HOW TO TEST GRAPHICAL USER INTERFACES

Developing a tester that simulates the user and can be used to automatically test GUIs in the MATLAB environment.

Tamás Dabóczi,
István Kollár,
Gyula Simon, and
Tamás Megyeri

Most of today's application software provides some kind of graphical user interface (GUI). A GUI should be easy to use by following logical and intuitive steps to reach the desired result. It makes it easy to visualize the progress of data processing and to give the necessary background information. The actions (if appropriately constructed) can be accessed by simple mouse clicks on graphical objects. GUIs may also provide textual information where necessary and give the possibility of entering arbitrary alphanumerical values into edit boxes. Graphical visualization is not a constraint but rather an added benefit [1], [2].

Motivation: Improving the Reliability of Software with GUIs

Application software with a GUI is usually a much more complex program than conventional alphanumerical control. The possibility of undetected mistakes in the program increases with the many parallel program branches. Error messages should provide pathways to correct the error. Reliability of the program is essential. There is an increasing demand for testing of a GUI and for testing the whole system governed by a GUI to ensure reliability.

1094-6969/03/$17.00©2003IEEE

Currently, embedded systems such as mobile phones and automatic teller machines or measurement instruments such as a digital oscilloscopes and car diagnostic analyzers have GUIs that play a crucial role in the system. The performance of such a system depends strongly on the software running on the dedicated hardware. The quality of service requirements cannot be met without ensuring a certain level of software reliability. In the world of virtual instruments, the role of testing the software is even more important, because the application software is embedded in a large operating system [3]-[5]. The environment in which the application software runs is often not well documented (think of a PC assembled from components available from various vendors) or is released in certain cases with bugs and awkward features. Moreover, exhaustive testing is not a practical solution, since it is very difficult, if not impossible, to produce and test all the possible configurations and situations.

It is also necessary to ensure that error messages are understandable to the GUI user. The GUI must perform graceful recovery from errors. The user should not be frightened away by technical messages or by catastrophic errors (e.g., freezing); instead, a short hint should appear, with the information about what to do.

A graphical user interface should be easy to use by following logical and intuitive steps to reach the desired result.

Syntactic Versus Semantic Errors

There is a difference between syntactic and semantic errors. The *IEEE Standard Glossary of Software Engineering Terminology* [6] defines these terms as follows:

- *Syntax:* The structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs.
- *Semantics:* The relationships of symbols or groups of symbols to their meanings in a given language.

A syntactic error is a violation of the structural or grammatical rules defined for a language, e.g., "*for i=1 to 10*" in C language instead of "*for (i=1; i<10; i++),*" or a *begin* without the corresponding *end* in Pascal. A syntactic error shows up usually at compilation time. The syntax can also be checked without running the program.

A semantic error results from misunderstanding the relationship between symbols or groups of symbols and their meaning in a given language, e.g., calculation of sin*(x)* instead of cos*(x)*. There are certain expressions that may produce an error depending on the current conditions; e.g., dividing a number with a variable (usually) produces an error if the variable is zero. Similarly, calculating the square root of a negative number produces an error message if complex numbers are not interpreted. Some other examples from this group include indexing an array out of range and allocating a large block of memory beyond the available limit.

It is important to mention that not all semantic errors generate error messages. Thus, it is possible to run a program without getting error messages and yet obtain a wrong result. Automatic testing, to our knowledge, aims to catch errors that produce an error message. Catching semantic errors without an error message is based mostly on human interaction. For example, a specialist can see if the computed results are incorrect by looking at some informative plots like the pole/zero pattern or the transfer function, but the computer cannot recognize such errors.

"Natural intelligence" sometimes outperforms the computer. Therefore, having a good engineer before the screen, which provides easy-to-capture information for the user, can be more effective for GUI testing than any kind of automatic testing. A method that tirelessly allows "nonsubjective" trials in the software, combined with a human observer, is very effective. Testing for all types of semantic errors is expensive and time consuming for people. Exhaustive test, however, cannot be implemented in practice as there are too many possible combinations.

