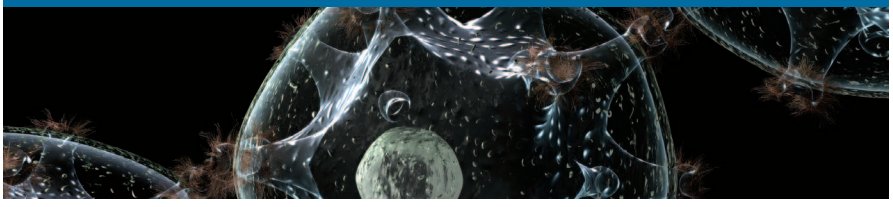
Using Mutation to Enhance GUI Testing Coverage

Izzat Mahmoud Alsmadi, Yarmouk University, Jordan

// A case study of an automated, model-based testing process shows the feasibility of incorporating mutation testing as a standard technique to improve software testing coverage. //



SOFTWARE TESTING CONSUMES a good percentage of overall project expenses, and testing research often focuses on reducing those expenses while maintaining or improving coverage. Test automation techniques support this goal by using artificial intelligence (AI) algorithms to replace or simulate tester activities such as test-case generation, execution, and verification. Mutation testing is an AI approach that has advantages in detecting bugs and exposing software errors that other testing methods might not discover. However, it's often viewed as a general, supplemental test activity that must compete with other activities for limited testing resources.

I've developed a mutation technique specifically for GUI testing. It reverses the traditional mutation testing process of changing a small part of the specification or code and expecting the test cases to discover and reject the mutant elements. Instead, my technique changes a small part of the test cases and expects a model of the GUI to discover and reject the mutants—or in the language of mutation testing, kill them. The technique is based on earlier work developing GUI Auto, a tool for capturing a GUI model.[1,2] GUI Auto addresses both structure and events and can therefore generate test cases directly from the GUI structure file, offering a fully automated approach to mutation testing.

In this article, I extend GUI Auto by using mutation to evaluate the captured model's reliability. Case study results show its potential to make test-case mutation a cost-effective software testing option.

Generating Valid Test Cases

In GUI testing, most components have one main event interaction. For example, a button's main event interaction is the double-click, a textbox's main interaction is text entry, an option item's main interaction is a selection from a menu, and so on. I developed GUI Auto to abstract a model that captures GUI components, their attributes and associations with each other, and one main event for each component.[1,2] Because the abstraction accounts for both structure and events, it can generate test cases directly from the GUI structure file (see the sidebar for related work in how to model user-interface structure and events).

Each automatically generated test case contains a sequence of GUI components, or controls. The program executing those test cases will interact consecutively with the components and try to execute the default action for each one (for a button, a click; for a textbox, typing a text; for an option list, a selection; and so on). If a mutation occurs from changing one component in a GUI level with another one from the same level, the test case might still pass, but it will produce different results.

We can then classify the expected

0740-7459/13/$31.00 © 2013 IEEE

RELATED WORK IN MODELING GUI STRUCTURE AND EVENTS

Paul E. Ammann and his colleagues used a model checker to generate test cases and mutants and evaluated specification and branch coverage.[1] They demonstrated their approach on a popular cruise-control system application. The approach I describe in this article is similar, but it generates the test cases from the dynamic version of the application rather than the design or a static model, which makes it more realistic. It also uses the generated GUI XML graph and automatically generated test cases from that graph to create the mutants, whereas Ammann generated mutants based on the model or model checker.

Aparajita Rao and her colleagues used mutation testing to study the effectiveness of test-case generation.[2] However, they implemented their experiments for preliminary applications and mutant types. Gordon Fraser and Franz Wotawa used the SMV (symbolic model verifier) model checker for automatic test-case generation and mutation testing in evaluating property coverage.[3] A model satisfies a property if all its executable versions do so.

