
<td>Mutant killed within a short time</td>
<td>– The expected normal case; no improvement is needed</td>
</tr>
</tbody>
</table>
|               | Mutant killed after a long time | – Identifying missing test keywords for forcing the SUT to fail  
|               |                                    | – Refining data set for enabling a particular fault-exposing data combination to form |
| Alive         | Equivalent | – Removing mutant from the next round of testing to save time |
|               | Unreachable by use case | – Fixing the missing parts of use case that deviate from the behaviors implemented in the SUT |
|               | Undiscovered | – Identifying missing test keywords for forcing the SUT to fail  
|               |                                    | – Refining data set for enabling a particular fault-exposing data combination to form  
|               |                                    | – Allowing more time for testing |

In Table 1, the strategies that can be applied to improve mutation score are identified. Note that only the three strategies italicized in Table 1 can be counted as improvement strategies:

1. identifying missing test keywords for forcing the SUT to fail,  
2. refining data set for enabling a particular fault-exposing data combination to form, and  
3. fixing the missing parts of use case that deviate from the behaviors implemented in the SUT.

These will be referred to as strategy 1, 2 and 3, respectively, in the sections below.

A. Mutants that are killed

For each mutant in the first group, the time spent by Test-Duo in killing it is recorded. Based on the recorded time spent, the mutant is classified into one of the two categories:
**Mutant is killed within a short time:** this represents the normal case we expect in conducting mutation testing. No improvement strategy is needed.

**Mutant is killed after a long time:** this represents that there is still room of improvement since we want to shorten the time spent. There are two applicable strategies for shortening the time for killing a mutant in this category:

- The mutant has a fault that becomes active only if several conditions are satisfied simultaneously by the test data selected, which can take a long time since test data is selected randomly. Providing data sets that satisfy the conditions more easily can thus set the mutation-faulted code into action at an earlier time (*strategy 2*).
- The mutant causes a failure but no assertion keywords yet exist to detect it at the first opportunity, and the mutant is killed by an existing test keyword due to another failure indirectly caused by the same mutation fault. Since we are interested in killing mutation directly and as early as possible, both the use case and the source code where the mutation fault is located are analyzed. A new keyword is introduced in the annotations to the use case so that the assertion fails at the first opportunity (*strategy 1*).

It should be noted that whether a mutant is killed in a short or long time depends on the circumstance in mutation testing and the characteristic of the SUT. For example, the allowed testing time and available computing power are typical conditions to be considered. The tester decides what is long or short by considering the circumstance on hand.

B. Mutants that remain alive

A mutant in the second group is classified into one of the three categories depending on the surviving reason:

**Equivalent mutant:** the mutant always produces the same output as the original program regardless of whether the internal system state is changed or not. Such mutant is removed from next round of mutation testing to save testing time.

**Unreachable by use case:** the mutant has a syntactic change introduced at a code location that is never touched upon in executing the use case. This indicates that the implementation of the SUT contains functionalities that are not mentioned in the use case. During its lifetime, a software system may have new features added to it without the use case document being updated. Use case refinement is required to kill such mutants (*strategy 3*). New descriptions are added to the use case to describe the omitted behaviors of the SUT. The related annotations and test data are also added for the new descriptions of the use case accordingly.

**Undiscovered mutant:** the mutant should have been killed, but the test cases generated by Test-Duo are insufficient to identify it. A mutant can survive either because the current test materials are insufficient for Test-Duo to generate any test case to kill it or because time runs out before it is killed. Strategies for dealing with the undiscovered mutants are similar to the case of shortening the time spent: refine annotations and data sets to force the SUT to fail as early as possible. When all else fails to produce an improve-
ment, it is always possible to increase the time allowed for Test-Duo to kill more mutants. It should be noted, however, since Test-Duo uses state space search in generating the test cases, the effectiveness of increasing time depends on the search regime used. In the case of depth first iterative deepening used in this paper, a linear increase in the length of test cases requires an exponential increase in execution time. In general, refining annotations and data sets should be preferred over allowing more time for testing.