Testing Approaches

Random Human Testing

Some software companies tests GUIs by letting some users play with the software for a couple of weeks, with the goal of catching errors. Although this approach might be useful, it is rather arbitrary, time consuming, and expensive and does not provide reliable coverage of the frequently occurring cases. People are different; some try features that are never touched by others. The repeatability of the testing is also hard to ensure.

Predefined Test Sequences

Another usual approach is to write a program that contains calls to the GUI to be tested. Theoretically this might also work; in many cases, however, such an approach is as arbitrary as the random human testing; moreover, writing such a program is cumbersome, and it is difficult to achieve good coverage of the cases. An advantage is that repeatability is straightforward, if the test is always started from the same initial state.

Formal Description of the System

Considering the possibilities of testing, the systematic one is based on a formal description of the software system, which allows automatic generation of a test program. This works well in theory, but in practice, it is rare to have a good formal description of the system. Therefore, this is a theoretical option rather than a practical one.

Simulation of the User

Why not let the computer act as an ordinary user would, providing mouse clicks, entering numeric or alphanumeric inputs, activating user controls, and so on? By simulating the user, we test the software for errors that produce error messages.

Exhaustive testing of the application software by a person is usually not possible because the number of possible inputs from the user is virtually unlimited. Random testing can find many bugs. Completely random testing is not always a good approach because it does not describe reality well (i.e., the random inputs may be far from the actual usage of the system). Thus, we have found a heuristic approach with guided randomness appropriate.

Let the computer act as an ordinary user would. By simulating the user, we test the software for errors that are producing error messages.

The Proposed Approach for Random Testing

Our goal was to develop a tester that simulates the user and can be used to automatically test GUIs in the MATLAB environment [7]. The tester is a software program that provides graphical interaction with the GUI in a heuristic way. We intended to find software failures that would cause an abrupt termination of the application software. This abrupt termination could be due to an unexpected input from the user, for which the application software is not prepared. (Users are very talented in this. They will enter the most extraordinary strings and numeric values into textual edit boxes. So the application software needs to be tested for a very wide set of possible inputs.)

Although the automatic tester software was originally developed for improving the reliability of the "Frequency Domain System Identification Toolbox (FDIdent)" of MATLAB (*Editor's note:* see the previous article), the principles are very general and can be applied to other MATLAB GUIs or to other environments [8]. The GUI of the tester software can be seen in Figure 1. The GUI of the tester can be easily adapted to any application software.

Testing Procedure

The testing procedure takes the following steps:

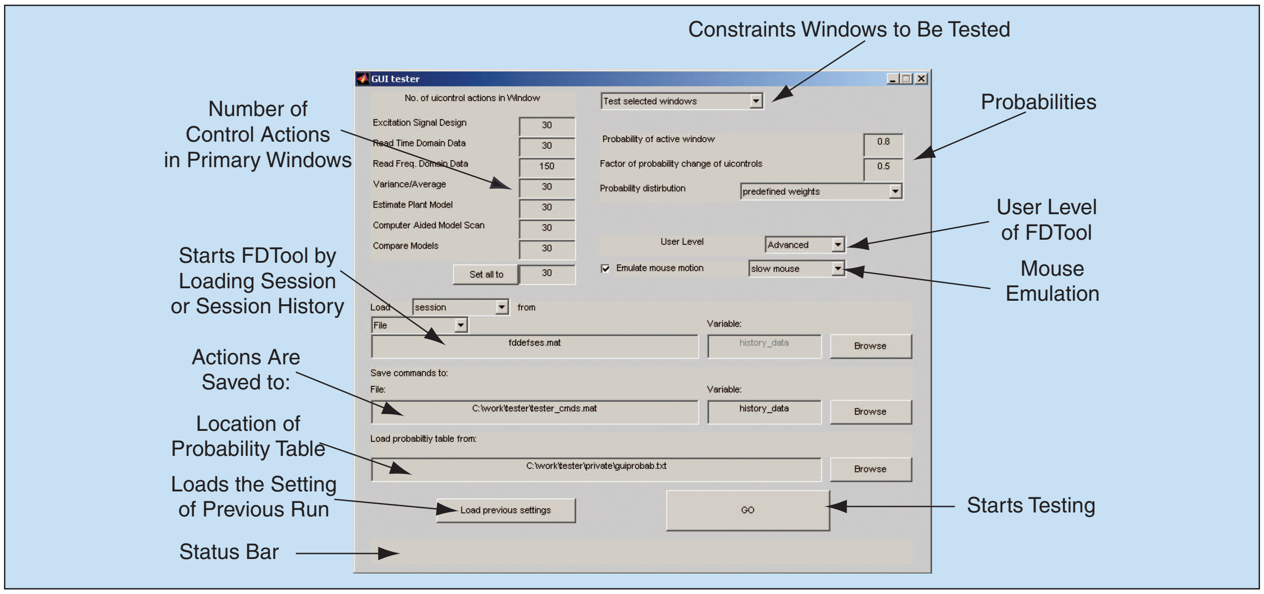
- 1) bring the software environment into a defined initial state
- 2) select a GUI object
- 3) store the selection before execution
- 4) activate the GUI object
- 5) if error then stop, else repeat from step 2.

Initial State

To be able to repeat the sequence of testing actions, the software environment needs to be brought into some predefined state before starting the testing. This may be a clean or a well-defined workspace. MATLAB provides both possibilities, which we used for the tester software. After an error has been caught, the environment needs to be brought into the same condition as before the beginning of the test. The action history can then be repeated step by step, and the workspace can be investigated after every step.

MATLAB Graphical User Interface

MATLAB has many graphical objects, like figures, lines, texts, surfaces, user interface controls, and menus. All of the graphical objects are uniquely identified with handles. The graphical objects have different properties, depending on their type. The most important properties are as follows:


Constraints Windows to Be Tested
Number of Control Actions in Primary Windows
Probabilities
User Level of FDTool
Mouse Emulation
Starts FDTool by Loading Session or Session History
Actions Are Saved to:
Location of Probability Table
Loads the Setting of Previous Run
Starts Testing
Status Bar
GUI tester
Set all to
Browse
Load previous settings
GO

No. of uicontrol actions in Window

Excitation Signal Design	30
Read Time Domain Data	30
Read Freq. Domain Data	150
Variance/Average	30
Estimate Plant Model	30
Computer Aided Model Scan	30
Compare Models	30
Set all to	30

Test selected windows

Probability of active window	0.8
Factor of probability change of uicontrols	0.5
Probability distirbution	predefined weights
User Level	Advanced
✓ Emulate mouse motion	slow mouse

Load session from File: fddefses.mat Variable: history_data

Save commands to: File: C:\work\tester\tester_cmds.mat Variable: history_data

Load probabiltiy table from: C:\work\tester\private\guiprobab.txt

Fig. 1. GUI of the tester.

- action performed if the graphical object is activated (e.g., click on a push button with left, right, or double mouse click)
- visibility of the object
- whether it is active or disabled (grayed out).

The basic operation for an action recorder is the mechanism to capture and replay each action performed by the GUI.

Selecting a Graphical Object to Be Tested

There are many graphical objects in a GUI, but not all of them are active at any one time. Some of them might be inactive, disabled, or invisible. There might be several windows containing different graphical objects. In MATLAB, a window can be made invisible instead of deleting it, which allows it to be available quickly if it's required again (e.g., a help window).

There are two possibilities to select a visible and active graphical object. The first approach is to collect the active objects from all visible windows and to select one from this set. The second approach is to select first a window and to collect the active objects only on that one. We chose this later approach because it describes better how a user interacts with the GUI. Moreover, this approach is faster.

