Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences

Lee White and Husain Almezen
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio 44106-7071

Phone: (216) 368-3919
E-mail: {leew, almezen}@alpha.ces.cwru.edu

Abstract:

Testing Graphical User Interfaces (GUI) is a difficult problem due to the fact that the GUI possesses a large number of states to be tested, the input space is extremely large due to different permutations of inputs and events which affect the GUI, and complex GUI dependencies which may exist. There has been little systematic study of this problem yielding a resulting strategy which is effective and scalable. The proposed method concentrates upon user sequences of GUI objects and selections which collaborate, called complete interaction sequences (CIS), that produce a desired response for the user. A systematic method to test these CIS utilizes a finite-state model to generate tests. The required tests can be substantially reduced by identifying components of the CIS that can be tested separately. Since consideration is given to defects totally within each CIS, and the components reduce required testing further, this approach is scalable. An empirical investigation of this method shows that substantial reduction in tests can still detect the defects in the GUI. Future research will prioritize testing related to the CIS testing for maximum benefit if testing time is limited.

Keywords: Graphical User Interfaces, GUI Testing, GUI Object Collaboration, Complete Interaction Sequences

1. Introduction

The GUI testing problem is difficult and challenging for a number of reasons:

1) The GUI possesses an enormous number of states that may need to be tested.
2) The input space size is extremely large due to the number of different permutations of inputs and events which affect the GUI.
3) There may exist very complex dependencies in the GUI system.

There is another aspect to GUI systems which make them very difficult to understand and to test. External effects can be caused by the GUI at any time, both known and predictable, as well as unknown and unpredictable. These effects are generally a function of the GUI objects, their states, and the order in which they are encountered, as well as the specific selections made at each GUI object. This issue will be addressed explicitly in the research discussed in this paper.

Before proceeding to testing, we first identify all the objects within the GUI system such as screens, icons, windows, commands, menus, etc. For each of these objects, identify all possible selections available, all possible states, possible links to other GUI objects, and also possible links to other non-GUI components of the system. In order to do this, it might be necessary to consult the GUI design document, the user manual, the code of the GUI system, or to informally interact with the GUI system. Clearly, the success of this or any other GUI testing approach will depend upon the accuracy and extent of this information.

We utilize the concept of *responsibility*, i.e., a GUI activity that involves one or more GUI objects; this activity produces an observable effect on the surrounding environment of the GUI such as a memory change, a change in the behavior of a peripheral device, a change in the underlying software or application software, etc. For each identified *responsibility*, we also identify the corresponding *complete interaction sequence* (CIS); this CIS consists of the sequence of GUI objects and selections that will invoke the identified responsibility.

0-7695-0807-3/00 $10.00 © 2000 IEEE

2. An Overview of the GUI Testing Method

1) The responsibilities within the GUI are identified using all the available information sources that we have concerning the GUI system.
2) For each identified responsibility, the corresponding CIS is defined.
3) For each CIS, a finite-state machine (FSM) model is constructed. In order to simplify the testing, a number of transformations will be applied that will lead to a *reduced FSM* for each given FSM. These transformations follow the logic investigated and developed by Chung-Wah Norris Ip [4, 5]. The reason that we are able to avoid the state explosion of the entire GUI FSM is that we concentrate on each individual CIS, and then further reduce the number of tests required by reducing the number of states in each corresponding FSM by these transformations. The current state reduction algorithm includes the following two transformations:
   - abstracting strongly connected components (a transformation we developed);
   - merging CIS states that have structural symmetry (developed by Ip [5]);
5. This reduced FSM will allow us to test the given CIS, and to identify a defect in the CIS if one exists. In Section 3, a detailed discussion will describe how to test each component identified by one of the reduction transformations. These tests will take the form of a directed path through each component, starting with a specific input to that component, and terminating with a specific output from that component. When the tests are designed as in step 5) below for the reduced FSM for the given CIS, the set of final tests will also include the set of tests for each component, in that each such component test will be included in at least one overall final test, and every time an overall final test encounters a *superstate* corresponding to a reduced component, this overall final test will include one such component test, matched up by the appropriate input to and output from that component.
6. Given the reduced FSM, with a number of superstates corresponding to one or more reduced components, two sets of tests will be designed for this FSM. One will be *design tests*, which assume that the originally given FSM was implemented as designed (although an error in this implementation might have occurred). The other set of tests will be more extensive *implementation tests*, which for each CIS, assumes that potential transitions not in the design might also occur to and from all states of the given CIS reduced FSM. A more detailed description of this testing will be given in Section 3.3.

3. Testing Properties of the State-Reduction Component Operations

For each of the two component reduction operations defined below, we will conceptually replace the given subset of states and all transitions between them as a *superstate* (single node) in the FSM for the CIS. Further, we will need to retain the subset of tests required within each component, one subset of *design subtests*, and another subset of *implementation subtests*. These subtests will be implemented by adding them to arbitrary tests of the reduced FSM so that each subtest will occur in at least one test of the entire FSM.

3.1. Strongly Connected Components

A subFSM is a *strongly connected component* if for every ordered pair of states (s1, s2) in the subFSM, there exists a directed path from s1 to s2. An example is shown in Figure 1; it is clear that this component is strongly connected. We want to design tests of this component so as to reduce the number of tests of the overall CIS FSM, and yet the component tests must allow for all possible effects between different orders of states and transitions. First, it should be clear that we must consider tests which match up each input with each output. In order to test for all possible effects of states and transitions, each test must include all states and transitions, which is always possible because the component is strongly connected.