3.3 Improvement Stage

In this stage, with the objective to increase Test-Duo’s capability in detecting bugs in the SUT, the test materials are refined according to the classification and the corresponding improvement strategies identified in the mutation stage. Test-Duo is engaged again to test the SUT with the purpose to find unrevealed faults with the improved test materials. Existence of any test failure indicates either that the SUT has a bug or that the generated functional tests are incorrect. Depending on the cause of failures, the SUT is fixed (until the captured failing test cases pass) or the test materials are refined again so that the generated functional tests are correct. The improvement stage activities are repeated until no test failures are found by Test-Duo with the current test materials, which is exactly the precondition of entering the mutation stage for the next iteration of improvement.

4. CASE STUDY

4.1 Design of the Case Study

In this case study, the Test Director is implemented with Prolog [5] while the Test Driver is implemented with ROBOT framework [4], which is also used as the automatic testing platform. The quality of functional tests generated with Test-Duo is measured with mutation score. We use plots of number of surviving mutants against number of executed test cases to illustrate the performance of Test-Duo in killing the mutants. As the number of surviving mutants decreased, the mutation score (and therefore the quality of generated functional tests) increased. The target is to kill as many mutants as possible and kill the mutants as early as possible. The case study setup is described next.

The overall process to apply our approach in conjunction with mutation can be divided into five steps below and as illustrated in the case study:

1. Tester annotates the use cases and prepares test materials using the supporting tool.
2. Tester implements the test keywords according to SUT implementation.
3. Tester creates mutants for the SUT with the mutation generation tool.
4. Tester feeds the use case and the test materials for Test-Duo to generate test cases automatically to kill the mutants.
5. Tester reviews the surviving mutants, refines the use cases and test materials, and repeats the step 4 to kill the surviving mutants if necessary.

How to annotate use case and perform mutation-improvement loop can be found in previous sections. The further information on case study setup is detailed below.
A. The system under test

The SUT is a POS program similar to the one described in Larman [13]. Specifically, the use case “Process Sale” in Fig. 3 is tested. In addition to the ordinary checkout process of tallying the items purchased by the customer and displaying running total before and after discount (see Fig. 9), the POS program has a special feature of supporting multiple (possibly competing) discount rules to be applied to the sale based on promotion and customer identity.

B. Creation of mutation for the SUT

MuJava [18] is used to introduce errors into the POS program, in order to simulate possible errors that can be introduced by programmers into the program text. In this case study, we deal with mutants that have one-place syntactic change (e.g., a “+” that is changed to a “-”, a convention in mutation testing) introduced by MuJava. All the mutation operators supported by MuJava both at method-level and class-level are applied to the POS program. A total of 227 mutants are created from the POS program.

C. How to deal with time budget for each round of testing

For the first round of mutation testing, all the 227 mutants are tested and classified. Therefore, we allow more time in the initial round of mutation testing because the equivalent mutants are included in the 227 mutants. Although techniques exist in detecting the equivalent mutant, however, detecting all of them automatically is not yet feasible [29, 30]. As mentioned before, the equivalent mutants cannot be killed by any test cases because they produce the same output as the original program does. Thus we allowed extra time for Test-Duo to perform the mutation test and classified the equivalent mutants into surviving mutant group automatically to help us to identify them later. The methodology and the effect of detecting equivalent mutants are beyond this research and will not be discussed here. In this case study, we plan to test every mutant for at least 100 test cases in the initial run. If the mutant is killed within 100th test case, the saved time is used to test other mutants that remain alive. Therefore, more than 100 test cases will be generated to test the mutants that are hard to kill. Before the mutation testing, we measure the average time to run a test case to test the SUT and 11.5 seconds per test case is found.
Thus, a total of 227 * 100 * 11.5 seconds or approximately 72 hours, is set for the initial round of mutation testing.

For the remaining round, the time budget is decided with the same rule with the following data: the number of mutants, the average time to run a test case and the minimum number of test cases we plan to test each mutant. Note that the time is expected to decrease since the equivalent mutants are removed and the test materials are enhanced.

D. Tool support for use case annotation

In our previous work, an Eclipse plug-in called Acceplipse has been developed to assist the tester in preparing the test materials [31]. The tester edits the use case via the GUI interface to maintain use case contents, annotate the use case and create data sets. The created artifacts are used by the Test Director to generate the legal moves among use case steps, test keywords, and data sets as a collection of Prolog relations for use in test case generation. For the Test Driver, Acceplipse generates a table containing all the test keywords to be implemented on ROBOT framework. The tester has the responsibility to implement these keywords according to the implementation of the SUT for the Test Driver to drive it.