Most of the related mutation approaches use formal methods, tools, or syntax to drive the test cases automatically from either the requirements or the design. One approach is distinguished from the others largely by where the mutation is used and what type of coverage is evaluated. My approach uses mutation on the test cases rather than the actual model, generates the test cases from the GUI graph or structure, and calculates coverage by the ratio of wrong test cases discovered to the total number of wrong test cases.

References

1. P.E. Ammann, P.E. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," *Proc. 2nd Int'l Conf. Formal Eng. Methods*, IEEE CS, 1998, pp. 46–54.
2. A. Rao et al., "A Model for the Development of a Mutation Testing Cum Test Case Generation Tool," *Proc. Int'l Conf. Advances in Recent Technologies in Communication and Computing* (ARTCom 09), IEEE, 2009, pp. 812–814.
3. G. Fraser and F. Wotawa, "Using Model-Checkers for Mutation-Based Test-Case Generation Coverage Analysis and Specification Analysis," *Proc. Int'l Conf. Software Eng. Advances* (ICSEA 06), IEEE CS, 2006, p. 16.

behaviors from test-case mutations in GUI testing into three levels:

- *Level 1*—The majority of mutated test cases are killed because they produce invalid test cases that won't fully and successfully execute.
- *Level 2*—Some mutations pass the validation process and produce a valid test case, but they produce results or behaviors that differ from the original test cases.
- *Level 3*—A few mutations are not killed because they prove valid by producing results identical to the original test case or behaviors or with differences that can't be distinguished or observed.

The complete test-automation process requires an execution and verification tool that can distinguish the results of mutant test cases from those produced by the original test cases.

Validating Test-Case Generation and Execution

If a system is expected to accept valid test cases, then in principle, it should also reject invalid test cases. This is the main hypothesis on which I base my mutation approach. It lets us test the GUI model according to its capability to reject invalid test cases. Mutation is used to make some test cases invalid and to test the system's capability to discover and reject those mutations.

If none of the test cases in a test suite detects the difference in behavior between the original and mutant code, it means either that none of the test cases can reach the mutant or, if it's reachable, that the mutant generates an external behavior similar to the original code's. Thus, a test-case mutant remains live if

- it's equivalent to the original test case, and the application can't tell the difference between the original and the new behaviors—for example, the mutants could be functionally identical but syntactically different (that is, equivalent test cases); or
- the difference in the mutant's behavior isn't propagated to the external interface, so the program is incapable of killing it.

The conditions necessary to kill a mutant are reachability, infection, and propagation. We can therefore measure test coverage according to the fraction of dead specification or code mutants:

$$\text{Coverage} = \frac{\text{Mutant test cases discovered}}{\text{Total mutant test cases}}.$$

This definition of coverage is somewhat new and is a direct indicator for the GUI model quality. In this approach, complete coverage means kill-

ing all nonequivalent test cases. Typical coverage types evaluated in the testing scope include requirements, code, statements, branch, and path coverage, and traditional mutation requires testers to add new test cases to kill the mutants. By comparison, my approach addresses a problem in the GUI model, determining why it couldn't reject an incorrect test-case sequence. Because this is model-based testing, I'm only considering mutation modifications related to the GUI structure. I don't consider mutations related to the specification, such as invalid user inputs based on boundary values and equivalent partitions, because these mutations will affect the code but not the GUI structure.

This reversal of the mutation evaluation process by fixing the code and specification and mutating the test cases isn't a matter of testing the test cases. Rather, we can classify it as a model-based testing approach. The GUI model is expected to discover and kill the mutants, and test adequacy is therefore measured by the number or percentage of the failed test cases. Initially, the approach requires calibration to make sure that all original test cases pass. (Alternatively, we could take the number of the successful test cases as the denominator in the coverage equation.) In the first stage, before mutating the test cases, all test cases in the suite are tested to make sure the GUI model accepts and validates them.

This research extends the GUI Auto test-automation tool by adding mutation testing. The original tool 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, along with other algorithms and techniques that can trigger other automation activities such as test-case generation, execution, and verification. The extensions here are based on GUI mutation operators that can be generated and executed automatically. The new mutation component focuses on GUI component mutations, ignoring the code behind or the actual code in the GUI event triggers.