We operate on a selected window called "window in focus." We associated a certain increased weight to the window in focus and unity weight to all other windows. The window in focus can be changed on a random basis, making use of the weights. This emphasizes that users usually activate controls on the foreground window.

The tester explores the visible and active graphical objects in the selected window. Selecting callback function through the action recorder activates one of the objects.

We also provided the ability to test only a selected window (the window in focus). This is useful if the testing is done after some software modifications that affected just one particular window and the others don't need to be tested again.

Activating the Graphical Objects Through the Action Recorder

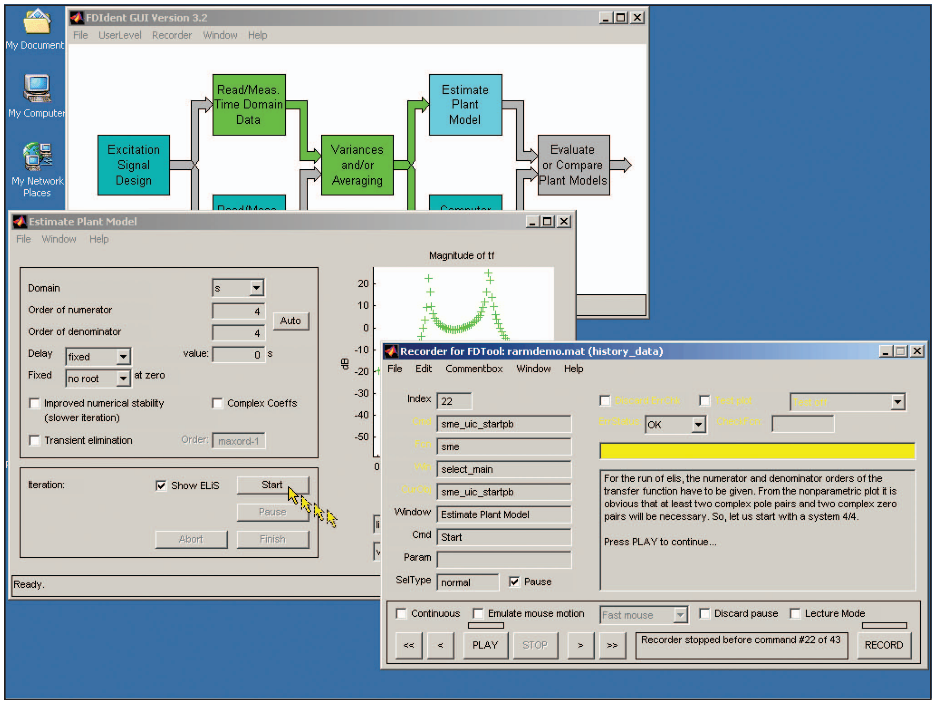
We chose to interact with the graphical user interface through a so-called action recorder, which will be described later. We feed actions into the recorder as if the user actions had been recorded and replay them (Figure 2).

Catching Errors

The tester stores the initial condition of the software and the actions on the hard drive before executing them. This ensures that the history of actions and the last action causing the error or an abrupt termination is saved, even if the application software freezes or becomes unstable.

If an error is caught, the testing procedure stops. The whole testing can be replayed step by step, starting from the same initial state as the tester did. The workspace can be investigated after every step.

There is also the possibility to continue the testing after an error has been caught (certainly not from the erroneous state). We can bring the system to a predefined initial state, and we restart the heuristic random test. If another error is caught, the action history is automatically stored on the hard drive with a different name. This possibility is advantageous at the first couple of test runs of a GUI test because it collects many errors without the need of user interaction. It might be useful also after a major change in the GUI software.



Domain: s
Order of numerator: 4
Order of denominator: 4
Delay: fixed value: 0 s
Fixed: no root at zero
Improved numerical stability (slower iteration)
Complex Coeffs
Transient elimination Order: maxord-1
Iteration: ✓ Show ELiS
Ready.