E. Preparing test materials for Test-Duo

The use case “Process Sale” is annotated with Acceplipse and the annotated use case is shown in Fig. 10. The “Main Success Scenario” section shows the steps of the
main success scenario while the “Extension” section shows all the extension flows and marks them with an E. Each step in the use case is marked with an S and the loop of steps is marked with an L. Indented under each step are zero or more test keywords, each marked with a K. Further indented under each test keyword are zero or more arguments of the keyword. A T-argument is static in the sense that it can be determined before the test case is executed. A D-argument is dynamic in the sense that it can only be computed. Test-Duo randomly selects T-arguments from a data set constructed with Acceplipse, and generates D-arguments by calling the expected result generator, which is supplied by the tester. For the use case “Process Sale”, the data sets created is shown in Fig. 11 and an implementation of the expected result generator is shown in Fig. 12. Each conceptual class in the data sets is marked with a C. Inside each conceptual class are their realization concrete objects and an O is used to represent such objects. Indented under the concrete objects are use case vocabularies (which marked with a V) that form the conceptual class and the realistic value of the corresponding use case vocabularies.

\[
\text{getOracle_total_should_be}(\text{Value}) :- \\
\text{get_SavedItem_byAction('input data to the system', RawItemLists),} \\
\text{calculateSaleItemSum(RawItemLists, Value).} \\
\]

\[
\text{getOracle_total_after_discount_should_be}(\text{Value}) :- \\
\text{get_SavedItem_byAction('input data to the system', RawItemLists),} \\
\text{getSaleItemList(RawItemLists, AllItemList),} \\
\text{getItemCount(AllItemList, [], ItemCountList),} \\
\text{calculateTotalByRule(RawItemLists, ItemCountList, Value).} \\
\]

Fig. 11. The created data sets for the use case process sale.

Fig. 12. Expected result generators in prolog.
F. How Test-Duo kills the mutants

Test-Duo kills the mutants in rounds of testing. In the beginning round, Test-Duo generates the minimum-length use-case terminating test case. In the “Process Sale” use case, the minimum length is 10 since the step number in main success scenario is 10 and all of the extensions merge into the main success scenario without terminating the use case. Arguments required by the test case are filled in by Test-Duo as previously described. The test is then applied to the first mutant. If the test case fails, the failed test case is saved and the mutant is marked as killed. Otherwise, if the test case passes, mutant survives the test case. This process is repeated for each of the remaining mutants using a new set of arguments similarly prepared by Test-Duo. When all mutants are tested, the surviving mutants are collected, and the above process is repeated for next minimum-length use-case terminating test case. After all the test cases with length 10 are tested, Test-Duo increases the length to 11 and tests all the test cases with length 11 according to the applied iterative deepening search algorithm. The whole process is controlled by a script and is repeated until all mutants are killed or a specified duration of time expires.

G. Feedback loop for improving the quality of functional tests and test materials

At the completion of a round of mutation testing, the killed and surviving mutants are analyzed using the classification framework in section 3, which leads to improved use cases, extended annotations, and enhanced data sets.

4.2 Results

After three rounds, all non-equivalent mutants are killed by Test-Duo. Table 2 shows the results of the mutation testing.

<table>
<thead>
<tr>
<th>Testing round</th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tested</td>
<td>227</td>
<td>179</td>
<td>179</td>
</tr>
<tr>
<td>Non-Equivalent</td>
<td>179</td>
<td>179</td>
<td>179</td>
</tr>
<tr>
<td>Killed</td>
<td>146</td>
<td>162</td>
<td>179</td>
</tr>
<tr>
<td>Mutation score</td>
<td>81.6%</td>
<td>90.5%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testing round</th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent</td>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unreachable by use case</td>
<td>17</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Undiscovered</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A. First round – mutation stage

As mentioned before, the mutation stage is entered if no further bugs can be found
using existing test materials within the given time constraint. In the initial round, a time limit of 72 hours is set to test all the 227 mutants. At the conclusion of testing, 146 mutants are killed while 81 mutants survive. The surviving mutants are analyzed and categorized as described in Table 3. The equivalent mutants are removed from next round of mutation testing to save time. From the mutants that are unreachable by the use case, it is found that the use case omits to include the extension flow “removing items from sale” while the function had already been implemented. Mutants introduced in the code that implements the extension flow “removing items from sale” are unreachable by the test cases. For the mutants that are undiscovered by Test-Duo, it is found that additional test data are required to kill them.