TABLE 1

GUI features of the four applications under test (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

TABLE 2

Summary descriptions of seven mutation operations.

Mutation operator type	Description
1	Switching GUI components
2	Changing the name in one control of a sequence
3	Changing the name of every control in a sequence
4	Changing one control from a sequence with another control from same level
5	Changing one control in every test case with another one from a different level
6	Deleting a control from the sequence
7	Adding a control to the sequence

Case Study Evaluation Results

I selected several small, open source applications for the case study evaluation, placing two conditions on their selection.

First, because GUI Auto uses reflection to serialize .NET-managed code and extract all GUI components from it, I selected only .NET-managed applications. A managed code is a program—in this case, written in a .NET programming language (such as C#, Visual Basic .NET, managed C++, or JScripts)—that executes within a runtime engine, such as .NET framework or Java Virtual Machine, installed on the same machine. (An unmanaged code can't be serialized to its components using reverse-engineering processes.)

The second condition restricted the selection to applications that contain a reasonable number of GUI components in many forms, so I could construct a GUI hierarchy and generate different test-case sequences on several levels.

Table 1 summarizes the GUIs of the four applications selected for this experiment.

On the basis of the GUI model, structure, and components, I proposed seven mutation operators to evaluate the proposed mutation effects (see Table 2). The GUI model takes all test cases as a sequence, and the automatic execution process applies those test cases to the actual GUI. An automatic verification method checks the

```
1,FRMDATADISPLAY,GROUPBOX3,LSTFIELDS,,SETTINGS
2,FRMDATADISPLAY,MENUSTRIP1,OPENANEWCONNECTIONTOOLSTRIPMENUITEM,,PROGRAM,
```

(a)

```
1,GROUPBOX3,FRMDATADISPLAY,LSTFIELDS,,SETTINGS
2,MENUSTRIP1,PROGRAM,FRMDATADISPLAY,OPENANEWCONNECTIONTOOLSTRIPMENUITEM,,
```

(b)

FIGURE 1. Test-case example mutations for operator type 1: (a) original test cases and (b) test cases after mutation. In test case 1, **FRMDATADISPLAY** switches with **GROUPBOX3**; in test case 2, **MENUSTRIP1** switches with **FRMDATADISPLAY** and **PROGRAM** moves between them.

TABLE 3

Execution effectiveness for type 1 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.82
Notepad	50	0.816	0.802
BirdWatcher	30	0.86	0.84
CourseReg	50	0.71	0.73

** Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.*

test cases that were successfully executed and defines an execution coverage metric as the number of executed controls relative to the number of input or generated controls. The execution process tries to execute the controls in each test case one by one, after which it calculates the effectiveness for each test case. The overall average for the test cases becomes the execution coverage.

Mutation Operator Type 1

The first mutation switches two GUI components in each test case, as shown in Figure 1, where **FRMDATADISPLAY** switches with **GROUPBOX3** in test case 1, **MENUSTRIP1** switches with **FRMDATADISPLAY** in test case 2, and **PROGRAM** moves between them.

Table 3 compares the test-execution effectiveness between the original and mutated test cases. The effectiveness metrics in Table 3 and all subsequent tables reflect the ratio of the number of GUI components successfully executed and verified to the number of GUI controls applied. The number of GUI controls in each test case varies from three to seven in the selected applications. My earlier definition of effectiveness as the number of failed (detected) mutants relative to the total number of inputs is usually the complement of the effectiveness calculated in these tables. In Table 3, we're not focused on the effect of changing the test-case generation algorithm, because it might not be important to the test-case verification processes. Instead, the focus is on the effects of the execution process when test cases are mutated.

