Abstract

The challenging goal of designing interactive systems is to prevent faults caused by the control unit that could potentially lead to failures. When designing interactive systems, it is the behavior that is dominant in the process of modeling. Nevertheless, structural features in software application programming and their organization are seldom taken into account for test derivation. The consequence is a reduction in fault coverage with regard to find structural defects in such systems. This paper introduces an approach to reveal structural features of such systems that expose further test opportunities. The approach uses a graphical, event-based model of which structural features are extracted by simple transformations resulting into an increased fault coverage.

1. Introduction

Testing is an important step in development as it checks the compliance of the system to the user requirements. One of the advantages of software testing as opposed to methods such as formal verification is that the former allows direct execution of the system under test (SUT) as well as an observable output. Therefore testing has become the most popular verification and validation method in industry.

User interfaces (UIs) of interactive systems are intended to control system behavior and system functionality. Therefore the development of UIs is mainly handled separately from its application system because it requires different techniques and skills. This fact has been recognized very early, defining a user interaction management system that is independent from the application, graphics package, etc. [1]. Most UIs are graphical (GUI) and serve as a valuable and comfortable means for the user to communicate with the system in accordance with the application.

The implementation of a GUI requires familiarity with the programming platform, language, etc. [2]. The programming languages and their software development tools nowadays support the programmer with a wide range of GUI widgets, i.e., windows, menus, dialogs, checkboxes, etc. Therewith, the programmer has control over which standard window controls are available, but although the operating system handles the window’s behaviour, the programmer must handle the impact on the application. In some circumstances for instance, closing a window before completing a transaction may leave the application or the database in an inconsistent state. Such complications are aimed to reveal by testing the windowing system of a GUI.

Testing of GUI requires both, a good understanding of user requirements, and familiarity with the technical equipment. This paper is about extracting a structural representation of the GUI from its behavioral model to increase the efficiency in finding defects in the structure of a GUI. The structural representation generated from the software describes the relationship between widgets of a GUI. Our approach analyzes the structure of a GUI, which is represented in corresponding, event-based graphical models focusing on the windowing system of the SUT.

The paper is structured as follows: The next section summarizes event sequence graphs (ESG). This background is necessary to present our approach. Section 3 analyzes GUI structures and introduces the view-ESG model. Section 4 explains how to extract GUI structures from a given ESG and how to transform this model into a view-ESG to positively influence fault coverage. In Section 5 the generation of a view-ESG is demonstrated for a GUI sample supported by a developed tool. Section 6 gives a short review of past work pointing out the differences with this paper. Finally, the benefits of this contribution are summarized and areas for future work are proposed.

2. Event Sequence Graphs

In [3], [4], [5] a graphical representation has been introduced of both the behavioral model and the fault model of the system under test which enables a scalable generation and selection of test cases. That work uses event sequence graphs (ESG) for representing the user and the system behavior as well as user-system interaction. Mathematically, an ESG is a digraph and may be thought of as an ordered pair

\[ ESG = (V, E), \]
being a set of vertices (events) representing input symbols of an alphabet Σ and E a non-empty relation on V, with elements in E representing directed edges between the vertices in V. As a convention, a dedicated start vertex is the entry of the ESG whereas a final vertex represents the exit, denoted by symbols "[" and "]", respectively.

The semantics of an ESG is as follows: Each vertex v ∈ V represents an event that is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response, punctuating different stages of the system activity. In this paper, we assume that events of an ESG represent elements of a GUI. For two events represented by vertices v, v′, it holds that (v, v′) is an element of the set E if event v′ is enabled after the execution of v. If the same events are used in different contexts, such events have to be indexed to highlight their contextual position. It is clear that an ESG disregards the detailed internal behavior of the system and, hence, is a more abstract representation compared to, for example a finite-state automation (FSA). A simple example of an ESG consisting of three events is given in Figure 1.

![Figure 1. Event sequence graph ESG = (V, E)](image)

Any sequence of vertices (v₁, ..., vₖ) is called an event sequence (ES) if (vᵢ, vᵢ₊₁) ∈ E, for i = 1, ..., k − 1. An event sequence is a complete event sequence (CES), if v₁ is an entry and vₖ is an exit vertex.