![Mutation Testing with Test-Duo](image)

**Fig. 13.** Mutation testing with Test-Duo: numbers of killed and surviving mutants.

The killed mutants are also analyzed for the amount of time that Test-Duo takes to kill them. In Fig. 13, the cluster of 7 mutants takes relatively more time to kill. As shown in the inset of Fig. 13, each of the 7 mutants needs more than 501 test cases to kill, which collectively take 16416 seconds, while each of the other 139 mutants are killed within 90 test cases, which take 3505 seconds in total. Apparently, a significant amount of time can be saved if these 7 mutants are killed within the same amount of time it takes to kill the other 139 mutants. Upon analysis, it is found that the 7 mutants are hard to kill because they are located in the discount rules and some additional conditions must be satisfied before code implementing the discount rules is exercised.
B. First round – improvement stage

In first round of improvement, we deal with the undiscovered mutants and the mutants that take long time to kill. By analyzing the location of the syntactic change applied in source code of each mutant in these two categories, we found that a large portion of the mutants are located in the logic of applying discount rules, which required relatively more items to be purchased in a sale. According to the use case descriptions, we need the scenarios that repeatedly purchase the same item several times to trigger the SUT to execute the code where the syntactic change located. To shorten the time required for their detection, the use case is re-examined. In particular, it is observed that by resuming a suspended sale with a large number of items in the purchase list, the discount logic can be enacted by purchasing a small number of additional items after the sale resumes. Thus, an improvement for data sets is done by adding a new suspended sale which has many items in the purchase list (see Fig. 14).

C. Second round – mutation stage

Since the equivalent mutants are identified and removed in round 1, only non-equivalent mutants (179 mutants) are tested in this round. After collecting the testing result in round 1, the average run time of a test case is updated to 12 seconds and we found that most of the mutants are killed within 40 test cases as shown in Fig. 13. A minimum testing time of 179 * 12 * 40 seconds (23.9 hours) is needed. Thus, an estimation of 24 hours is set as time budget in the second round of testing. After the testing, all mutants that fall in the category of undiscovered are killed and only mutants that are unreachable by use case survive. Furthermore, the time spent by Test-Duo in killing the mutants is improved. Test-Duo now takes only 786 seconds in total to kill the 7 mutants in comparison to 16416 seconds in round 1. The accumulated time spent by Test-Duo on the killed mutants is reduced as illustrated in Fig. 15. In round 1, it takes 19921 seconds to kill 146 mutants while round 2 takes 5081 seconds in killing 162 mutants, which shows that more mutants are killed while less time is spent. Note that the time used on the surviving mutants is not accounted.

D. Second round – improvement stage

In the second round of improvement, we deal with the mutants that are unreachable by use case. Use case is refined to describe “removing items from sale” and a new test keyword “remove item for sale from list” is added accordingly; see Fig. 16. The expected result generator is updated by the tester as well in order to take into account of the newly covered functionality. It is interesting to note that after the test materials are up-
dated and Test-Duo is performed to test the SUT, a real fault of the POS program is uncovered. Specifically, when the total price of the purchase items before discount reaches a specific level, e.g., $300, a discount is applied immediately to this transaction. However, the POS program fails to cancel the discount if items are removed from the sale and the total falls below the level of applying discount. This fault is revealed because the use case description and annotations related to “removing items from sale” are added in this round.

E. Third round – mutation stage

After the revealed bug is fixed and no more faults can be found by the updated test materials, the mutation stage is re-entered with mutants regenerated for the changed code. In this round, all non-equivalent mutants are tested with the updated test materials and a time budget of 24 hours. All of the 179 non-equivalent mutants are killed before the 200th test case (see Fig. 13) with an accumulated time of 8126 seconds (see Fig. 15) that is within the time limit. Comparing to the second round of mutation testing, both the maximum number of test cases and the testing time to kill all mutants increase because a new extension is added; this increases the number of test cases generated. Thus the extra
time spent is required to cover newly discovered omissions.

No further improvement stage is performed and the overall process stops because
the mutation score of 100% has been reached and testing time of 8126 seconds is ac-
ceptable to us. At the completion of mutation testing, a total of 179 non-equivalent mu-
tant-killing test cases are collected. These test cases are executable ROBOT framework
test cases that cover the common errors that a development might make. They can further
be used as the minimum set of test cases to verify the system correctness.

