Computer Science Journal of Moldova, vol.**20**, no.1(58), 2012

Using Test Case Mutation to Evaluate the Model of the User Interface

Izzat Alsmadi

Abstract

Mutation based testing is used to discover new possible errors in software applications. This is since in this testing approach, intentional incorrect lines of codes are injected to check the software ability to produce results that are different from the correct or original code. In this paper an automatic technique to generate valid and mutant test cases is proposed and developed. In most mutation techniques, one or more values or parameters in the specification, code, model, etc are intentionally modified and then test cases are generated to see if injected modifications can be detected. However, in this paper, test cases are mutated (i.e. mutants are generated from the test cases) after they are generated from the GUI model. Mutations are then applied to the GUI model to test its ability to kill those mutants by rejecting them. Typical to mutation testing, the goal of this approach is to discover possible errors or problems in the program that may not be discovered by other methods. A robust model is expected to differentiate between a valid and an invalid sequence of events. An automatic execution and verification technique is also developed to evaluate the test cases that were rejected by the model and calculate coverage based on the number of rejected test cases to the total number of test cases. Results showed that in user interfaces, and based on the nature of the mutation process implementation, mutation can find new areas or types of errors that may not be found using other approaches of testing.

Keywords: Mutation testing, Test case generation, test case execution and verification. Random test case generation, and GUI modelling.

1 Introduction

It is widely acknowledged that testing activities consume a significant amount of software project resources. This is why research projects in software testing focus on aspects that can reduce those expenses while maintain or improve coverage. Test automation techniques are used to achieve this goal. In order to use test automation, Artificial Intelligent (AI) algorithms are used to replace or simulate tester activities. Those activities include: test case generation, execution and verification. Mutation is a surplus testing activity used in general to improve test case generation and verification effectiveness. This is accomplished by changing a small part of the code or the specification. Test cases are then applied to see the test cases that can kill (i.e. discover) those mutants. In this paper, mutation is used to evaluate the reliability of the GUI model. In traditional code mutation processes, mutation coverage can show whether test cases would expose the use of wrong operators and also wrong operands. It works by reporting coverage of conditions derived by mutating (i.e. substituting) the program's expressions with alternate operators, such as "less than" substituted for "more than". In the traditional mutation, mutation is occurred to the code or the specification and test cases are expected to discover this mutation. In this paper, the process is reversed. Mutation occurs in test cases and the GUI model is expected to discover those mutations. Such approach may fall under model based testing techniques where the GUI model is tested for its ability to kill (i.e. reject, in the scope of this paper) wrong test cases.

Why would someone apply mutation to test case generation?! In GUI testing, most GUI components have one main event interaction. For example, a button main even interaction is the "double click", the textbox main interaction is "entering a text", the option item main interaction is selecting one or options, etc. A GUI abstraction model that considers the GUI components, their attributes and association with each other along with one main event for each component is developed [9, 10]. As such, the abstraction model considers both GUI structure and event models. In this model, test cases can be generated directly

```
<GUI-Forms><Root>GUI-Forms</Root>
<Parent-Form>frmDataDisplay</Parent-Form>
<Name>Settings</Name>
<Parent-Form>frmDataDisplay</Parent-Form>
<Name>Program</Name>
<GroupBox>
<Parent-Form>frmDataDisplay</Parent-Form>
<Name>GroupBox2</Name>  </GroupBox>
<ListBox><Parent-Form>GroupBox2</Parent-Form>
<Name>lstViews</Name> </ListBox>
```

Figure 1. A simple GUI structure sample generated from an application in an XML format.

from the GUI structure file. Figure 1 shows a simple screen shot sample generated from an application for a GUI structure. The XML file (generated automatically from the application at run time using reflection; a reverse engineering process to discover the program GUI components from its executable) contains all GUI components along with each component parent. Test cases are then automatically generated from this model through traversing through GUI paths starting from the entry point to an end or leaf point. For each component, test cases are considering the component default event (in order to generate the test case that will be executed automatically).

Is it significantly useful to make an effort to inject errors in test cases and then make extra effort trying to find them ?! A mutation in a test case means that we will try to execute a test case with invalid sequence or combination of GUI components. Such sequence should not be executed successfully. This is somewhat similar to the specification based testing that tests the application using valid and invalid inputs. The application is expected to accept valid inputs and produce consistent results while rejecting invalid inputs and halt the execution. The next section introduces the related work. Section 3 lists the goals of this research and describes the work done toward those goals. Section

4 presents the conclusion and future work.

