A Visual Test Development Environment for GUI Systems

Thomas Ostrand  Aaron Anodide  Herbert Foster  Tarak Goradia
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540  U.S.A.
tjo@scr.siemens.com

ABSTRACT
We have implemented an experimental test development environment (TDE) intended to raise the effectiveness of tests produced for GUI systems, and raise the productivity of the GUI system tester.

The environment links a test designer, a test design library, and a test generation engine with a standard commercial capture/replay tool. These components provide a human tester the capabilities to capture sequences of interactions with the system under test (SUT), to visually manipulate and modify the sequences, and to create test designs that represent multiple individual test sequences. Test development is done using a high-level model of the SUT's GUI, and graphical representations of test designs. TDE performs certain test maintenance tasks automatically, permitting previously written test scripts to run on a revised version of the SUT.

KEYWORDS:
testing, GUI-based system, capture/replay, test scenario, test designer, test generation, visual editor, test maintenance, test coverage

1. MOTIVATION
More and more applications with complex graphical user interfaces are being built. Such applications can have dozens of screens, each with many input fields and controls, and hundreds of legal combinations of user inputs. Thorough testing is essential, yet the sheer magnitude of the job can be overwhelming.

Imagine you are part of a team charged with developing a comprehensive and effective test suite for such an application with a graphical user interface. The requirements on your tests are:

1. they must exercise all areas of the system's functionality.
2. they must exercise all aspects of the system's GUI: all the input forms, individual user choices, and sequences of user actions that are likely to occur in practice.
3. the tests should be maintainable and remain usable as the system changes; minimal effort should be needed to allow previously recorded test scripts to run on the new version.

In addition to these effectiveness, coverage, and maintenance requirements, you are expected to produce the tests quickly, provide thorough documentation of the tests and evidence of their comprehensiveness, and demonstrate their thoroughness when executed.

To test the application, you can use a commercial capture/replay tool, such as QA Partner® [2], JavaStar® [5], WinRunner® [3], or SQA RobotTM [4], which permits the tester to record, and later play back, sequences of actions on the interface. Correctness of a GUI-based system is defined both by proper behavior of the GUI and proper computations of the underlying application. Test cases mirror the actual use of the system, which consists of many interactions between the user and the system, rather than uninterrupted computations that are based on a set of inputs and produce a set of outputs. Capture/replay tools make it easy for the tester to create executable test scripts that are made up of these multiple interactions. They remove the need to explicitly describe the desired sequences of actions to be performed, and make it simple to run large sequences of recorded tests without human supervision.
The specific abilities of CR tools that are relevant to our work are the following:

- they record scripts of user/system interactions
- they give the user access to the recorded scripts for editing and maintenance.
- they allow the user to insert state and output validation commands in a script.
- they permit the replay of a recorded or edited script on the original system, usually on any hardware platform that hosts the system.

A CR tool collects and makes transparent to the user a substantial amount of information about the structure of the objects that make up the application’s GUI. They capture object information and record test scripts in logical form, so the tester need not know about underlying implementation details.

A capture/replay tool is generally packaged as part of a testing product that provides a variety of capabilities, some of which might be test planning and management, requirements analysis, test case generation, test script generation and execution, and test result storage and reporting.

Figure 1 shows the structure of a typical commercial capture/replay tool. The tester exercises the system under test (SUT), and the capture/replay tool records a script of the user actions and system responses. The script is recorded in a test script language, usually proprietary for each individual tool. A recorded script can be used as a test script and replayed through the SUT, or it can be edited by the tester to create new and additional scripts. Although a recorded script consists of a straight-line sequence of user actions and system responses, the script language typically contains many constructs found in programming languages, and can be used to create “test programs”. Through direct editing of a recorded script, the tester can create script variations, insert code that checks for special conditions, and create scripts that allow different action sequences to be exercised depending on the user’s input or the system’s responses. However, successful script editing requires thorough knowledge of the script language’s details.