To illustrate this in the example in Figure 1, we need two design tests that match up both inputs I1, I2 with the output O1:

{(I1,B,C,D,A,B,C,O1), (I2, A,B,C,D,A,B,C,O1) }.

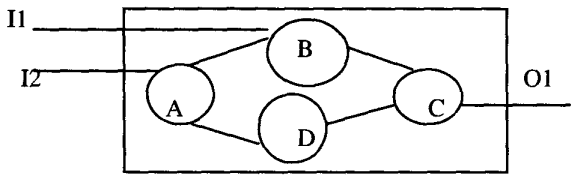

Figure 1 Design Subtests for a Strongly Connected Component

Implementation tests will involve all other transitions for this component which are not included in the design.

To find these, we must check all unused selections for GUI objects of the component, and if these correspond to actual state transitions to states within the component, these must be considered for tests; if these selections correspond to state transitions outside the component to other states of the CIS, then they yield new outputs of the component not yet considered. To obtain any new inputs to the component, all states outside the component within the CIS need to be considered for possible transitions to states of the component.

Figure 2 shows the example from Figure 1, where additional transitions have been discovered not in the original design. For the implementation subtests here, four are required in Figure 2 rather than just the two required for the design subtests in Figure 1. New transitions have been discovered: (I3, D), a new transition (D,A*), (D,B), and (D,O2); the reason that a new transition (D,A*) exists in addition to the original one in the design is that a different GUI selection gives rise to this transition in addition to the selection from the original design. The six new implementation tests which correspond to these four new transitions are:

{ (I1, B, C, D, B, C, D, A, B, C, D, A*, B, C, O1),
(I1, B, C, D, B, C, D, A, B, C, D, A*, B, C, D, O2),
(I2, A, B, C, D, B ,C, D, A, B, C, D, A*, B, C, O1),
(I2, A, B, C, D, B, C, D, A, B, C, D, A*, B, C, D, O2),
(I3, D, A, B ,C, D, B, C, D, A*, B, C, O1),
(I3, D, A, B, C, D, B, C, D, A*, B, C, D, O2) }.

When the design tests (implementation tests) for the entire FSM are produced, and the component is considered a superstate, at least one design subtest (implementation subtest) involving this superstate must be expanded to include each distinct subtest, and the input and output to and from the component must match the input and output used for the component from the entire FSM. Note that sometimes the total number of design tests (implementation tests) must be increased in order to satisfy this requirement.

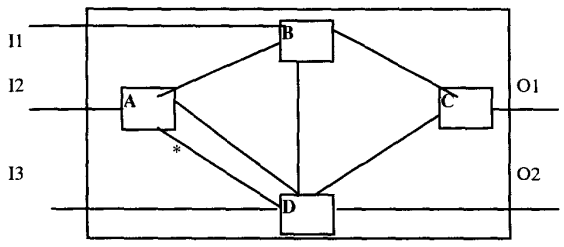

Figure 2 Implementation Subtests for a Strongly Connected Component

In order to provide some practical examples of strongly connected components, to see how they are tested, and why overall testing can be reduced with this type of component, consider Figures 3 and 4.

The CIS in Figure 3 achieves a Select Text-Edit-Cut-Copy operation. State A corresponds to the selection of text, whereas only the indicated states B, C, and D correspond to the strongly connected component. The testing of this component is somewhat subtle, as we first consider the two different ways to get to state E, corresponding to Cut and Copy being selected, respectively. Furthermore, for each of these, there are different ways of beginning the sequence, one by choosing C first, then D; both must alternatives must be tested in order to detect possible interactions encountered with each selection. Thus four tests are needed as follows:

{ (A, B, C, B, D, B, C, E), (A, B, D, B, C, B, C, E), (A, B, D, B, C, B, D, E), (A, B, C, B, D, B, D, E) }.

Figure 4 shows a Withdrawal CIS. When this is selected from A, there is a choice of Select Account or to go on to Select Other Transactions. If a specific Account is selected, a Withdrawal amount is selected, and a Printout of the Transaction is made. The strongly connected component is indicated as just states B, C, and D. Only one test of this component is required:
{ (A, B ,C, D, B, E) }, because all possible interactions will be detected by just this test.

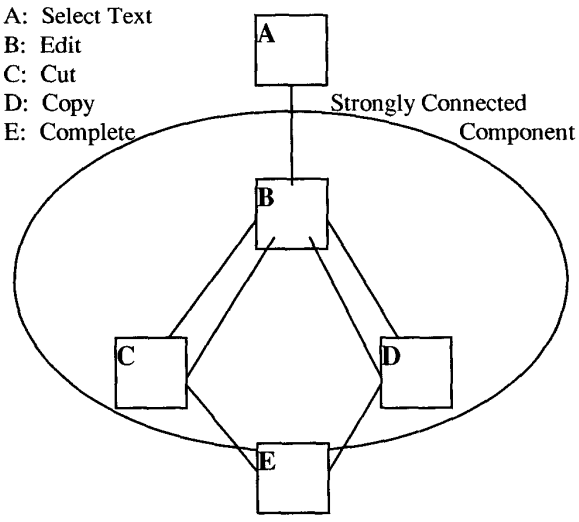

Figure 3 Edit-Cut-Copy CIS

B: Select Account C: Withdraw from Balance
D: Print Result E: Select Other Transactions