2 Related Work

In this literature survey, several relevant papers are discussed. Those papers discussed using mutation for evaluating coverage and test case effectiveness. All papers listed as references in this paper focus on generating mutation operators based on one aspect of software products and then evaluate test coverage or effectiveness from this mutation process. The major difference that distinguish one paper from the others is the software aspect that mutation operators are generated from (e.g. requirements, software model, code, state diagram, etc).

One of the prominent researchers in the area of testing in general and mutation testing in particular is Jeff Offutt at George Mason University (cs.gmu.edu/ offutt). He has several books, book chapters and relevant papers sole or with friends and students. He also developed the widely used mutation tool muJava (http://cs.gmu.edu/ offutt/mujava/). Examples of some of those contributions include: [1,2,3,4,5,6,7,8]. These papers discussed developing and using mutation tools such as Java and Mothra. They also discussed Mutation operators and using mutation in source code, Web applications and object oriented code. Coverage (e.g. code, and path) was a criterion to evaluate the effectiveness of mutation against it. Mutation can be divided based on the software stage where it occurs, or based on the software product component(s) upon which the mutation process occurs. For example, there are several papers that discuss: source code mutation (e.g. Java, object oriented code), Windows or Web mutation, test cases' mutation, database, integration testing, design or requirement mutation. The selection of the papers in this literature review of related work is based on selected variations of these different mutations.

In [1], Choi et al presented one of the earliest mutation based testing environment. Other examples of mutation tools include: Java, Clipse, Javalanche, Jumple, Certitude, Jester, Proteum, and SQLMutation.

In our paper, a new mutation tool is developed for the particular mutation in user interface components. The Mothra testing project was

initiated in 1986 by members of the Georgia Institute of Technology's software engineering research center. Mothra is written in FORTRAN and consists of a collection of individual tools, each of which implements a separate, independent function for the testing system. Examples of some of its mutation operators include relation operators' replacement. In original Mothra, the tool converts the tested code to an intermediate code in order to execute its mutated version by an interpreter.

In a recent paper [2], Mateo et al proposed mutation operators that are somewhat related to GUI components as some of these operators were trying to evaluate whether a component is interchanged with an earlier version of the same component which is something that may occur frequently especially with an evolving application. This system level mutation is a continuation for a work done in this area previously by Delamaro et al [14, 16, 17, and 24] work on MuJava. On the GUI level, the mutation operators proposed consider a small subset of the possible GUI mutants such as GUI components' position or order interchange, component deletion and modification. However, as the paper had a large scope, extensive evaluation and implementation of these mutations were not mentioned. As our paper came as an extension of a GUI test automation tool [9, and 10] which includes the automation of all GUI testing activities: generation, execution and verification, the tool is utilized and extended to generate GUI mutants along with implementing the ability to automatically execute these mutants and evaluate their results.

In [3], Offutt et al described how to use the information of equivalent mutation for the problem of some paths' feasibility. An equivalent mutant is the one that will always produce the same output as the original program, so no test case can kill it. This affects the mutation score and causes it always to be less than complete coverage. Earlier, in his dissertation, Offutt proposed using Constraint-Based Testing (CBT) for detecting equivalent mutants [5].The paper presented a method to detect equivalent mutants in code through using constraints. These constraints are applied on the input domain to narrow its scope. GUI mutation has some possible equivalent mutation operators that will be

discussed in this paper.

Li et al evaluated mutation as a coverage criterion in comparison with other testing coverage criteria such as path, edge, etc. [6]. Coverage is a test case metric that is used to measure the ability of a particular test approach to cover one or more aspects of the software code that may include: statement, branch, path, etc. coverage. The study focus was on unit testing and showed that mutation can actually be more effective in terms of coverage from many other test criteria. Unlike mutation score, mutation coverage calculates the number of faults that can be detected using the mutation process. In our GUI testing approach, the GUI is serialized from the actual application dynamically using a reverse engineering process. (i.e. .NET reflection). As a result, the GUI model is represented by an XML file which includes the GUI components, hierarchy and attributes. Mutations are created based on the XML file and applied on the actual GUI during the execution process. However, since a GUI test case looks like a GUI path (e.g. File,Save,Exit), mutation can be also applied on or generated from the test cases. In typical cases, the program is mutated and the test cases are used to detect this mutation. However, in GUI, it is possible to reverse the process through generating a mutated test case (that may include for example an invalid GUI component) and then execute it on the application where its failure is an indication that the mutant is killed. If the mutant test case failed then this is a possibility of two: either the mutant test case is equivalent and looks like a normal test case to the GUI, or the test case is killable but the current state of the GUI failed to kill this mutant.