2. TDE OVERVIEW

With a capture/replay tool, the basic testing requirements 1-3 can be only partially met. Although it is easy to record individual user action sequences, they are difficult to modify. The tools generally provide no high-level view of the functionality that is tested by a recorded sequence of actions; it is up to the tester to write test script documentation. Furthermore, since only a single script can be recorded at one time, creating many test scripts is a tedious and expensive process. Either a script corresponding to each desired test script must be recorded individually or else a set of master scripts must be recorded and then edited by hand to create variations. Test maintenance is also difficult, since script updating can only be done at the level of the tool’s scripting language, and is tedious, time-consuming, and error-prone.

The test development environment TDE aims to raise
the productivity of the GUI tester and raise the effectiveness of tests produced for GUI systems, by improving these deficiencies of standard capture/replay tools. TDE replaces the low-level scripting language of the capture/replay tool with a higher level scenario language, and allows the tester not only to record, but also to modify and create variations of recorded sequences using a visual representation of the GUI.

The components of TDE are a test designer, a test generation engine, and a test design library, as shown in Fig. 2. They are described briefly below. These three key components are combined together with a standard commercial capture/replay tool, and made accessible to the tester through a visual display/definition/editing interface.

The Test Designer collects descriptive information about the GUI from the capture/replay tool, provides a visual interface to the system model and to captured scripts, and allows the user to modify and generalize scripts directly in terms of the operations of the SUT. If necessary, the tester always has access to the underlying scripting language of the capture/replay tool, and can edit test scripts at that level.

Using the Test Designer, the user produces a high-level test design that can represent one or many individual executable scripts. The Test Designer is also the tester's interface to the Test Design Library.

The Test Design Library contains recorded scripts, test scripts, and general test patterns that can be used to build specific scripts.

The Test Generator Engine converts the high-level test design into scripts that can be executed on the SUT through the normal playback mechanism of the capture/replay tool.

Table 1 shows the sequence of phases that occur when TDE is used to develop, execute, and maintain test scripts for a GUI system. The second and third columns specify the actions carried out by the tester and by TDE during each phase. The essential first phase is the construction of a high-level model of the GUI of the SUT. The model is the basis for the tester's navigation of the system while tests are designed. The tester uses the model to add specific steps to a test specification. It also serves as a basis for evaluating the thoroughness of the tests that have been defined.

The system model has two components. In the top-level graph, each node represents a window of the SUT's GUI, and an arc from Window A to Window B represents a user action in Window A that invokes...
<table>
<thead>
<tr>
<th>TDE Use Phases</th>
<th>Performed by Tester</th>
<th>Carried out by TDE</th>
</tr>
</thead>
</table>
| **INITIAL INFORMATION CAPTURE** | • Identify GUI's top-level windows  
• Carry out sample sequences of GUI use | • Capture GUI information  
• Build high-level system model  
• Record user/system sequences |
| **TEST SCENARIO CREATION, EDITING, GENERALIZATION** | • interact with GUI  
• define data variations  
• define path variations  
• specify constraints  
• use test library | • Convert recorded sequences to TDE scenarios  
• Evaluate test coverage on high-level model |
| **TEST GENERATION** | | • Convert scenarios to test generator input  
• Generate test scripts in capture/replay tool scripting language |
| **TEST EXECUTION** | • Choose desired test scripts | • Prepare scripts for playback  
• Replay scripts and record results |
| **TEST RESULTS EVALUATION** | • Debug/redesign software | • Measure and report coverage of executed tests |
| **TEST SCRIPT & SCENARIO MAINTENANCE** | • Identify changed GUI elements  
• Confirm/reject TDE proposed updates | • Determine detailed changes to GUI  
• Make simple updates to test scenarios  
• Propose complex updates to scenarios |

Table 1: Phases in TDE Use

Window B. One level down are the component representations, individual models of the windows of the SUT's GUI. These models are faithful representations of the actual windows of the GUI. The tester can interact with them to create and edit test scripts and scenarios. The system model represents only the behavior of the system's GUI, not the functionality of the system application.

The first two TDE actions shown in Table 1 are carried out when TDE encounters a new GUI. The constructed system model remains in use until the GUI is redesigned, and TDE must construct a new model. In the section AUTOMATED SUPPORT FOR TEST MAINTENANCE, we describe how TDE aids in the process of updating previously developed test scripts when the SUT and its model undergo change.
3. THE VISUAL DISPLAY AND EDITING ENVIRONMENT OF TDE

A captured script is a sequence of user actions, recorded in the order they are performed. It can be replayed as a test script on the SUT, and corresponds to a single execution path of the SUT. A captured script is converted by TDE into a test scenario, which can be displayed and edited graphically. A major goal for TDE is to provide an easy-to-understand language for building test designs. The display and editing environment consists of a visual display of the SUT model and test scenarios, a test scenario editor, and a test design library. The visual representation of test scenarios uses graphical symbols to indicate the scenarios' control elements and visual positioning to indicate the scope of each element.