Index: 22
Cmd: sme_uic_startpb
Fcn: sme
Win: select_main
CurObj: sme_uic_startpb
Window: Estimate Plant Model
Cmd: Start
Param:
SelType: normal ✓ Pause
ErrStatus: OK
CheckFcn:

For the run of elis, the numerator and denominator orders of the transfer function have to be given. From the nonparametric plot it is obvious that at least two complex pole pairs and two complex zero pairs will be necessary. So, let us start with a system 4/4.

Press PLAY to continue...

Continuous   Emulate mouse motion   Fast mouse   Discard pause   Lecture Mode

Recorder stopped before command #22 of 43

Fig. 2. Storing a mouse click with the action recorder.

Guided Random Search

DEFINING PROBABILITIES TO UICONTROLS/UIMENUS

To speed up the testing procedure, it is worthwhile to influence the selection of the graphical objects to be activated. Some functions are known to be critical, whereas others do not perform complicated computations or do not contain many branches. It is worth investigating the critical parts more thoroughly. That is why we introduced weights to graphical objects. The larger the weight, the greater is the probability that the tester activates it. Assigning zero weight excludes the object from testing. The weights are stored in a lookup table, which can be edited with a standard text processor. This requires a unique identification of all graphical objects. MATLAB assigns a unique handle to each of them; however, this happens at runtime. Fortunately, it is possible to assign a tag (i.e., a string) to every object statically. We assign a unique tag to every object and identify them based on this. If the programmer of the graphical user interface did not pay attention to that or the identification is not unique, then the program assigns weights. Selecting the object based on a uniform distribution is still possible.

Our goal was to develop a tester that simulates the user and can be used to automatically test GUIs in the MATLAB environment.

One graphical object can be activated several times. Another heuristic, which we introduced, is to decrease dynamically the weight of the activated object. The factor for decreasing the weight can be adjusted. Assigning a zero factor gives the possibility to test an object only once.

DEFINING SETS OF POSSIBLE INPUTS TO EDIT BOXES

A complicated question is what numeric or textual input needs to be entered into the edit boxes. As the behavior of the application software depends on the data to be entered, an exhaustive test theoretically is not possible.

We define a set of possible inputs to edit boxes. These sets can be assigned either to one particular object or to a group of objects. The sets can be conveniently edited with text processors and can be either additions to completely random inputs or exclusive sets.

Defining the sets of possible inputs is a hard task. This is the point where an expert on the application software needs to cooperate with somebody who has never used the software before. An expert is bound to propose for the possible input set data that might be valid or in the range of the required input. It is obvious that such data should be used for the test. However, there is a much greater chance that the software to be tested is not prepared for an unusual input; e.g., a negative number for distance, a string for numeric input, or funny characters in filenames, etc. These inputs also need to be introduced into the possible set. A colleague who does not know much about the software might be much more useful for proposing unexpected values. (A child can be also of great help to extend the set of possible inputs.)

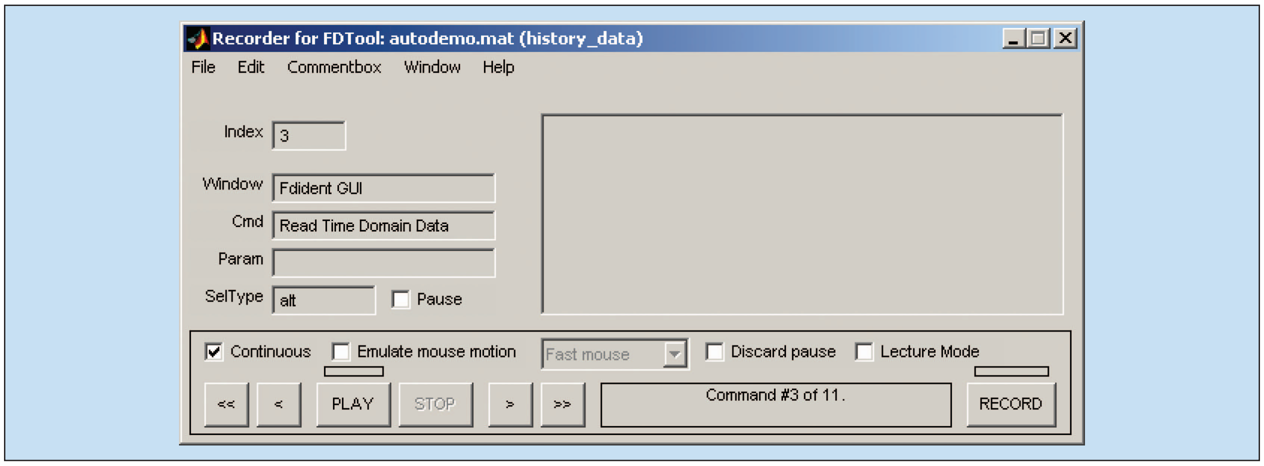
CAPTURING/REPLAYING ACTIONS: THE ACTION RECORDER