The expectation is that the mutants will be less effective than the original test cases and will thereby indicate the application's capability to reject wrong test cases. In normal situations, switching test-case elements should cause a test failure. However, in this experiment, some mutants of this type weren't detected, which might not imply a failure of the GUI model itself to detect such mutations but only the execution and verification process's inability to detect this type of mutation. This is because the execution process segments each test case into its components and then tries to verify the successful execution of each GUI component individually, independently from the other components in the same test case. The automatic execution and verification process executes and searches for every control from each test case in the application assembly (which contains all GUI application components) and verifies that control's existence and ability to be executed. This is why switching test-case elements didn't have a major effect in Table 3, when comparing type 1 effectiveness before and after.

In reality, the automatic verification process is complex and subject to several environmental factors. For example, timing and synchronization between the currently visible forms or components is very hard to accomplish. The automatic execution algorithm might be expecting a button click at a moment when the opened form isn't yet visible or ready. Another problem is modeless modules, forms, or dialogs that don't accept any further commands before closing. It's also a challenge for the automatic verification to define some GUI components' visibility, especially containers, and executing them can be difficult.

The trials to automatically verify the execution of a complete GUI path were unsuccessful for type 1. In the future, I plan to implement a modified execution algorithm to verify the test case in the same sequence and so cover this weakness in the verification process.

Mutation Operator Type 2

The type 2 mutation operator changes the name of one control in the sequence—for example, by adding or removing one letter. The goal is to keep the control entity but change its identity. If each GUI control is defined by its name only, the GUI model should detect this mutation type.

The algorithm randomly selects the location of the mutated letter and the control (from the test case). Table 4 shows the effectiveness results from this mutation gathered from the actual applications. A decrease in execution effectiveness for type 2 indicates a lower percentage for only the controls that were located before mutation. We can theoretically calculate the controls that were affected by the mutation process relative to all controls as follows:

$$\mathsf{NewEff} = \mathsf{OrgEff} - (\mathsf{NoMut} / \mathsf{NoControlsTotal}),$$

where NewEff is the effectiveness after mutation, OrgEff is the effectiveness before mutation, NoMut is the total number of modified controls (through their names), and NoControlsTotal is the total number of controls in all test cases applied.

For example, for DBSpy, OrgEff = 0.75, NoMut = 30, and because NewEff = 0.53, the total number of controls in all test cases will be (30/0.22) or about 136 control (average controls in each test case = 136/30 or about 4.5). However, this theoretical value assumes that all mutated controls are undetected and all unmutated controls are detected in the same way as before the mutation process is applied.

TABLE 4

Execution effectiveness for type 2 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.53
Notepad	50	0.816	0.75
BirdWatcher	30	0.86	0.64
CourseReg	50	0.71	0.468

** Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.*

TABLE 5

Execution effectiveness for type 3 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0
Notepad	50	0.816	0.03
BirdWatcher	30	0.86	0
CourseReg	50	0.71	0

** Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.*

Mutation Operator Type 3

In an extension of type 2, the type 3 mutation changes the name of every control in the sequence by adding or removing one letter, for example. To distinguish a node (control) failure from a path failure, this mutation modifies every control name in the test path. Table 5 shows the results of applying this mutation.

Changing all the control names should bring all test cases to a complete failure. The few exceptions occur in the rare case when removing a letter from a control changes its name to another valid one.

Mutation Operator Type 4

This mutation replaces one control from the sequence with another control from the same level, bypassing some GUI structure constraints in which a test case should contain GUI components from the different levels. In some cases, changing this control can change the test case, but will also produce another valid test case.

For example, in

MAINMENU,EDIT,UNDO--TO--FILE,EDIT,COPY,

UNDO and COPY are two controls from the same level. Table 6 shows the results in effectiveness of applying mutation type 4.

As expected and explained earlier, switching GUI controls doesn't affect test-case effectiveness because it modifies the test case without invalidating it. In our mutation testing, we're not testing whether the value before and after mutation stays the same. The tests on the GUI model focus only on verifying whether the model will accept or reject the new mutated test cases. As a result, even though this type of mutation changes both the test case and the path it's testing, the new test case is valid. In some cases, as in the CourseReg application, the mutation improves the effectiveness.