Representing the System under Test (SUT)

Figure 3 shows the TDE control panel (A) and the two main components of the SUT model, the Interactions Window (B) and the Component Window (C).

The application shown in the figure is a simple interactive text editor, whose operations include text insertion, search, find/replace, etc. The application's main window is the Text Editor, where the current editor buffer is located, and where the user enters and modifies text. This window is currently displayed in the Component Window of TDE. From the Text Editor's main window, the five other windows Find, GotoLine, Replace, FileOpen, and FileSaveAs are accessible by menu selections. An SUT window's image is shown in the TDE Component Window when the tester clicks on that window's node in the TDE Interactions Window.

The Interactions Window gives an overview of the system under test, and allows quick navigation to any specific window the tester wants to examine. Since the individual transition actions are relatively long strings that occupy a large amount of space, they are shown only on demand, when the tester places the cursor on the small oval that is embedded in the transition arc. Clicking the oval on the arc from a source window to a destination window causes the source window to be displayed in the TDE Component Window, and the action of the transition (e.g., a menu selection or a button push) to be highlighted in the source window. In the Figure, the tester has put the cursor on the arc from TextEditor to Replace. This causes TDE to display the arc's transition action at the bottom of the Interactions Window:

Click on MenuItem (Text Editor.Search.Replace).
Figure 4. Scenario generated by TDE from recorded sequence of user actions

The tester has also clicked on the arc, causing TDE to highlight the Replace menu item in the TextEditor window (C).

In order to build the visual representation shown in Figure 3, TDE needs information that describes the structure and behavior of the GUI's objects. This information is collected by the capture/replay tool, and passed to TDE during the initial capture phase (the first phase shown in Table 1). The "GUI Info" box in Figure 2 represents the explicit transfer of this information from the tool to TDE. The initial capture phase requires a tour of the interface, i.e., successive invocation of each window. In the TDE prototype the tour is done manually; in production versions the process will be automated.

Representing Scripts and Scenarios

Figure 4 shows the text editor's Replace window, and a scenario that was produced by TDE from one captured script that was recorded by the capture/replay tool. Each diamond-shaped box in the scenario represents a single user action. The scenario produced from the captured script can become the basis for additional scenarios developed by the tester. Each user action that appears in a test scenario is visually linked to the component window of the GUI where the action was performed. If the tester clicks on a step in the scenario, TDE highlights the object in the component window where the action was performed (button, text entry field, menu selection, etc.). In the figure, the tester clicked on the outlined Replace.ReplaceWith.SetText user action, causing the cross-hatched text field in the component window to be highlighted.

Editing Test Scenarios

Existing scenarios can be augmented or modified, and new scenarios can be created using visual editing tools in TDE. To modify a scenario, the tester first clicks on an element of the scenario, making it the current editing point (the outlined step in Figure 4). Additional
actions can be inserted into a scenario simply by carrying out the actions on the relevant TDE component window, as if the tester were interacting with the application itself. Steps can be copied, cut, and pasted, within a single scenario or across different scenarios.

4. VARIATIONS IN TEST SCENARIOS

A test scenario can be generalized by defining *data variations* and *path variations*. A data variation specifies a set of possible values, any of which can be used in place of a fixed value that appears in a scenario, such as a user input to a numeric or text entry field. A path variation specifies variations in the execution order and choice of user actions that are executed in the scenario. A generalized test scenario represents a set of test scripts; the TDE test generator engine transforms it into executable test scripts that are written in the capture/replay tool's script language.

**Path variations** are defined with the use of *sequence*, *choose*, and *repeat* constructs. Each of these is a meta-operation that applies to a group of user actions of the scenario; the meta-operations instruct the test generation engine which specific test scripts to generate from a test scenario. Individual actions or groups of actions can be inserted inside the scopes of the constructs, either by edit operations, or by direct performance of the actions in a component window.