Let us summarize again the features we would like for the tester:

- use the GUI like a user
- if an error occurs, store enough state to be able to reproduce the error.

Both requirements tell us that we want to act *as the user would.* We need to produce and collect computer-simulated user actions and replay them when necessary. That is, we need something that records and replays actions. What is more intuitive for a person than to have an *action recorder*, similar to tape recorders, dealing with actions instead of voice and sound? We developed such an action recorder. The action recorder is not only a handy tool for testing, but its capabilities go much further. Before going into details, we need to consider the system requirements for the action recorder. We need to be able to:

- bring the GUI into a well-defined initial state
- act on any controls (pushbuttons, edit boxes, menus, etc.) as the user does; that is, we need to be able to emulate all user actions



Index 3
Window Fdident GUI
Cmd Read Time Domain Data
Param
SelType alt
Pause [ ]
Continuous [✓]
Emulate mouse motion [ ]
Discard pause [ ]
Lecture Mode [ ]
Command #3 of 11.

Fig. 3. Display of an action recorder.

Automatic testing, to our knowledge, aims to catch errors that produce an error message.

- record a sequence user actions
- replay the sequence.

For testing, it is enough to be able to record programmed actions only, but if properly programmed, the action recorder can record both programmed test actions and human actions. This allows additional testing. The most typical user actions (typical according to the designers) can be recorded and stored for later replay; this allows a quick test of the most common actions when using that GUI. Such a possibility can also be used to store demonstration sequences, which can provide an introduction to the GUI, and at the same time can be used as test sequences, assuring that all introductory steps work indeed. The examination of the results of such sequences by the human tester enables an approximate check of the semantics (proper calculations) of the software behind the GUI.

The basic operation for an action recorder is the mechanism to capture and replay each action performed by the GUI. There are two possibilities. One is when the operational system or the application program under which the GUI is realized can return the information of each action. In many cases, however, this functionality is not provided. In such cases, we are referred to the second one: the routines, called after each user action (callbacks), contain a program sequence that stores the corresponding action when executed.