Offutt et al discussed a selective mutation process to reduce the expenses of mutation as the number of possible mutation operators for even a small program can be significant [7]. A selective mutant can be selected from many mutants if they produce the same results. Lee et al. presented a Web based scenario level mutation based on interaction scenarios written in an Interaction Specification Model (ISM) [8]. ISM is an XML based interaction constraint language. Message mutation can be applied in a wide range of applications that use messaging such as distributed systems, networks, etc.

Hierons paper represents another paper in specification or modeling mutation [24]. The paper discussed Finite State Machines (FSMs) mutation based on the basic elements of FSMs which are: states, events and transitions. In FSM, each state is recognized by preconditions which represent the constraints that are required to occur or be true in order for the transition to the target state to occur, and post conditions that represent the expected results from the state transition. In relation to this subject, we are planning to extend GUI mutation in future to cover GUI state based mutation aspects.

Other papers such as [11] and [14] discussed using mutation for evaluating states coverage. Examples of mutation operators for finite state machines include: event, arc, or output missing, extra or exchanged. For user interface mutation this also can be considered for GUI events interaction, however, the focus of our mutation operators here is on the GUI structural mutation.

There are some other papers such as [13 and 18] that used the mutation process as a technique for validating a particular testing algorithm or approach. Bradbury et al [13] used a subset of the Concurrency Mutation Analysis (ConMan) operators that are discussed in the authors' earlier paper [23]. However, by larger, model checking is a testing area where mutation is not thoroughly investigated. In this paper and in mutation research papers in general, one weakness of such studies is that same authors are the one who usually create mutations and create tests to detect them. If the goal of mutation is to try to imitate real life bugs and test abilities to detect them, the creation of mutation and the detection process should both be independent from each other.

Papers from [15, 16, 17, 18, 19, 20, 21 and 22] include examples of using mutation in other software artifacts such as: Web, object orientation and aspect oriented programming.

3 Goals and Approaches

3.1 Using Test Case Mutation to Evaluate the Model of the User Interface

Mutation is used in testing for various purposes. It is first used to inject faults into the system and measure the system or the test cases' ability to catch those mutants. As a result, mutants can indicate the effective test cases through their abilities to detect errors which allow us to eliminate ineffective test cases and improve test effectiveness. Mutation can be also used to indirectly generate test cases. Mutation testing can be also used to indirectly verify requirements. In mutation, three steps are accomplished: First mutants are created according to mutation operators; second program is executed with normal and mutant inputs. Finally verification and coverage analysis is implemented based on the percentage of mutants discovered. In this paper we will present mutation for the test cases. In our GUI model, each test case contains a sequence of GUI components. This means that the program executing those test cases will interact with those components consecutively with the right or typical type of interaction (e.g. a button click, a textbox type text, an option list option select, etc). Default types of interactions are defined for each control type. As such, the mutations that will be evaluated first are changing one component or widget in each test case. From previous knowledge, we know that if the mutated control is changed to a control in the same level, the new test cases can still be executed (however, it should produce a different behaviors). As such, we can divide the expected behaviors from test cases' mutation into 3 levels:

- It is expected that the majority of mutated test cases should be rejected as they will produce invalid test cases that will not fully and successfully be executed. Execution and Verification tool should be able to distinguish that such mutant test cases produced different results relative to the original ones.
- Some mutations will pass the validation process and produce a valid test case. However, they will produce different results or

behavior relative to the original test cases.

- It is expected that few mutations will not be killed at all as they will be valid and produce identical results compared to the original test case or the different behavior can't be distinguished or observed.

3.2 The Execution and Verification Process

Implementing and verification process for testing is one of the challenging processes that have several obstacles. An algorithm is developed to read test cases one by one and execute them on the actual application. Each GUI component is then tested to see if it is successfully executed or not. Once all test case components are successfully executed, the test case suite passes the execution and verification process. In such processes many problems can occur. Timing is one of the problems where one form or web page needs to wait for another execution to finish while it is expecting it earlier. Synchronization and dealing with multithreads are also other examples of such problems. In order to focus on evaluating and comparing original test cases with mutants, any test case from the original test fails the execution and the verification process will be eliminated. The importance of evaluating this execution and verification effectiveness is that it can test our mutation process by measuring the ratio of killed (i.e. invalid) mutants to valid ones. Any invalid component should not be executed successfully. Figure 2 shows a sample log of the execution and verification process output. The test list count shows the input GUI components to the execution process and the test execution count shows the number of GUI components that were successfully executed. The difference between the two numbers (i.e. test list count – test execution count) represents the number of GUI components that fails in the execution process.