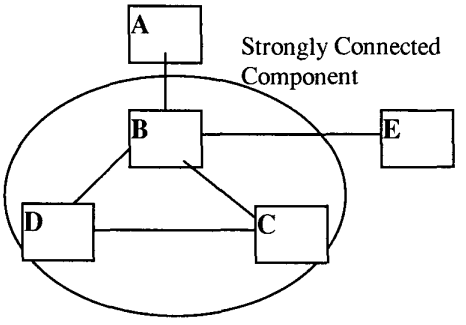

Figure 4 Withdrawal CIS

3.2. Structural Symmetry Components

A component has *structural symmetry* if the component has one input to state s1 and one output from state s2, with a number of directed paths from s1 to s2. Although we can considerably generalize this structure, here we will assume that each of these paths contain exactly one state other than the initial and final states; an example is shown in Figure 5.

However, unlike the strongly connected component, the structure alone does not totally determine symmetry. Ip [5] has studied these structures extensively, and developed a sophisticated state calculation to determine whether symmetries exist; Ip indicates that the idea is not new, as many other researchers have investigated the concept.

A: Open File Dialog
B: Select File by Name
C: Select File by Selection
D: File Selected

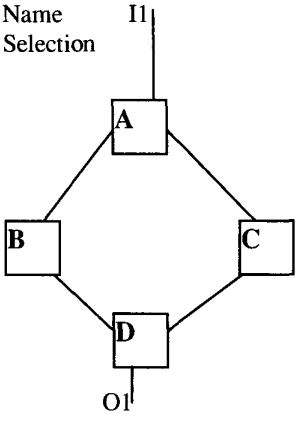

Figure 5 Example Structural Symmetry Component

For the purpose of GUI modeling, we require that structural symmetry components further satisfy the following conditions:

- For each component path selected, it makes no difference in that transition or the internal state (such as B or C in Figure 5) as to what path was followed in the GUI to get to the input of the component.
- Similarly, at the output of the component, for any transition or state executed after that point, it makes no difference which path of the component was selected.

For complex situations, it may be difficult to ascertain these conditions, and one might have to resort to utilizing the state calculations of Ip. However, most applications will be considerably simpler. For example, in Figure 5, a file is to be selected either by typing its name or by selecting the file by name. Clearly both conditions will be met that no previous state changes or transitions will affect this selection, or the selection will have no effect on subsequent operations with that file. If we add *undo* operations from states B to A and from C to A in Figure 5, symmetry conditions will still be met. Having established the concept of the structural symmetry component, next consider the process of testing it. Two design tests for the component in Figure 5 are selected as:

{ (I1, A, B, D, O1), (I1, A, C, D, O1) }.

For implementation testing, additionally discovered transitions will generally invalidate the symmetry of the structural symmetry component. However, although unlikely, it is possible that one would still achieve a symmetrical structure. For example, suppose two extra transitions (A, E) and (E, D) are discovered, and in the symmetrical component in Figure 6, the path (A, E, D) also just selects a file with no other effects; then the symmetry conditions are satisfied. The additional implementation test over the design tests is:

{ (I1, A, E, D, O1) }.

Clearly with the identification and testing of the structural symmetry component, there is a reduction in the overall number of tests of the CIS FSM, for any one of the paths through the component can be selected in an overall test of the CIS FSM, as long as each is utilized at least once.

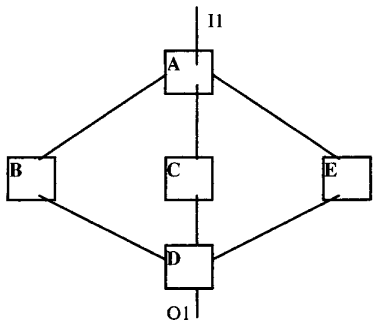

Figure 6 Implementation Tests for Structured Symmetry Component

3.3. Testing the Reduced FSM of the CIS

Testing the resulting reduced CIS FSM is considerably simplified after the state reduction transformations are applied, and components are replaced by superstates. Because of the nature of GUI Complete Interaction Sequences, we will assume that there is just one initial state and a single terminating state, but the method described here can certainly generalize to multiple initial states or multiple terminating states or both.

It may not be clear how long or difficult it is to identify the two types of reduction components, strongly connected or structural symmetry. Both will require an interplay of manual effort and automation. For the strongly connected component, one must avoid a strongly connected component that involves all the states of the CIS FSA; this can easily be identified, usually if there is a feedback arc from the terminal state back to the initial state; clearly no advantage would be gained from identifying such a component with no inputs or outputs. Generally, relatively small strongly connected components are most useful, involving many cycles and a few states. Such candidates can easily be spotted manually; in the example of Figure 7, the cycles between Exit, Cut and Copy are clearly indicative of a useful strongly connected component. Then if it is unclear how extensive the strongly connected component is, an automated technique can identify all the states in the component, together with all required test paths between inputs and outputs.