4.3 Observations

(1) Test-Duo can generate high quality functional test cases when the tester continuously
improves use cases, annotations and data sets. In this case study, a mutation score of
100% has been reached after three rounds.

(2) The improvements performed against the use cases can help to identify the incon-
sistencies between the SUT and its requirements captured in the use cases.

(3) Mutation testing can help to identify real bug in the process of attempting to increase
mutation score. A real bug has been found in the improvement stage of the second
round in our case study.

(4) By applying mutation testing with Test-Duo, a set of test cases can be identified au-
tomatically for verifying the SUT and further regression testing.

5. DISCUSSIONS ON APPLYING THE PROPOSED APPROACH

5.1 Early Development of Testing

By applying the Test-Duo framework, the design of the tests can be performed in
early stage of the development. Once the use cases are created, the tester can annotate the
use cases to get the test materials and prepare test keywords for the SUT. With the evo-
lution of the SUT development, the test cases created previously can be used as regres-
sion test cases when new use cases are implemented. Since testing is performed in the
early stage of development, the change of the requirements should be considered. In our
approach, the Test-Duo generates test cases based on the annotated use cases. When the
requirements change, the test materials in the use cases are revised to reflect the new
scenarios on how the SUT behaves. The revised test materials will enable Test-Duo to
generate the new test cases accordingly.

5.2 Reusability of Test Keywords

In our approach, the Test-Duo generates keyword-driven test cases. The use of
keywords is separated from their implementation to make them reusable. On the one
hand, if the implementation of the SUT changes, it is unavoidable that the keyword im-
plementation needs to be modified. However, we do not need to modify the test materials
if the use cases on how the SUT is used do not change. For example, if the presentation
of the total in POS system changes from using a text field to using a label, what we need
to do is changing the keyword implementation to get the total value from a label instead
of a text field. On the other hand, if the use cases change, we can reuse the test keywords
if the implementation of actions that the keywords deal with does not change. For example, if we extend the POS system to input the quantity of the item to be purchased, what we need to do is update the use cases and implement new keyword for the quantity attribute. The original keywords can be reused. Furthermore, since the test keywords are designed for the SUT, they can be reused in other use cases if these use cases share some interactions between the SUT and the actor.

5.3 Reusability of Data Set

The conceptual classes of the data sets are created with the domain modeling guidelines [13] for the domain of the SUT. These data sets can be shared among the use cases since they describe the use scenario of the same SUT. However, it is possible that the conceptual classes need to be extended when new use cases are annotated. In this case, not only the conceptual classes are changed but also the concrete objects need to be modified. Designing generic conceptual classes for different use cases can decrease the modification frequency of the data sets and increase the reusability of them. The methodologies on developing good conceptual classes can be found in other research [13].

5.4 Known Limitations

For the Test-Duo framework, the generated functional test cases are highly dependent on the complexity of the use case and the quality of data set. Since Test-Duo traces all possible test scenarios of the use case, the number of test cases can easily explode if the use case structure is complex, e.g., multiple extensions branching from the main success scenario, extensions branching from extensions, multiple loops, and so on. Currently, Test-Duo does not support setting priorities to different use case scenarios. As a result, it can waste a lot of time to test the scenarios of little value but spending little time on the frequently used scenario. Finally, test data are randomly selected from the data set and filled in to the test cases. Even if the data set contains concrete objects whose combinations can lead to the failure of a test case, such a combination may be difficult to come by through random selection. A more clever way of selecting test data seems necessary.

In the mutation-improvement loop, it is possible that the mutant score will not reach 100%. This is because only the scenarios in the use case are tested; mutants with syntactic changes located at the code not covered by the use case cannot be killed. Although the tester can refine use case to cover more scenarios, the use case description could become overly complex. An alternative is to customize test cases manually to kill specific surviving mutants if high mutation score is needed.

6. RELATED WORKS

A large body of literature exists on generating functional test cases from use cases [32-36]. In this section, three groups of related works are reviewed.