Figure 2. A sample output from the GUI components execution process.

3.3 Using Mutation in Test Case Generation and Execution

The majority of research papers that discussed mutation focused on code, requirement or model mutation. In this part, we will consider test cases' mutation. If a system is expected to accept valid test cases, in principle, it should also reject invalid test cases. This is the main assumption that the paper hypothesis is based on. The GUI model will be tested based on its ability to reject invalid test cases. Mutation is used to make some test cases invalid and test the system ability to reject and catch those mutations.

In specification based mutation, each specification element is replaced with its possible alternatives. A test case that is able to detect the difference between the original specification and the mutated one will kill (i.e. discover) the mutant. Similarly, in code based mutation, a code element (e.g. ">"; the "larger than" symbol) is replaced by one of its possible alternatives (e.g. >=, <, <=). If none of the test cases in the test suite was able to detect the difference in behavior between the original and mutant code, this means that this mutant is not reachable by any one of the test cases. Another possible reason is that it is possible that the mutant is reachable but shows a similar external behavior relative to the original code.

A test case mutant remains live either:

- Because it is equivalent to the original test case and the application cannot tell the difference between the original behavior and the new one. They could be functionally identical although syntactically different. (i.e. equivalent test cases).

- Or, the program is incapable to kill the mutant. This means that the different behavior is not propagated to the external interface. Those summarize the three conditions to kill a mutant: reachability, infection and propagation.

Test coverage can, therefore, be measured according to the fraction of dead specification or code mutants.

Coverage = Number of mutant test cases discovered / the total number of mutant test cases.

This definition of coverage is somewhat new and is a direct indicator for the GUI model quality. The complete coverage in this approach equals to killing all non-equivalent test cases. Typical coverage types evaluated in testing scope include: requirement, code, statement, branch, path, etc. coverage.

In traditional mutation situation, new test cases are added to kill the mutants. In this approach, this indicates a problem in the GUI model that should be addressed (i.e. why it could not reject an incorrect test case sequence). As this is a model based testing, mutation modifications considered here are only those that are related to the GUI structure. Mutations that are related to the specification such as: invalid user inputs based on boundary values and equivalent partitions are not considered in this research as they will affect the code and not the GUI model.

In this research, the mutation evaluation process is reversed. In the traditional mutation process, the code, or the specification is mutated and the test cases are fixed. It is expected that those test cases can show the difference between the original code or specification and the mutated one. In this research, the code and the specification are fixed and the mutation is occurred in the test cases. It should be mentioned however, that we are not testing the test cases. This type of test case mutation can be classified as a model based testing approach. The GUI model is expected to discover and kill the mutants. As such test adequacy can be measured by the number or the percentage of the failed test cases. Initially, the approach requires calibration to make sure that all original test cases pass (or else take the number of the successful test cases as the denominator). In the first stage before applying test case mutation, all test cases in the suite must be tested to make sure that the GUI model accepts and validates them.

The typical definition of coverage is calculated through the code or specification percentage that is tested through the test cases. In this research, coverage (which is the number of the test cases that fail to the total number of mutated test cases) reflects other quality attributes

in the system. Opposite to this research approach, some good quality attributes of the system such as robustness express the system dynamic range and its ability to tolerate inputs or user mistakes. However, tolerating wrong user inputs should not be mistaken with accepting wrong user inputs. A fault tolerance application may not crash if a user input an invalid input; however, it should reject such input and stop further program execution. This is the quality attribute that this approach is trying to discover in the program; testing its ability to distinguish a correct input from an incorrect one. As the focus of this paper is GUI testing, we will survey some of possible incorrect inputs that an application may experience.

3.4 The automatic execution and verification

Despite the fact that the subject of this paper is test case mutation in GUI models, however the automatic execution and verification process is important in order to evaluate the validity and the value of the proposed mutation operators.

The need for this automatic process was necessary to check whether the GUI model will accept or reject the applied test cases. The main problem was that we are not simply trying to measure expected and actual numeric values which makes the automatic verification process simple. In this approach, there is a need to verify the GUI state before and after executing each test case. Building a research tool to do such tasks may not be easy. There are some known commercial tools such as IBM Rational Robot which may have the capability to do such complex processes. Most commercial tools are using the record/replay methods and few of them use the object data approach that is adapted in this research.