As for structural symmetry, we can write a simple automated routine to detect candidates for these components; in Figure 7, it is easy to select the two potential components either manually or automatically as states Open, NameFile, SelectFile and Highlight, or as states Open2, SelectFile2, NameFile2 and FileReady. The real problem is to establish that each of these components possess the symmetry property. We have recommended that this be done manually, but in general, it will be difficult to ascertain that the symmetry property will hold. For the two components in Figure 7 that are file selection routines, it is clear that they are totally symmetric. Another approach would be to use the complex state calculation of Ip [5], but this would als be very complicated to apply and understand.

To design test the reduced FSM, it will be necessary to construct sufficient test sequences starting at the intial state and stopping at the terminal state so that:

- all distinct paths of the reduced FSM are executed; each time a path enters a superstate corresponding to a component, an appropriate test path of the component is inserted into the test at that point;
- all the design subtests for each component are included in at least one test; this may require additional tests of the reduced FSM to satisfy this constraint.

In order to conduct implementation testing of the reduced FSM, it will be necessary to check all selections of GUI objects involved in the CIS to see if they invoke any new transitions in the reduced FSM. Note that it has been assumed that this has already been done in order to obtain the implementation test information for each component. As a result, some components may be invalidated, and must be restored as part of the general CIS FSM to be tested. Any new transitions applying to the reduced FSM will be noted, and included in the following implementation testing strategy.

To implementation test the reduced FSM, it will be necessary to construct sufficient test sequences at the initial state and stopping at the terminal state so that:

- all the paths in the modified reduced FSM are executed;
- all the implementation tests for each remaining component are included at least once.

Effects of the GUI are not explicitly modelled by the FSM, and thus we must examine the result of every distinct path, as a different effect in the GUI might be produced. Thus no cycle needs to be traversed more than once at any point in a path, but might have to be traversed any number of times each at different points in that path, for different effects might be caused by transitions or states in that cycle depending on when it occurs in the path.

Thus we need to generate all distinct paths from the initial state to any final state, but exclude any path where a cycle would be traversed more than once at a particular

point in that path. This testing is much more extensive than that required to test just potential errors in states or transitions of an FSM (see Marick [7]); that would require only that every state and every transition occur in at least some test path. It should be clear that only a finite number of test paths will be required to test for GUI effects, but for complex FSMs with many cycles, this number may be large. For the example in Figure 7, fifty test paths are required.

4. Example of Testing the Reduced FSM of the CIS

The following example CIS will show how the reduction transformations will simplify the testing by reducing the FSM for the CIS, and in turn considerably reduce the number of overall tests required for that CIS. The CIS in the example in Figure 7 is an Edit-Cut-Paste-Copy-Paste sequence which contains two open file sequences. If the required testing process is applied to the CIS in Figure 7, 50 design tests are needed. This number is obtained by tracing all distinct paths of the FSM from Initial to Finish, including all circular paths once on every distinct path where they can be reached. Now if we apply the three reductions highlighted in Figure 7, consisting of two structured symmetry components (both select file components as given in Figure 3), and one strongly connected component (Edit-Cut-Copy, similar to that given in Figure 5), producing the reduced FSM shown in Figure 8. Two design tests are needed for each of components 1 and 3; as previously argued, four design tests are needed for component 2. As a result, the reduced FSM only requires eight tests instead of 50 if components were not identified and tested.