The *sequence* construct acts as a grouping mechanism for the actions in its scope. The actions are executed without interruption in the order they appear.

The *choose* construct indicates that the actions or groups of actions in its scope are alternatives for execution. A single choose construct with k actions or groups in its scope corresponds to k different test scripts.

The *repeat* construct takes an integer argument n, and indicates that the actions or groups in its scope are to be repeated n times.

Figure 5 shows a generalized scenario, with sequence, repeat, and choose constructs represented respectively by a square, a circle, and an inverted triangle. The scope of each construct is attached to the branch at the right side of the symbol. The branch at the left indicates how the flow of the scenario continues after the construct. The two Sequence constructs in the figure have no left branch because nothing follows them within the scope of the Choose.

The separate choices of a Choose are attached to its right-hand (scope) branch. Their order is irrelevant, because execution of a Choose means that any one of the separate choices can be taken. Because the Choose in the figure has two choices, the scenario represents two different test scripts. These two scripts are identical up to the TextEditor.Search.Replace.Pick step; then the first contains the two Replace actions of the first Choose branch, while the second contains the two different Replace actions in the second Sequence construct. Following the Choose is a Repeat construct, with a variable *RepeatCount* whose value specifies the number of times the scope of the Repeat is to be iterated.

**Data variations** are defined by replacing a fixed value
with a variable that can take on the values specified by a data definition scheme. A data definition scheme for a variable is a context-free grammar whose start symbol is the variable, and whose terminal symbols are used to form legal values that are appropriate for the user input. The right-hand side of a production in the grammar can contain actual values, other variables, or a concatenation of values and variables. Each variable that appears in a scenario or a data definition scheme is defined in a Data Variation window.

Figure 6 shows part of the data definition scheme for the variable FileName, which specifies different file names to be opened in the generalized test scenario. The top Data Variation window shows the data definition scheme for FileName; the variable can be replaced by any of three possible choices. The first of these, FileChoice1, is defined as the concatenation of three additional variables: Prefix, Core, and Index. Each of these variables is further defined in its own data definition schema. The second Data Variation window shows the three choices available for Core. The overlay arrows on Figure 6 show where 1) the variable name is brought from the Component window and inserted into the scenario; 2) the choices for the variable are defined; 3) the variable Core is defined in its data variation window to have the three possible choices Alpha, Num, and AlphaNum.

Data definition schemes are also used to assign a range or set of values to numeric variables such as the RepeatCount variable. Each individual value $n$ causes the test generator to produce an individual test script with $n$ repetitions of the repeat scope.

The use of data and path variations enlarges the set of test scripts that a scenario represents. It is also frequently necessary to reduce the number of scripts, to keep the number of tests down to a level that can be executed within the available time and equipment resources. The tester can establish constraints for a scenario that limit the unrestricted combining of all possible data and path variations. Several examples of constraints are given in the next section.

The final tool that can be used during test scenario editing is the test design library. The library is a stored collection of test scenarios that can be inserted, either directly or by reference, into the scenario currently being edited. For example, the library might contain a test paradigm with the user actions: Open a window, Click an entry field in the window, Type an input, Click a search button; followed by the test steps: Wait for the result, Compare the actual result to an expected result, Log the test result. Such a paradigm can be imported by the tester and made specific to a particular window and test situation.

5. TEST GENERATION

In this section we describe how the Test Generator Engine creates a set of test scripts from a scenario. A key goal of TDE is to have the test designer's view of test scenarios, as far as possible, independent of the
details of both the specific test execution environment of the SUT, and of the test scripting language of any capture/replay tool that is used to run the tests.

To achieve this independence, TDE separates the test designer module from the test generator. Test scenarios are stored and displayed in a high-level format that represents their path structure and data definition schemes. Test generation is a two-step process. First, a test scenario is converted into the specification form that is recognized by the TDE test generator engine. For the current prototype of TDE, the test generator is the TSL system [1]. Second, the test generator processes its input to produce test scripts in the scripting language of the capture/replay tool.

A test scenario is a generalization of a recorded script that contains three element types:

- **user actions:** the individual operations the system user performs when interacting with the GUI
- **control elements:** instructions that specify how to combine and order user actions into test cases.
- **data elements:** variables that can be replaced by specific input values supplied by the system user to an input field of the GUI.