The algorithm we developed to accomplish the automatic execution and verification process depends on using reflection and the fact that managed code includes GUI control details in their executable. The process will use the reverse engineering reflection method to get all GUI controls, their associations and attributes to the memory in order to validate the actual and mutated test cases on them. However,

validating each test case as a one unit was impossible as the process will simply get the GUI controls from the test cases one by one and see if they exist in the GUI or not. The alternative was to consider that if any control fails in a test case, the whole test case will be assumed fail. However, we decided to go with the first option and hence calculate effectiveness based on the controls rather than the test cases. However, the alternative can be later considered to see which approach provides more realistic results.

3.5 The mutation tool

As an extension for GUIAuto [9, 10] which is a test automation tool developed previously by the author, the tool is developed to execute all mutation process activities. The tool first uses reflection (a reverse engineering process to assemble the application components from its executable) to extract all GUI components and their data to an XML file. Other test automation activities such as test case generation, execution and verification can be triggered based on several algorithms and techniques. In the developed tool, GUIAuto is extended based on the GUI mutation operators that will be described later. Those operators can be generated and executed automatically. The focus of the developed mutation tool is on GUI components mutation ignoring the code behind or the actual code in the GUI events triggers.

3.6 The case study

Several small size open source applications are selected for the evaluation in this study. There are two conditions in the selection of those open source codes. The first one is that since the tool uses reflection to serialize .NET managed code and extract all GUI components from it, all selected applications are .NET managed applications. A managed code is a program (in a .NET programming language: C#, VB. Net, managed C++, or JScripts) that is executed within a runtime engine (such as .NET framework and Java Virtual Machine (JVM)) installed in the same machine. The unmanaged code is an executable program that runs as a standalone, launched from the operating system. The

program calls upon and uses the software routines in the operating system. However, it does not require other software application in order to be used.

The second criteria for selection is that the selected application should contain a reasonable amount of GUI components in many forms or web pages in order to construct a GUI hierarchy and be able to generate different sequences of test cases in several levels. Table 1 shows the summary of the 4 selected AUTs for this specific experiment.

Table 1. GUI Summary of the AUTs

AUT	No. of Controls	No. of Paths	No. of Forms
DBSPY	26	19	2
Notepad	158	176	11
BirdWatcher	23	21	3
CourseReg	89	65	2

Based on the GUI model, and based on its structure and components, several mutation operators are proposed. In order to evaluate proposed mutation effects, an automatic execution process is developed. The GUI model takes all test cases as a sequence and the automatic execution process apply those test cases on the actual GUI. An automatic verification method is also developed to check those test cases that are successfully executed. We define execution coverage metric to be the number of executed controls to the number of input or generated controls. The execution process tries to execute the sequence of controls in each test case one by one. The effectiveness for each test case is calculated. The average for the overall test cases is taken to be the execution coverage.

1. **Mutation Operator Type 1:** Switching GUI components. In this mutation, two components in each test case are switched.

For example, for original test cases of:

1,FRMDATADISPLAY,GROUPBOX3,LSTFIELDS,,SETTINGS

2,FRMDATADISPLAY,MENUSTRIP1,OPENANEWCONNECTIONTOOLSTRIPMENUITEM,,PROGRAM

mutation will be:

1,GROUPBOX3,FRMDATADISPLAY,LSTFIELDS,,SETTINGS

2,MENUSTRIP1,ROGRAM,FRMDATADISPLAY,OPENANEWCONNECTIONTOOLSTRIPMENUITEM

Note the two controls that are replaced with each other. Table 2 shows test execution effectiveness comparison between original and mutated test cases. All effectiveness metrics calculated below in all tables were based on the number of GUI components successfully executed and verified to the number of the GUI controls that were applied. The number of GUI controls in each test case varies from 3-7 controls in the selected applications. It should be mentioned that earlier we define effectiveness as the number of failed (i.e. detected) mutants to the total number of inputs. This is usually the complement of the effectiveness calculated in the tables below. In Table 2, we didn't focus on the effect of changing the test case generation algorithm as this may not be important to the test case verification processes. This will focus on the impact on the execution processes when test cases are mutated.