Mutation Operator Type 5

In this mutation, one control in each test case is replaced with a control

TABLE 6

Execution effectiveness for type 4 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.73
Notepad	50	0.816	0.815
BirdWatcher	30	0.86	0.81
CourseReg	50	0.71	0.82

* Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.

TABLE 7

Execution effectiveness for type 5 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.64
Notepad	50	0.816	0.812
BirdWatcher	30	0.86	0.80
CourseReg	50	0.71	0.82

* Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.

TABLE 8

Execution effectiveness for type 6 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.71
Notepad	50	0.816	0.80
BirdWatcher	30	0.86	0.80
CourseReg	50	0.71	0.62

* Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.

TABLE 9

Execution effectiveness for type 7 mutations.*

AUT	No. of test cases	Effectiveness before	Effectiveness after
DBSpy	30	0.75	0.76
Notepad	50	0.816	0.77
BirdWatcher	30	0.86	0.77
CourseReg	50	0.71	0.75

* Effectiveness = the number of components successfully executed and verified / the number of GUI controls applied.

randomly selected from the pool that contains all Applications Under Test (AUT) controls without observing the location of the newly selected control. Table 7 shows the results from applying this mutation.

Similar to type 1, this mutation should noticeably decrease the effectiveness of mutated test cases. However, the automatic execution and verification process that verifies the existence of each executed control in the managed code (without considering whether its test case is still valid or not) is limited. So, rather than lowering test effectiveness, using mutants improves effectiveness. Switching the locations of some controls therefore made them more visible, and the execution algorithm located them successfully.

Mutation Operator Type 6

Deleting a control from a sequence. This mutation deletes one randomly selected control from each test case. Table 8 shows the results.

This mutation should affect execution effectiveness by deleting controls from the test cases solely because of the deletion since the new calculated effectiveness will be based on the new test cases taken the deletion into consideration. However, all AUTs showed reduction, which indicates that deleting some controls may affect the visibility of some other controls.

Mutation Operator Type 7

Rather than switching an existing control with another one, the last mutation randomly selects a control to add to each test case. Table 9 shows the results. It makes no difference whether we add the same randomly selected control to all test cases or add a new randomly selected control every time.

As with removing a control, merely adding a control doesn't impact effectiveness because the addition is reconsidered when calculating effectiveness.

Results showed that in all mutation cases, test effectiveness after mutation should be less than test effectiveness before; however, because the verification process verifies the controls one by one, the addition of some GUI controls increases effectiveness, although this might not reflect the test-case generation's actual effectiveness.

The automated process I've described for validating test cases has some weaknesses—in particular, it can't verify test cases as complete units but instead verifies each control individually regardless of its test case. Nonetheless, the results showed promise for using test-case mutation to verify certain GUI model properties.

In typical mutation testing, if results from a mutated test case are different from the original case, the mutant is killed. In my approach, the validation of results considers killing mutants by rejecting them. This complicates the automatic verification process by the need to define GUI correct and incorrect states. I plan future extensions to this approach by including the built-in tests that should be designed and implemented for all software applications. Such tests can be useful in defining general proper usage and behavior of software user interfaces.

References

1. A. Izzat and K. Magel, "GUI Path Oriented Test Generation Algorithms," *Proc. Int'l Assoc. Science and Technology for Development Conf. Human-Computer Interaction* (HCI 07), ACTA Press, 2007, pp. 216–219.
2. A. Izzat and K. Magel, "An Object Oriented Framework for User Interface Test Automation," *Proc. Midwest Instruction and Computing Symp.* (MICS 07), 2007; http://static.aws.pdf-archive.com/2011/08/13/10-1-1-101-3642/10-1-1-101-3642.pdf.

ABOUT THE AUTHOR

IZZAT MAHMOUD ALSMADI is an associate professor in Yarmouk University's Department of Computer Information Systems in Jordan. His research interests focus on software testing and metrics. Alsmadi received his PhD in software engineering from North Dakota State University. Contact him at ialsmadi@yu.edu.jo.