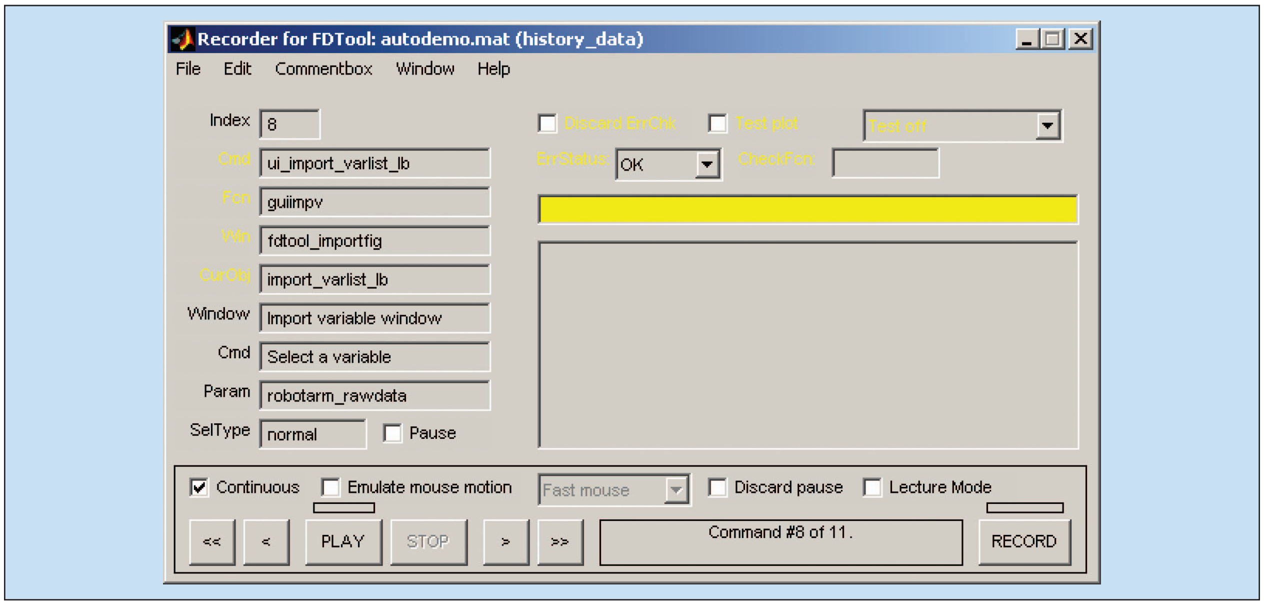
Replaying previously recorded actions can also be done in two ways. If the operational system or the application program offers the possibility to program the mouse to move above an object and perform the push or writing, the user action can be precisely emulated. If this is not the case, the callbacks need to be activated.

The MATLAB environment we used in this project allowed the callback-based operation only, so we implemented the recorder in this way. The functionality of each control is implemented also as a callable function; therefore, the recorder actions can also be programmed, allowing the implementation of the random tester (see below).

Functionalities of a Recorder

Figure 3 illustrates the graphical interface of the recorder. Meanings of the controls of the recorder are as follows:

- The buttons « ⟨ PLAY STOP ⟩ » provide the usual event recorder functionalities using a traditional tape-recorder front-end: fast backward (home), one step backward, play, stop, one step forward, fast forward (end).
- The text box on the right-bottom side is a message display, providing information about the state of the recorder.
- RECORD is the start button for recording; this starts the recording and steps forward after each recorded action.
- The tickbox "Continuous" selects between nonstop or step-by-step record and replay.
- The tickbox "Emulate mouse motion" makes the recorder move the mouse cursor above the active control during replay.
- The tickbox "Pause" allows one to set a breakpoint: continuous replay can be stopped if this tickbox is on for an action (this can be edited manually, after recording).
- The tickbox "Discard Pause" allows continuous execution even if the Pause tickbox is set for certain actions, to allow continuous execution of a demonstration that usually contains pauses.
- The tickbox "Lecture mode" allows special stopping: when the recorder stops, the active GUI window, not the



Index: 8
Cmd: ui_import_varlist_lb
Fcn: guiimpv
Win: fdtool_importfig
CurObj: import_varlist_lb
Window: Import variable window
Cmd: Select a variable
Param: robotarm_rawdata
SelType: normal
☐ Pause
☐ Discard ErrChk
☐ Test plot
Test off
ErrStatus: OK
CheckFcn:
☑ Continuous
☐ Emulate mouse motion
Fast mouse
☐ Discard pause
☐ Lecture Mode
Command #8 of 11.

Fig. 4. Action recorder in development mode.

recorder display, will appear in the foreground to allow explanations during a lecture.
- The large text box on the right-center side is for longer explanations for a user who is observing demonstrations.