Table 2. Execution effectiveness for Type1 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.82
2	50	0.816	0.802
3	30	0.86	0.84
4	50	0.71	0.73

The expectation is that effectiveness for mutants should be less than those of the original test cases. This indicates the application ability to reject wrong test cases. In normal situations, switching elements of the test case should cause a test execution failure. The first type does not imply a failure from the GUI model itself. It implies the inability of the developed execution and verification process to detect this type of mutation. The reason is that the execution process segments each test case in its components and then tries to verify the successful execution of each GUI component individually independent from the

other components in the same test case. The automatic execution and verification process executes and searches for every control from each of the test cases in the application assembly (which contains all application GUI components) and verifies its existence and ability to be executed. This is why switching test case elements didn't affect majorly the effectiveness in Table 1 and made the difference in effectiveness negligence. In reality, the automatic verification process is very complex and subjected to several environmental factors. For example, timing and synchronization between the forms or components that are currently visible is very hard to accomplish. For example, the automatic execution robot maybe expecting a button to be clicked at a moment while the opened form is not yet visible or ready. Another problem is the fact that some modules are modeless and do not accept any further commands before closing. The visibility of some GUI components (especially containers) is also a challenge for the automatic verification where defining its visibilities and executing them can be difficult.

The trials to automatically verify the execution of a complete GUI path were unsuccessful. In future, a modified execution algorithm to verify the test case in the same sequence should be implemented to cover this weakness in the verification process.

2. **Mutation Operator Type 2:** Changing the name of one control in the sequence (by adding or removing one letter, for example).

In Type 2, a letter from one control in every test case is removed. The goal is to keep the control entity but change its identity. If each GUI object is defined by its name only, this mutation type should be detected. Location of the mutated or modified letter and the control (from the test case) are selected randomly. Table 2 shows the effectiveness results from this mutation gathered from the actual applications. Results showed that execution effectiveness is reduced to indicate reduction percentage for all controls that were located before mutation only. This can be calculated theoretically by:

$$NewEff = OrgEff - (NoMut/NoControlsTotal)$$

Where $NewEff$ is the effectiveness after mutation, and $OrgEff$

is the effectiveness before mutation, $NoMut$ is the total number of modified controls (through their name) divided by the total number of controls in all test cases applied.

For example, in Table 2, AUT1, $OrgEff = 0.75$, $NoMut = 30$, and as $NewEff = 0.53$, we can find the total number of controls in all test cases which will be 3000/22 or about 136 controls (average controls in each test case = 136/30 or about 4.5). However, this theoretical value is assuming that all mutated controls are undetected and all un-mutated controls are detected in the same way as it was before. Table 3 shows variations of the results between the 4 tested applications.

Table 3. Execution effectiveness for Type 2 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.53
2	50	0.816	0.75
3	30	0.86	0.64
4	50	0.71	0.468

3. **Type 3.** Changing the name of every control in the sequence (by adding or removing one letter, for example). In an extension to type two, and in order to distinguish between a node (i.e. control) failure from a path failure, in this mutation every control name in the test path will be modified by changing only one letter. Table 4 shows the results of applying this mutation.

Table 4. Execution effectiveness for Type 3 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.0
2	50	0.816	0.03
3	30	0.86	0.0
4	50	0.71	0.0

In Table 4, changing all names of controls should bring all test

cases to a complete failure. The few exceptions occur in rare cases where removing a letter from a control change the name to another valid one.

4. **Type 4:** Changing one control from the sequence with another control from the same level. This mutation will bypass some GUI structure constraints where a test case should contain GUI components from the different levels respectively. In some cases, changing this control may change the test case. However, it will produce another valid test case.

Example:

MAINMENU,EDIT,UNDO–TO–FILE,EDIT,COPY

Where Undo and Copy are two controls from the same level. Table 5 shows the results in effectiveness of applying mutation type 4.

Table 5. Execution effectiveness for Type 4 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.73
2	50	0.816	0.815
3	30	0.86	0.81
4	50	0.71	0.82

As expected and explained earlier, switching GUI controls did not affect test case effectiveness as it will modify the test case without invalidating it. In our mutation testing, we are not testing whether the value before and after mutation stays the same. The tests on the GUI model focus on only verifying whether the new mutated test cases will be accepted or rejected by the model. As a result, despite the fact that this type of mutation changes the test case and that the path that it is testing, however, the new test case is a valid one. In some cases as in the last application, effectiveness is improved.

5. **Type 5:** Changing one control in every test case with another one from a different level. In this mutation, one control in each test

case is replaced with a control randomly selected from the pool that contains all AUT controls without observing the location of the newly selected control. Table 6 shows the results from applying this mutation.