The first group of existing methods does not directly process use cases but focus on creating intermediate models from use cases to enable test case generation. In the method proposed by Heumann [37], scenarios of exercising an use case are first obtained by the
tester by enumerating the possible execution flows in the main success scenario and the extensions. For each use case scenario, one or more test cases are created by the tester, who does so by re-reading the use case description and identifying the conditions, input data, and expected results required. These test cases are collected into a table called the test case matrix that describes the test in texts. Fröhlich and Links treat the test generation problem as a STRIPS planning problem [38]. They propose to systematically transform use cases into UML state charts, state charts into STRIPS operators, and STRIPS operators into functional test sequences through Graphplan [39].

Though the test matrix and test sequences can be used to verify the SUT, it can incur a significant amount of effort to perform the test repeatedly without automation support. Our proposed method follows the same idea in generating test cases from a use case but introduces the following elements to automate the process: (1) annotations adorned to preconditions, post-conditions, and steps in the use case to enable generating the test scenarios; (2) prepared data sets and interface for hooking up expected result generators to enable filling in the arguments of test actions; and (3) the generated test cases to be readily executed by the underlying automatic testing platform.

El-Attar and Miller proposed a use-case driven approach for developing executable acceptance tests using use case models, robustness diagrams and domain models [40]. The high-level acceptance tests (HLAT) are created by the tester examining the use case descriptions and identifying test scenarios, required inputs, triggers and expected output. In the meantime, robustness diagrams are created by the tester through robustness analysis in which the use case model and the domain model are consulted to identify object-level information corresponding to the inputs and outputs. FIT/FITnesse based executable acceptance tests are then created manually from the HLAT and object level information.

The proposed method shares with El-Attar and Miller’s method in beginning with use cases and ending with executable test cases but achieves a higher degree of automation, most notably in Test Director’s automatic assignment of test data from the data set while running the test case. On the other hand, our method has a different scope in focusing on functional testing, which is part of acceptance testing [41].

The second group of existing methods focuses on the part of the test case generation process by directly processing use cases. Carvalho et al. present a strategy to generate test cases from natural language requirements which are written according to a Controlled Natural Language to avoid textual ambiguity. Each syntactically valid requirement is mapped into a semantic representation from which an SCR (Software Cost Reduction) specification is derived. The T-VEC tool is then used to generate tests from SCR [42]. Although not specifically restricted to use cases, their approach can obviously be applied to process requirements written in use cases. Mingyue and Zuohua propose to write the use cases in a restricted yet acceptable language style. An Extended Finite State Machine (EFSM) are extracted from the use cases and the test cases are generated from the EFSM [43]. Though the use cases are written with nature language, a set of sentence styles are introduced to write the use cases. Somé and Cheng present an approach for system-level test scenarios generation from textual use cases which are written with restricted natural language syntax through a hierarchy of tuples leading from use case to step and to operation [44]. Control flow-based state machines (CFSM) are generated from each use case while global CFSM is created to represent the sequencing relations between use cases. Test cases are manually derived from test scenarios which are path
sequences in CFSM. Gutiérrez et al. propose to generate test cases by translating the use cases written in Navigational Development Technique (NDT) format [45, 46]. The use cases are first translated to UML activity diagram by any tool that supports NDT. The activities in the diagram are created from the use case main success scenario, conditions and alternative steps to represent the different scenarios or instances of the use case.

In contrast to these existing methods, our proposed method uses annotations adorned to the use case as an extra aspect. Since the annotations rather than the use case texts are processed for generating test cases, the proposed method does not introduce syntactical restrictions as to how use case description should be written.

The third group of existing methods generates test cases from use case-derived artifacts. Inspired by Briand and Labiche’s UML-Based approach to system testing called TOTEM [47], an approach to generate functional test cases from use cases by applying contracts is proposed by Nebut et al. [48]. In the first phase of this method, the UML use cases are extended with contracts. In the second phase, contract-enhanced use cases and their attached sequence diagrams are translated into executable JUnit test cases. Instead of being applied before and after a use case is performed, the contracts are used to infer the ordering of use cases that the system should offer. The valid sequences of use cases extracted that satisfy the given coverage criteria are called test objectives. For each test objective, test scenarios are automatically generated by replacing use cases with corresponding sequence diagrams. Events in the sequence diagrams are translated as function calls to invoke underlining implemented method.