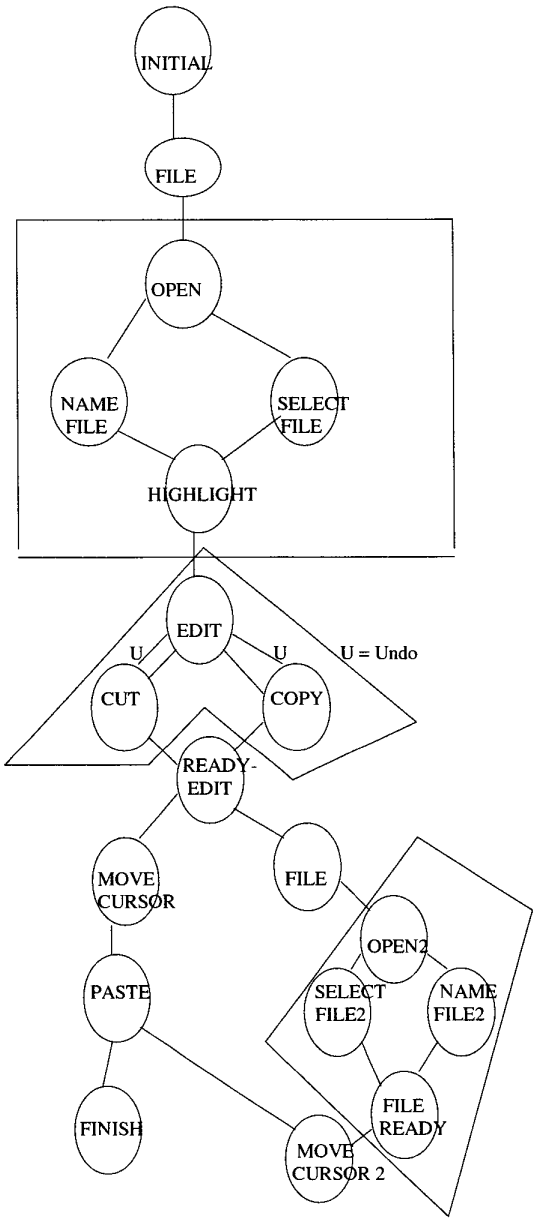

Figure 7 The Entire FSM for the Edit-Cut-Paste-Copy-Paste CIS Sequence

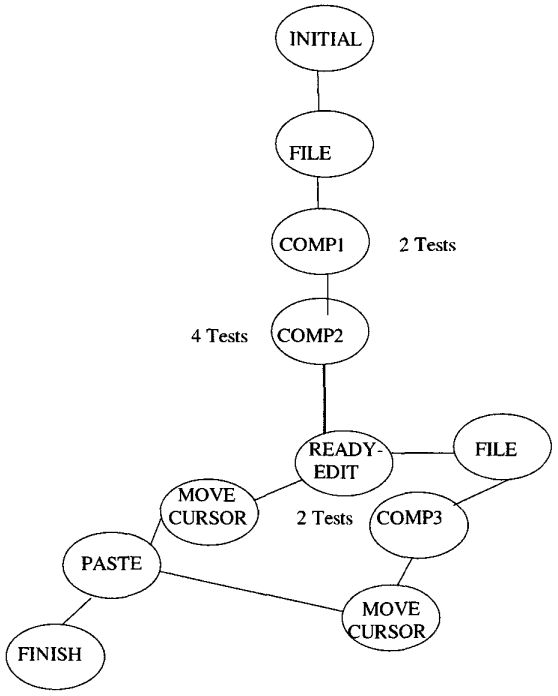

Figure 8 Reduced-FSM for Figure 7

5. Empirical Investigations Using the Proposed GUI Testing Method

We conducted two empirical investigations using the proposed GUI strategy. The first investigation utilized three versions of Microsoft Windows-98. The second investigation involved testing a portion of a multimedia distributed object-oriented database system called GVISUAL.

Each of these investigations were performed in the same manner. A person was identified who knew the GUI system very well, and their only assigned task was to identify each responsibility and define the complete interaction sequence (CIS) that invokes that responsibility. Another person was assigned the task of independently testing the CIS responsibilities using the proposed testing method. In each case, neither of these individuals were one of the coauthors of this paper, although one of the authors clearly had to explain the testing method to the individual who performed the testing.

It should be explained that in both empirical investigations, the person designated as tester was instructed how to develop tests for GUI FSA not utilizing reduction components, and also utilizing the two types of reduction components. As we have argued, the former approach will require more tests than the latter. When we prepared Table 1 to report the results of these experiments, it became too complex to report both sets of tests, and redundant to produce two distinct tables of data. Since our argument is to use the more sophisticated approach of reduction components when appropriate, we will report this data in terms of required tests in Table 1. Later in Section 5.3, we will report the number of design tests required for all our experiments using both approaches.

In testing and evaluating the proposed GUI testing method, we need to distinguish between *defects* and *surprises*. *Defects* are well understood as serious departures from specified behavior, whereas a *surprise* would be a user-recognized departure from expected behavior, but that behavior (or departure) is not explicitly indicated in the specifications of the GUI. If a detailed specification document exists, then a defect can be identified, but it may require interpretation as to whether the defect is serious or not (i.e., its level). A surprise must be evaluated by someone to establish whether most users would find the surprise objectionable behavior or not. For GUI systems, because of usability issues, many surprises might well be considered more serious than many defects.

5.1. Description of Empirical Investigation 1

It was known to us that MS-Windows 98 with Arabic Enabled had many defects and surprises, and we wanted to evaluate the proposed GUI testing method against it. We did indeed find a large number of defects and surprises, but planned to test two other MS-Windows 98 systems in exactly the same way to see what defects and surprises would be detected using the GUI testing method. In all three systems, only a portion of the interface was used for invoking, installing, uninstalling and running the following applications:

- Microsoft Word
- Microsoft Internet Explorer
- Freetel Chatting application
- Photo Studio
- Outlook Express Mailing program
- Microsoft Visual C++ and Development Studio
- AOL Communication software
- Games such as: FreeCell, Hearts, Minesweeper and Solitaire

As a result, 13 CIS responsibilities were identified; the following are five examples:

MS Windows 98	# of CIS	# of Tests	Defects	Surprises	Faults/CIS	Tests/CIS	Faults/Test
Arabic Enabled	13	Des: 28	6	3	1.23	2.15	0.32
		Impl: 32	12	4		2.46	0.5
Arabic Version	13	Des: 28	4	2	0.46	2.15	0.21
		Impl: 32	4	2		2.46	0.19
English Version	13	Des: 28	4	1	0.54	2.15	0.18
		Impl: 32	6	1		2.46	0.22
Multimedia DB Template/ Query	52	Des: 58	1	2	0.08	1.12	0.052
		Impl: 112	2	2		2.15	0.036
6 Seeded Errors	52	Des: 58	5	0	0.12	1.12	0.086
		Impl: 112	6	0		2.15	0.054

Table 1 Results of Investigations 1 and 2

1) Running each application either by clicking the icon twice, or
2) Choosing the application in the Start menu;
3) Installing each application from a CD;
4) Uninstalling each application either by its CD, or
5) By the uninstall wizard in the add/remove program option in the control panel.

For the MS-Windows 98 Arabic Enabled investigations, twelve defects and four surprises were identified as follows; throughout this discussion, the number of design and implementation tests required are given in Table 1. As mentioned before, the tests are those required using reduction components.