The control elements are slightly modified versions of the three operations that are used to form finite regular expressions: union, concatenation, and finite closure. These elements are used in a test scenario to describe sets of test scripts. When a scenario is processed by the Test Generator, it is treated as a finite regular expression over a base alphabet consisting of the user actions. The formal definition is the following:

A single user action a is a test scenario, representing the single test script a.

If S and T are test scenarios representing the sets of test scripts S and T, then

- **sequence (S,T):** is a test scenario, representing all test scripts of the form st, where s is in S, and t is in T.
- **choose (S,T):** is a test scenario, representing all test scripts that are either in S or in T.
- **repeat (n,T):** is a test scenario, representing all test scripts of the form t₁t₂...tₙ, where each tᵢ is in T.

Expansion of a scenario into test scripts is governed by a set of rules involving the relations among path variation directives, data definition schemes for variables, and the scenario's constraints. The rules are peripheral to the overall concept and structure of TDE, so in this paper we do not describe them in detail. In the following, we first give simple descriptions of the effects of path and data variations without reference to constraints. We then give several examples of the uses and effects of constraints.

**Path variation directives**

A choose directive with n branches causes the creation of n test scripts, each with one of the choices. Distinct choose directives expand independently, so that a scenario with two choose directives, with m and n branches, would be expanded into m * n scripts.

A repeat n directive expands into a single test script, with the scope of the repeat occurring n times in succession.

**Variables**

If the data definition scheme for a variable produces n different values of the variable, then the TDE test generator produces n different test scripts, each with one of the values. If several variables appear in the scenario, they are expanded independently; hence the test generator produces scripts with all combinations of the variables' values.

**Constraints**

The unconstrained use of path and data variations as described above can easily yield far more test scripts from a test scenario than are useful or practical. To prevent an explosion of test scripts, TDE's constraints restrict the generation of the scripts. To show the flavor of constraints, we give a few examples:

- if A and B are variables in a scenario, a constraint can say that a particular value of A may not occur in a test script together with a particular value of B.
- a particular value of A may be constrained to occur in no more than k generated test scripts.
- a particular branch of a choose directive can be constrained not to occur in a test script together with a particular branch of another choose directive.

**6. TEST COVERAGE**

In current software engineering practice, test thoroughness is usually evaluated in terms of the code (statements, branches, dataflow associations) covered during test case execution. Since TDE builds a
graphical model of the SUT, and the tester refers to this model when designing tests, it seems natural and useful to consider a model-based method of evaluating test thoroughness. One advantage of a high-level model-based measure is that it can be evaluated before test execution, using the high level scenarios.

The TDE prototype has experimented with several such measures, including a notion of event coverage, which measures the percentage of possible user actions carried out on all the widgets of a window by a given test set. We intend to study the usefulness of this and other measures, and compare them to traditional code-based test thoroughness measures.

7. AUTOMATED SUPPORT FOR TEST MAINTENANCE

Keeping large quantities of test cases up-to-date with respect to changes in the system being tested is a major issue for test developers. In some cases, when changes to the system are cosmetic, the tester may wish to replace existing scripts with new scripts that carry out exactly the same tests as previously, except that necessary renamings have been made. If existing functionality is changed or removed, or new functions added, an existing script might no longer be meaningful, or the set of existing scripts might no longer thoroughly test the system.

When test scripts are saved by a Capture/Replay tool, the tester must maintain a large volume of tests in the scripting language. Although some changes may require only modification of a few entries in a table, others may require detailed updates to all the relevant scripts. TDE stores the test information as scenarios, and generates actual executable test scripts only after a scenario has been edited to its final form. This allows maintenance to be done at the higher level, and in terms of the GUI itself.

The goal of test updating is to automate as much of the update process as feasible, with minimal intervention by the tester. At the same time, the system should not make unjustified assumptions about correspondences between old and new versions of the SUT. Automated update consists of three phases:

- identifying changes in the interface
- proposing and confirming the mapping from original version to new version
- carrying out the necessary changes to test scenarios.