Our proposed method is similar to the method of Nebut et al. in achieving a high degree of automation in test case generation and in introducing extra elements to adorn the use cases for the task of automation. On the other hand, our proposed method differs in generating black-box test cases rather than white-box test cases. Specifically, in generating test cases, use case contents are used as the source in our proposed method while sequence diagrams are used in Nebut et al. As such, the two methods can be applied without contraction. Lastly, our proposed work does not introduce formalism in describing a use case, which is viewed by Nebut et al. as a limitation to their approach by introducing contracts specified with OCL [49].

In this paper, we assume that human involvement is unavoidable in preparing high quality test cases. Thus, we leave the essential steps for tester to accomplish manually but aid the tester in steps that are highly amenable to automation. In the proposed method, the essential steps the tester performs include the use case annotation, test data set preparation, and executable test keyword creation. Except these steps, our approach provides a high degree of automation for the labor-intensive and error-prone steps which include systematically creating test scenarios, filling test steps and checkpoints in each test scenarios, and performing the concrete test execution. Besides the difference on how the test cases are generated, our approach also introduces mutation testing as a measuring mechanism to check the quality of the generated test cases, which creates an improvement loop for the tester to check test cases quality.

7. CONCLUSION AND FUTURE WORK

With the advances in research and practices in software testing methodologies and tool supports, the tasks of executing test cases and collecting test results have achieved a
high degree of automation with the widespread use of automatic testing platforms. However, despite the progresses already made in many aspects of test automation by the numerous efforts of researchers, the task of generating test cases remains mostly a manual process. In this paper, the Test-Duo framework has been proposed for further automating the task of generating functional test cases that are ready for execution on an automatic testing platform. With Test-Duo, the tester focuses on the tasks of refining and annotating use cases and preparing test data sets with tool support, leaving the tasks of generating the actual test cases to the Test Director component of Test-Duo. Since a large number of test cases can be generated, the Test Driver component of Test-Duo works closely with the Test Director to marshal execution of the test cases on the underlying automatic testing platform. Furthermore, the tester can systematically improve the quality of use cases and the performances of annotation with test data sets in finding potential SUT bugs. The improvement is accomplished by iteratively subjecting the generated test cases to mutation testing. Both the mutants that are killed and those that remain alive are analyzed, and corresponding strategies are applied to increase mutation score and to reduce the time required to kill the mutants. The effectiveness of this improvement process has been demonstrated in the case study. The use of mutation testing as a means for improving the quality of generated test cases by Test-Duo is an answer to Yue and Harman’s call for more works to be done in the direction of generating test cases to kill mutants and improving the generated test cases based on the associated mutation analysis [8].

A number of directions for future research are possible. Here we mention three of them. First, in the Test-Duo framework, the Test Director randomly picks data items from the data sets for use as arguments in the test cases. As has been shown, the data sets should contain items that enable a data combination for exposing a fault in the SUT while running test cases. While we have shown that data sets can be improved by incremental refinement, strategies for constructing and selecting effective data sets can help as well. Secondly, while mutation testing has been used as the measurement and improvement framework, other measurement mechanisms such as various coverage criteria should be explored as well. Lastly, mutation testing can involve a large amount of computation, especially when a large number of mutants are used. The architecture of Test-Duo can be extended to apply parallel and distributed computing techniques by deploying multiple instances of the Test Driver over clusters of computers to run the tests in parallel.

REFERENCES

1. K. Beck, Test-Driven Development: By Example, 2003, Addison-Wesley, MA.
5. I. Bratko, Prolog Programming for Artificial Intelligence, Addison Wesley, 2001, MA.
44. S. S. Somé and X. Cheng, “An approach for supporting system-level test scenarios


Chin-Yun Hsieh (謝金雲) is an Associate Professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. He received his MS and Ph.D. degrees from the University of Mississippi and the University of Oklahoma, respectively, both in Computer Science. His research interests include object-oriented software engineering, pattern languages and software testing. He is a member of the Software Engineering Association of Taiwan (SEAT).

Chen-Shin Tsai (蔡振欣) is a Ph.D. candidate at the Department of Computer Science and Information Engineering, the National Taipei University of Technology, Taiwan. His research interests include software testing and object-oriented analysis and design.

Yu Chin Cheng (鄭有進) is a Professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. Dr. Cheng teaches and conducts research in object-oriented systems, agile methods, software requirements and specifications, and software architecture. He is a member of the IEEE Computer Society, the ACM, and the Software Engineering Association of Taiwan (SEAT).