There are additional controls in the recorder display; these are usually filled in by the action-recording step, but they can also be programmed. They are as follows:

- "Index"—serial number of the current action
- "Window"—name of the window where the action takes place
- "Cmd"—name of the control to be activated
- "Param"—value of the action if necessary (tick/untick in tickboxes, string in edit boxes, etc.)
- "SelType"—selection type of the user action (e.g., single or double mouse-click).

The recorder also contains menu items; these allow easy modification of action records: save to a file, load from a file, clear/cut/paste action, and insert a MATLAB command (a special possibility, e.g., to quickly set environment variables by calling a MATLAB command).

The recorder also has built-in error handling and checking capabilities. This feature allows the playback of actions that normally would cause warnings or errors during execution; thus, the error handling of the program under test can be compared with the expected behavior. This is more than originally expected; thus, we can test proper error handling. Many users just stop experimenting with the program when an error they cannot understand occurs. This situation is what a software developer tries to avoid.

Effective use of the recorder for tests can be helped by additional features not available in demonstration mode. The controls not present in Figure 3 but visible in Figure 4 are available only in development mode for the advanced users or programmers.

The recorder can be used with any MATLAB-based GUI; see [9].

Conclusions

An approach for testing GUIs has been proposed. We developed a software tool that tests GUIs by simulating the user through an action recorder. We proposed a heuristic test procedure: providing random input to GUI, but guiding the randomness with predefined weights assigned to the user controls. The weights change during the testing process, as the controls are activated. The errors are collected for later investigation.

Acknowledgments

This work was supported by the Hungarian Scientific Research Fund (OTKA F034900) and the Research and Development Fund for Higher Education (FKFP 0098/2001 and FKFP 0074/2001).

References

[1] *IEEE Instrum. Meas. Mag. (Special Section on Virtual Systems)*, vol. 2, pp. 13-37, Sept. 1999.

[2] User Interface Guidelines. Available: http://www.dcc.unicamp.br/~hans/mc750/guidelines/newfrontmatter.html

[3] *G Programming Reference Manual*. National Instruments Corporation, Austin, TX, 1988, part no. 321296B-01. Available: http://www.ni.com/pdf/manuals/321296b.pdf

[4] E. Baroth, C. Hartsough, and G. Wells, "A review of HP VEE 4.0," *Evaluation Eng.*, pp. 57-62, Oct. 1997.

[5] IVI Foundation Home Page. Available: http://www.ivifoundation.org/

[6] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12, 1990.

[7] MATLAB home page. Available: http://www.mathworks.com/

[8] Frequency Domain System Identification Toolbox for MATLAB home page. Available: http://elecwww.vub.ac.be/fdident/

[9] Action Recorder for MATLAB home page. Available: http://www.mit.bme.hu/services/recorder/

Tamás Dabóczi received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology, Hungary, in 1990 and 1994, respectively. Currently, he is an associate professor in the Department of Measurement and Instrument Engineering, Budapest University of Technology and Economics. His research area includes embedded systems and digital signal processing, particularly inverse filtering.

István Kollár received the M.S. degree in electrical engineering in 1977, the Ph.D. degree in 1985, and the D.Sc. degree in 1998, respectively. From 1989 to 1990, he was a visiting scientist at the Vrije Universiteit Brussel, Belgium. From 1993 to 1995, he was a Fulbright scholar and visiting associate professor in the Department of Electrical Engineering, Stanford University, California. Currently, he is a professor of electrical engineering at the Budapest University of Technology and Economics, Hungary. His main interest is signal processing, with emphasis on system identification, signal quantization, and roundoff noise. He is a member of the IEEE Instrumentation and Measurement Society AdCom and EUPAS (European Project for ADC-based devices Standardisation).

Gyula Simon received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology, Budapest, Hungary, in 1991 and 1998, respectively. Since 1991, he has been with the Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest. His research interest includes digital signal processing, embedded systems, adaptive systems, and system identification.

Tamás Megyeri is a gradate student at the Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics. He began his studies in 1996. His specializations are embedded systems and digital signal processing.