Defects:

1) Some tasks or applications cannot be run due to low memory.
2) The system crashed due to memory conflicts between two or more applications.
3) The Email application in Outlook does not work at all; problems in system setting.
4) Programs designed for MS-Windows 95 do not work in this environment, such as MS-Word 95 or MS-Excel 95.
5) For certain applications, the printer and its port could not be located.
6) Some programs cause system failure when uninstalled.
7) Some applications must be installed/uninstalled repeatedly in order to work properly.
8) The system is insecure/vulnerable to hackers when accessing Internet with Freetel.
9) The system sometimes does not recognize the modem with Explorer and Outlook.
10) The system will not install applications such as Visual C++ Development Studio on the first try; repeated efforts were required.
11) Sometimes when restarting the system, previous errors were still present.
12) Internet Explorer browser cannot access or download certain web pages due to improper security settings.

Surprises:

1) When installing some network applications, the system is unable to identify the port of the modem, so that it must be done manually.
2) For some network applications, must manually set half- or full-duplex.
3) There was an illusion that an application was still running when it was not, i.e., a message would be displayed "Cannot Run Program Twice".
4) For some applications, there was a lagging of the mouse pointer, so there were multiple copies of the mouse pointer, but only one is active.

For the MS-Windows 98 Arabic Version investigations, only four defects and two surprises were identified as follows:

Defects:

1) Freetel had to be reinstalled many times in order for it to work properly.
2) The printer does not print the contents of documents correctly.
3) All of the games sometimes freeze indefinitely.
4) Uninstalling an application by the wizard program does not work.

Surprises:

1) When printing is requested, must manually specify printer name and type.
2) When connecting to a modem, a message is displayed "Connection Failed. Check Your Modem Connection", while the connection is operating smoothly.

For the MS-Windows 98 English/US version investigations, six defects and one surprise were identified as follows:

Defects:

1) MS Visual C++ Development Studio was successful, sometimes crashed the system.
2) Running the Freetel application sometimes froze the entire system.
3) The MS Internet Explorer cannot be uninstalled without reinstalling Windows.
4) Outllook Express had to be reinstalled many times in order to work properly.
5) Running two applications at the same time will crash one of the applications.
6) Sometimes printing documents does not work.

Surprise:

1) Installation process of some applications done twice; second had to be cancelled.

Since some of these problems may have been dependent upon memory or other hardware configuration parameters, the computer system hardware environment is indicated for each of these sets of results. Since each of the three investigations had to be run at different locations, slightly different hardware configurations were used at each location:

- IBM Compatible
- 32 MB RAM
- 256 KB Cache
- Arabic Enabled: 200 MMX Pentium I, 3 GB hard drive
  Arabic Version: 400 MMX Pentium I, 2.1 GB hard drive
  English/US Version: 166 MMX Pentium I, 1.8 GB hard drive

5.2. Description of Empirical Investigation 2

GVISUAL is a portion of an object-oriented multimedia distributed database system that deals with defining schemas, and processing and interpreting queries. GVISUAL users interface through a GUI that makes these tasks simple. Our approach in this investigation is to test the GUI of GVISUAL to see if it interacts correctly with both the user and the underlying application.

GVISUAL has two subsystems, one that deals with Templates (or schemas), and another that handles Queries. For the Templates, GVISUAL allows the user to define the following, with 28 responsibilities for the Template part of the GUI:

- new classes;
- methods local within the class and global among several classes;
- containment relations between classes called *Object Composition*;
- temporal operators *connect* for connecting two objects, *next* for sequencing between the objects, and *until* repetition of object instance for the defined classes;
- shapes that each group of defined classes could assume visually.

For the Queries subsystem, GVISUAL allows the user to insert the following; as a result, 29 CIS responsibilities are identified for the Query part of the GUI:

- new object instances of a defined class;
- predefined methods;
- a condition box that forces certain constraints on the relations between objects;
- queries, external or internal;
- temporal operators defined for the objects;
- the Queries subsystem can execute the query, produce a parse tree for the query, and send it to the administration server through a socket connection.

Table 1 shows the results of this investigation, the number of design and implementation tests required to detect the following defects and surprises:

Defects:

1) The icon for the Query *until* operator does not position properly on the screen, as it intersects with other icons.
2) For the Template definition, any errors entered by the user in this process would clear the screen, with no correction possible.

Surprises:

1) During Template definition, none of the icons for a class, method or operator are left on the screen; the user must select the item again from a list to edit it.

2) The status bar displays the message twice.

It should be noted that the designer of GVISUAL was very shocked to see the defects and surprises determined by this GUI testing method, for he thought that he knew of all the problems with this GUI design, and they had been resolved.

In addition to testing for unknown defects of the GVISUAL GUI, additional defects were intentionally seeded into the GUI in order to determine whether the proposal testing method could detect these seeded defects. Four individuals were identified to implement this process, with no contact between them:

a) one defined all the required CIS responsibilities;
b) another wrote the code implementing these CIS responsibilities;
c) another seeded the defects in this code;
d) another implemented the proposed testing method on each CIS.

The six seeded defects were as follows:

Template:

- Creating a method without identifying its base class.
- Creating a containment relationship without giving a name of the contained class.
- Creating a temporal operator *until* without associating any class (hence object).
- The polygon shape is not ended definitely with a right mouse click.