TDE can analyze and detect differences between the original and the modified GUI of the SUT. Some examples of detectable differences are: a menu with more or fewer items; a menu selection item is moved from Menu 1 to Menu 2; a window previously invoked from Window A is now invoked from Window B; Window A has been removed, and the items previously in it are now distributed in to new windows; Window A has been redesigned so that the items previously visible directly are now accessed through a hierarchy such as a Tab system.

Following detection of differences, TDE makes certain simple assumptions about correspondence of functionality from the old to the new version. For example, a menu item that formerly appeared in Menu 1 and now appears in Menu 2 in the same window, is assumed to be the same item, with the same functionality. In the example of the new Tab system, TDE can assume that the items accessible under tabs are the same ones that were formerly directly visible. The tester can always override the TDE assumptions, using the visual representations of the old and new versions to define the mapping. The tester first performs the old operations (button pushes, menu selection, etc.) on the old version of the GUI, and then performs the corresponding replacement operations on the new version.

Once the mapping has been established, scenario steps involving the old constructs are replaced with their corresponding steps for the new constructs. The changes can be made either autonomously by TDE, or with approval of the tester. The tester can use a visual search and replace editing command, modeled after the query-replace command of a text editor like Emacs, to iterate through the scenarios and decide each change individually.

8. IMPLEMENTATION

The TDE prototype has been constructed on a Sun workstation, using Tcl/Tk as the implementation language. We examined several different commercial capture/replay tools, including Mercury Interactive's XRunner/WinRunner®, Rational Software's SQA Robot™, Segue Software's QA Partner®, and Sun's JavaStar™. The current prototype uses QA Partner, but one of the TDE project's goals is to make TDE as nearly as possible independent of any particular capture/replay tool.
9. SUMMARY

TDE is an interactive, visual environment for defining, editing, maintaining and executing test scripts for systems with graphical user interfaces. The system uses a standard commercial capture/replay tool to gather information that it uses to construct a graphical model of the SUT and its windows. The tester uses the model to navigate the SUT, edit recorded test scripts, and create and modify test scenarios.

In general, a test scenario represents a set of sequences of operations that a user would carry out on the system's GUI. The basic constructs of a scenario are single actions, action sequences, repeats, and chooses.

TDE contains a visual editing system for scenarios, with the following features:

- scenarios are displayed in an entirely visual form to the tester, who can cut, paste, and copy in them, as well as insert new instances of the basic scenario constructs.
- TDE permits definition of a data definition scheme for specifying the possible data values for a GUI operation. An individual data variation object may be referenced by several GUI operations, in several scenarios.
- scenarios can also contain constraints, which are restrictions on how different parts of the scenario can be combined to form test cases. Constraints are used to limit the number of test cases that are generated from a scenario.

The scenario editor is visually integrated with the system under test. User actions on the system can be inserted into any scenario merely by performing the actions on a TDE representation of the SUT.

The format of a test scenario is independent of both the specific capture/replay tool used and the specific test generator engine, making scenarios portable across different test development toolsets and execution environments.

Test updating is easier than formerly, since test information is stored as high-level scenarios, rather than low-level scripts. Maintenance changes can be made directly in terms of the system's GUI. TDE analyzes the differences between old and new SUT versions, and provides some automated updating.

Test coverage of the SUT is analyzed in terms of the system model that TDE constructs.

10. CURRENT STATUS AND FUTURE WORK

Preliminary experiments have convinced us that the TDE interface is very simple to use, easy to learn quickly, and very productive for producing test designs that expand to large numbers of test cases covering all the features of a typical GUI system under test. For example, applying the prototype to generate test scripts for the user interface to a medical diagnosis machine, the tester needed less than 30 minutes to create a single test scenario that produced more than 2500 test cases. Collectively, the cases exercised every significant combination of input values and action choices that are available to a user of the diagnosis machine.

Work on TDE is continuing. We plan to add a facility to the visual editing system for defining result comparison steps in test scenarios. This will give the tester the ability to augment test scripts with oracles to check the state of the GUI as well as the system state and computation results. We will continue to experiment with various coverage concepts based on the graphical model of the tested system.

The real test of the TDE concept will come through its use in actual practice. Several Siemens business groups will soon be using prototype versions of TDE, allowing us to assess its effect on tester productivity and effectiveness.

REFERENCES