Table 6. Execution effectiveness for Type 5 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.64
2	50	0.816	0.812
3	30	0.86	0.80
4	50	0.71	0.82

Similar to Type 1, it is expected that this type should cause a noticeable decrease in effectiveness using mutated test cases. However, this was not the case due to the limitation in the automatic execution and verification process, which verifies the existence of each executed control in the managed code (without considering whether its test case is still valid or not). Rather than lowering the test effectiveness, using mutants improves effectiveness which means that switching the locations of some controls made them more visible and those controls were then located successfully by the execution algorithm.

6. **Type 6:** Deleting a control from a sequence. In this mutation, from each test case, one randomly selected control is removed from the test case.

Table 7. Execution effectiveness for Type 6 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.71
2	50	0.816	0.80
3	30	0.86	0.80
4	50	0.71	0.62

The execution effectiveness should be affected by deleting controls from the test cases solely because of the deletion since the new calcu-

lated effectiveness will be based on the new test cases taken the deletion into consideration. However, all Applications Under Test (AUTs) showed reduction which indicates that when some controls are deleted this may affect the visibility of some other controls.

7. **Type 7:** Adding a control to the sequence. Rather than switching an existed control with another one, in this mutation one randomly selected control is added to each test case. The randomly selected control can be the same added to all or can randomly be selected every time.

Table 8. Execution effectiveness for Type 7 mutation

AUT	No. of test cases	Effectiveness before	Effectiveness after
1	30	0.75	0.76
2	50	0.816	0.77
3	30	0.86	0.77
4	50	0.71	0.75

Similar to the case of: removing a control, adding a control, does not impact effectiveness as this addition is reconsidered when calculating effectiveness. Results showed that although in all mutation cases, test effectiveness after should be less than test effectiveness before, however, as the verification process verifies the controls one by one, the addition of some GUI controls causes the effectiveness to be increased (which may not reflect the test case generation actual effectiveness).

4 Conclusion and Future Work

Test cases are used to detect possible errors and bugs in software applications. In this paper, mutation based testing is used to test applications user interfaces and test if they can differentiate invalid from valid test cases. An automatic tool is developed to automatically generate test cases from applications user interfaces. Later on, and based on the generated test cases, an aspect of one component in each test case

is changed to create test case mutations. Examples of mutations that are considered in this paper were in: changing GUI controls location, name, adding, or removing those controls. An automatic execution and verification process is developed to evaluate the validity of the proposed mutations. The automatic execution and verification processes verify each control individually regardless of its test case. Nonetheless, results showed promising future in the ability of test case mutation to verify certain properties in the GUI model. In mutation original test cases and their results are stored. Those are considered as the baseline for mutation based testing. After generating mutation, to test those mutations, a mutation is said to be killed if its test case result is different from that of the original. The validation of the results considers killing mutants by rejecting them. This makes the automatic verification process difficult due to the difficulty of defining the GUI correct and incorrect states.

References

[1] Choi, B.J., DeMillo, R.A., Krauser, E.W., Martin, R.J., Mathur, A.P., Offutt, A.J., Pan, H., Spafford, E.H. *The Mothra Tool Set.* Proceedings of the 22nd annual Hawaii international conference on system sciences (HICSS'22), Kailua-Kona, HI , USA, (pp: 275-284), vol. 2 (1989).

[2] Mateo, P.R. Usaola, M.P. Offutt, J. *Mutation at System and Functional Levels.* Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 6-10, Paris, France, (pp: 110), (2010)

[3] Offutt, A.J. Jie Pan. *Detecting Equivalent Mutants and the Feasible Path Problem.* Proceedings of the 11th annual conference on computer assurance (COMPASS 96). 17-21 June, Gaithersburg, MD, USA, (pp: 224), (1996).

[4] Praphamontripong, U. Offutt, J. *Applying Mutation Testing to Web Applications.* Proceedings of the 3rd International Confer-

ence on Software Testing, Verification, and Validation Workshops (ICSTW), 6-10 April, Paris, France, (pp: 132), (2010).

[5] DeMillo, R.A. Offutt, A.J. *Constraint-Based Automatic Test Data Generation.* IEEE Transactions On Software Engineering (TOSEM), VOL. 17, Issue 9, (pp: 900), Sep. (1991).

[6] Nan, Li Praphamontripong, U. Offutt, J. *An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage.* IProceedings of the International Conference on Software Testing Verification and Validation Workshops, 1-4 April, Denver, CO, USA, (pp: 220), (2009).