Query:

- Although we can define a condition box, we cannot activate (or insert) it.
- The internal query does not specify any I/O parameter details.

All these six seeded defects were successfully detected by design and implementation tests for these three CIS responsibilities.

5.3. Results and Discussion of the Empirical Investigations

One of the questions we attempted to answer was whether the implementation tests were really necessary or cost effective, given that design tests were all executed. In Table 1, if we look at the situation for the Arabic Version of MS-Windows 98, the design tests detected all 4 defects and the 2 surprises, and the additional four implementation tests detected no additional defects or surprises. Yet from the rest of the data in Table 1, it is clear that although design tests might be useful as a first step, eventual use of implementation tests should also be conducted in order to assure thorough testing. This can also be seen from the last column of Table 1, called Faults/test. (A *Fault* here is identified as either a defect or a surprise). There is not much difference between design tests and implementation tests for the same configuration in terms of this metric.

The data in the column of Table 1 labeled as Tests/CIS represents the average number of tests required to test each CIS. For the Windows 98 configurations, and for both design and implementation tests, the data is very consistent, ranging from 2.15 to 2.46. Approximately the same result is also obtained for implementation tests of the Multimedia DB at 2.15. The design tests for the DB gave 1.12, which was much less. Still this data is so consistent, it might be used to estimate the number of tests required to test CIS for GUI systems.

The data in the column of Table 1 labelled Faults/CIS represents the average number of defects or surprises per CIS, and therefore is a measure of how contaminated that GUI is with faults. Clearly Arabic Enabled is much more contaminated as we suspected, whereas the Arabic version and English/US version are about the same by this metric. The contamination level of the Multimedia DB is far less using this criterion.

Returning to the data in the column of Table 1 labelled Faults/Test represents the detection capability of each test for each tested GUI configuration. For the Windows-98 configurations, again this criterion is quite consistent over all configurations, and for both design and implementation tests, only ranging from 0.18 to 0.50. However, the fault testing rates for the Multimedia DB are generally five times smaller, due to smaller numbers of faults and more tests required. It is interesting that data for seeded errors in the DB is comparable to the faults detected in the DB, but this is just coincidental, as the selected number of seeded defects, six, was arbitrarily chosen.

Data which is not shown in Table 1 is available giving the number of tests for all experiments when all distinct paths are used through each CIS FSM. A comparison with design tests using reduction components is given below. It should be reported that no additional faults, defects or surprises, were detected by testing all possible paths for either design or implementation tests over testing the reduced FSM for any CIS considered.

Design Tests	For Reduced FSM	For All Directed Paths
Windows-98	**28**	**40**
Multimedia DB	**58**	**86**

The difference in the number of tests required for the reduced FSM and tests required for all directed paths corresponds to the savings due to reducing components for the GUI configurations tested. The total difference in these investigations is not so great, so this indicates that none of the CIS were too complex (as complex as shown in Figure 7). This conclusion is also borne out and consistent with the data for average Tests/CIS in Table 1, between 1 and 2.5 tests per CIS, indicating that the CIS were not too complex. However, if a GUI were to be tested which had one or more very complex CIS FSM, then testing the reduced FSM should be easier with fewer tests required.

6. Related Work

Given the importance of the GUI testing problem, the research literature is not very extensive. White [14] applied the concept of Latin Squares for GUI test generation, achieving substantial reduction in the number of test cases, and also showing how complex the GUI testing problem can be. However, the assumption that interaction between GUI objects is primarily pairwise is too limiting, as very complex GUI interactions also cause failures.

It has been long recognized that the use of finite state machines can model at least part of the GUI testing problem. The classical work in the area of testing FSM has been that of Chow [2]. Another area with substantial testing experience relevant and appropriate for GUI testing is that of protocol testing. Bernhard [1] has utilized the FSM testing methods of Chow to obtain a substantially reduced test suite for protocol testing. Continuing in this tradition of using FSM is the research of Shehady and Siewiorek [11] to address the GUI testing problem. Recognizing the inordinate explosion in states, they achieved a substantial reduction in the number of states, and hence number of tests required, by utilizing new global variables defined and associated with existing states of the FSM; they term this new construct a VFSM, or *Variable Finite State Machine*. Despite the state and test reductions achieved with this construct, the number of tests and states is still extremely large. It would appear that the scalability of the CIS concept of this paper could be utilized to help partition their VFSM approach.

A most recent approach to GUI testing is the goal-driven approach of Memon, Pollack and Soffa [8, 9] that exploits the artificial intelligence technique of *planning* to generate test cases for the GUI. From plans to achieve goals at the most abstract level, a list of operators are used to map this plan to effective GUI selections. They have achieved an impressive reduction in the number of operators to keep the complexity of this approach from getting out of hand. Good experimental results have been achieved from this approach thus far. Again the concept of CIS and responsibility of this paper might prove to be useful for focus in planning, or in the resulting GUI objects and selections to be considered.

Dwyer, et. al. [3] has also approached the GUI testing problem by model checking. In this approach, a simplified GUI model is checked against the implementation using an abstract transition system. Although an interesting approach, it might report false negatives if the implementation includes harmless and unimportant features irrelevant to the essential operation of the GUI.

Although not really constituting research on the GUI testing problem, there have been a number of industrial papers which have provided some interesting insights into this problem. Most of these authors discuss the need for automation tools for GUI testing, including the following four authors. The' [12] also discusses defects in an environment caused by the GUI system, e.g., memory leaks and garbage collection, as well as stress testing for GUI. Kepple [6] describes "The Black Art of GUI Testing", commenting on the often ad hoc approach to this problem; it is precisely the goal of our contribution to pursue a more systematic method for GUI testing. Kepple goes on to suggest that tools identify the location of an object displayed on the screen, and eliminating synchronization and timing problems between GUI response and tool, rather than just blindly relying upon capture-replay to solve all testing problems. Walworth [13] concentrates on testing Java implementations; he indicates that although automated tools are essential for GUI testing, for Java the tools must be platform independent. Also human intervention may be necessary at critical junctures, and can cut down extensively on subsequent testing. Remy [10] also mentions the usefulness of state diagrams for GUI testing, and he recommends that the GUI be divided into appropriate subsets of objects for testing. These have been two aspects of the approach presented in our paper.

The research described in this paper has addressed the most important testing which should be performed on the GUI system from the user's perspective: testing for defects within a complete interaction sequence which invokes a responsibility of direct interest to the user. The

methodology described in this paper is simple and easy to apply, as demonstrated by the empirical investigations using third parties to carry out the testing after only a brief orientation. This CIS testing could be included as a base for other methods and approaches to use to ensure that their proposed method identifies all the defects and surprises detected by CIS testing.

7. Conclusions and Future Work

A systematic method has been proposed to test all known complete interaction sequences (CIS) in a GUI system that invoke a desired response for the user called responsibilities. As more such CIS are later discovered, they can be incrementally tested, since this method does not require initial knowledge of the full behavior of the GUI system. Two empirical investigations showed that this approach could detect both defects and surprises, and that two types of tests, design tests and implementation tests, were useful in this regard.

The number of required tests can be substantially reduced by identifying reducing components of the CIS that can be tested separately . The two empirical investigations showed a significant reduction in the number of required tests by these components in both investigations. Since consideration is given to defects totally within each CIS, and the components reduce the number of required testing even further, this approach is scalable.

Nearly all the future research plans call for the investigation into interactions of other portions of the GUI systems with each CIS, or into interactions between two or more CIS units. The key idea here is to prioritize tests of these interactions and subsystems that are likely to produce defects and surprises of concern to the users of the GUI. The reason for the prioritization is in case there is insufficient time to execute all tests. It is likely that slight deviations of each CIS corresponding to "mistakes" by the user would be a worthy candidate to explore for priority testing.

We would also seek to find additional reduction transformations and components to further reduce the required testing, especially for very complex GUI system CIS configurations.

8. References

[1] Phillip Bernhard, "A Reduced Test Suite for Protocol Conformance Testing", ACM Trans. on Software Eng. and Methodologies, Vol. 3, #3, pp 201-220, July 1994.

[2] Tsun Chow, "Testing Software Design Modeled by Finite-State Machines", IEEE Trans. on Software Engineering,Vol SE-4, #3, pp 178-187, May 1978.

[3] Mathew Dwyer, Vicki Carr and Laura Hines, "Model Checking Graphical User Interfaces Using Abstractions", Proc. of the 6th European Software Eng. Conf., pp 244-261, Sept. 1997.

[4] C. Norris Ip and David Dill, "Efficient Verification of Symmetric Concurrent Systems", Proc. of Int. Conf. on Computer Design: VLSI in Computers and Processors, Cambridge, MA, pp 230-234, Oct 1993.

[5] C. Norris Ip and David Dill, "Better Verification Through Symmetry", Formal Methods in System Design, Vol. 9, # 1/2, pp 41-75, 1996.

[6] Laurence Kepple, "The Black Art of GUI Testing", Dr. Dobb's Journal, pp 40-46, Feb 1994.

[7] Brian Marick, "The Craft of Software Testing: Subsystem Testing", Prentice-Hall Series in Innovative Technology, Englewood, NJ, 1995.

[8] Atif Memon, Martha Pollack and Mary Lou Soffa, "Using a Goal-Driven Approach to Generate Test Cases for GUIs", Proc. of the 21st Int. Conf. on Software Eng., Los Angeles, CA, pp 257-266, May 16-22, 1999.

[9] Atif Memon, Martha Pollack and Mary Lou Soffa, "Plan Generation for GUI Testing", the 5th Int. Conf. on Artificial Intelligence Planning and Scheduling, Breckenridge, CO, pp.226-235, April 15-17, 2000.

[10] Nicholas Remy, "An OO Approach for GUI Testing", Object Magazine, pp 19-24, June 1998.

[11] Richard Shehady and Daniel Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines", Proc. of the 27th Int. Symposium on Fault Tolerant Computing, Seattle, WA, pp 80-88, June, 1997.

[12] Lee The', "Stress Tests for GUI Programs", Datamation, pp 37-40, Sept. 1992.

[13] Alan Walworth, "Java GUI Testing", Dr. Dobb's Journal, pp 30-34, Feb. 1997.

[14] Lee White, "Regression Testing of GUI Event Interactions", Proc. of the Int. Conf. on Software Maintenance, Washington, DC, pp 350-358, Nov 1996.