[7] Offutt, A.J. Rothermel, G. Zapf, C. *An Experimental Evaluation of Selective Mutation.* Proceedings of the 15th International Conference on Software Engineering, 17-21 May, Baltimore, MD, USA, (pp: 100), (1993).

[8] Suet Chun Lee Offutt, J. *Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis.* Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), 27-30 Nov., Hong Kong, (pp: 200-209), (2001).

[9] Alsmadi I. Magel K. *GUI Path Oriented Test Generation Algorithms.* Proceedings of Human-Computer Interaction conference (IASTED HCI), Chamonix, France. March 14 - 16, (pp:02), (2007).

[10] Alsmadi I. Magel K. *An Object Oriented Framework for User Interface Test Automation.* Proceedings of The Midwest Instruction and Computing Symposium MICS07, 20-21 April, Grand Forks, ND, USA, (2007).

[11] Hierons, R.M. Merayo, M.G. *Mutation Testing from Probabilistic Finite State Machines.* Proceedings of the Academic and Industrial Conference – Practice And Research Techniques (TAICPART), 10-14 Sep., Windsor, (pp: 141), (2007).

[12] Masud, M. Nayak, A. Zaman, M. Bansal, N. *Strategy for Mutation Testing Using Genetic Algorithms.* Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE/CCGEI), 1-4 May, Saskatoon, Canada, (pp 1049- 1052), (2005).

[13] Bradbury, J.S. Cordy, J.R. Dingel, J. *Comparative Assessment of Testing and Model Checking Using Program Mutation.* Proceedings of the Academic and Industrial Conference – Practice And Research Techniques (TAICPART), 10-14 Sep., Windosr, (pp: 210), (2007).

[14] Pinto Ferraz Fabbri, S.C. Delamaro, M.E. Maldonado, J.C. Masiero, P.C. *Mutation analysis testing for finite state machines.* Proceedings of the 5th International Symposium on Software Reliability Engineering, (ISSRE), 6-9 Nov., Monterey, CA, USA, (pp: 220-229), (1994).

[15] Shufang Lee Xiaoying Bai Yinong Chen. *Automatic Mutation Testing and Simulation on OWL-S Specified Web Services.* Proceedings of the 41st Annual Simulation Symposium (ANSS), 13-16 April, Ottawa, CA, (pp: 149), (2008).

[16] Delamaro, M. Maldonado, J.C. *Interface Mutation: Assessing Testing Quality at Interprocedural Level.* Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC), 11-13 Nov., Talca, CHI, (pp:78), (1999).

[17] Delamaro, M.E. Maidonado, J.C. Mathur, A.P. *Interface Mutation: An Approach for Integration Testing.* IEEE Transactions on Software Engineering (TOSEM), VOL. 27, Issue 3, March, (pp: 228), (2001).

[18] Serrestou, Y. Beroulle, V. Robach, C. *Functional Verification of RTL Designs driven by Mutation Testing metrics.* Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD), 29-31 Aug., Lubek, (pp: 222), (2007).

[19] Hoijin Yoon Byoungju Choi Jin-Ok Jeon *Mutation-based Inter-class Testing.* Proceedings of the Asia Pacific Software Engineering Conference (APSEC), 2-4 Dec., Taipei, (pp: 174), (1998).

[20] Howden, W.E. *Weak Mutation Testing and Completeness of Test Sets.* Transactions on Software Engineering (TOSEM), Vol. SE-8, Issue 4, July, (pp: 371-379), 1982).

[21] Ferrari, F.C. Maldonado, J.C. Rashid. *A. Mutation Testing for Aspect-Oriented Programs.* Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, 9-11 Apr., Lillehammer, (pp: 52), (2008).

[22] Gupta, V. *Accelerated GWT: Building Enterprise Google Web Toolkit Applications.* Apress, 1st edition, (2008).

[23] Bradbury, J.S. Cordy, J.R. Dingel, J. *Mutation operators for concurrent Java (J2SE 5.0).* Proceedings of the 2nd Workshop on Mutation Analysis (Mutation), 7-10 Nov., Raleigh, NC, USA, (pp: 83-92) (2006).

[24] R. M. Hierons. *Testing from a finite state machine: Extending invertibility to sequences.* The Computer Journal, 40(4):220-230, (1997).

Izzat Alsmadi,

Received March 5, 2011
Revised April 15, 2012

Izzat Alsmadi
Yarmouk University
Computer Information Systems Department
IT Faculty
Irbid, Jordan
Phone: 96227211111
E–mail: *ialsmadi@yu.edu.jo*