To support abstraction and modeling in different levels, each vertex of an event sequence graph may be refined or structured through another ESG. Figure 2 shows an example: Vertex e₂ of event sequence graph ESG₁ is refined by ESG₂. Dissolving the refinement results in ESG₃.

![Figure 2. Refinement of e₂ of ESG₁ by ESG₂](image)

Failures of a system are caused by faults, including incorrect user inputs. Such inputs can be traced back to undesirable events that were triggered by a previous input. As an assumption for the traceability of incorrect user inputs the system must be still observable in its failure state, i.e., the fault does not cause a failure severity that crashes or shuts the system down. To consider these potential user errors as well, faults are modeled as undesirable events. This represents a complementary view of the behavioral model: The complementary ESG is defined as ESG = (V, E) where E = V × V \ E. As an example, the complementary graph of Figure 1 is shown in Figure 3. Any event pair (EP) of an ESG is called a faulty event pair (FEP) for ESG. Based on notions of CES and FEP a coverage-based test process can be defined [5].

![Figure 3. Complementary graph ESG = (V, E)](image)

Event sequence graphs can be applied to graphical user interfaces (GUI): An ESG of a GUI specifically describes possible sequences of user inputs through which the system can proceed during its lifetime as a result of reacting to discrete events. We assume that ESGs are given for SUT. Then, the property of structuring those ESGs is used to generate a new model. As an ESG is an intuitive and a more abstract representation of a reactive system than state-based diagrams its design can be also derived from the SUT in a legitimate time afterwards. The modelling of ESGs and its comparison to other models was investigated in a previous work [5].

3. GUI Structure

GUIs are the interface for the user to operate the application. A good interface can decisively influence the success of the system. The user should be offered a high flexibility in using the GUI. On the other hand flexibility can cause a high degree of complexity that makes the handling tedious. The key factor for a good GUI design is the understanding of their windowing. Windows have specific properties and are used to structure the GUI.

3.1. Hierarchical Structure

Thus, the number of interacting windows that are accessible at the same time should be reduced. For this reason...
the windows of a GUI are hierarchically structured. The application itself is represented thereby as a parent or root window. This window can invoke, i.e., lead to, another window which is called a child window of the starting window. Thus, GUIs are hierarchically structured as a tree of windows as illustrated in Figure 4.

![Figure 4. Hierarchical structure of windows as a tree](image)

Each window contains GUI objects which are needed to process the responsibilities they are grouped for. These objects of an application can be identified by a Capture-/Replay-feature coming with most commercial GUI test tools. The recorded objects are hierarchically structured by the windows they belong to. Figure 5 depicts the objects of an application captured by the test tool WinRunner of HP (formerly Mercury Interactive).

![Figure 5. A sample application and their captured objects by a commercial test tool](image)

These recorded objects have different properties, e.g., window class, label, etc., whereas only a minimum of the properties are recorded which can uniquely identify the object. Thus, each object is distinguishable from each other. As a result, each vertex of an ESG can be mapped to its corresponding object of the GUI.

3.2. Window Modes

Windows of commercial systems are nowadays mostly hierarchically structured as already pointed out, i.e., the root window invokes child windows that can invoke further (grand) children, etc. Some child windows can exist simultaneously with their siblings and parents; they will be called modeless (or non-modal) windows. Other children, however, must “die”, i.e., close, in order to resume their parents. They will be called modal windows.

Whenever an input from the user is needed, usually a modal window is used. Using modal windows has long been shunned by many developers as too constraining the user. However, modal windows do have many uses in complex applications since most people only work on one window at a time.

![Figure 6. Modal windows vs. modeless windows and an example of a modal opened window](image)

Figure 6 represents these window types as a “family tree” (cf. Figure 4). In this tree, a directed edge indicates a modal parent-child relationship. An undirected edge indicates a modeless one. Once again, modal windows must be closed before any other window can be invoked, so it is not feasible to consider any object of the parent window.

3.3. The view-ESG

A view-ESG is a digraph with two distinguished sets of edges and may be thought of as an ordered triple

\[ \text{view} - \text{ESG} = (V, E_{\text{modal}}, E_{\text{modeless}}), \]

\[ (2) \]

\( V \) being a set of vertices (windows), \( E_{\text{modal}} \) as a relation on \( V \), with elements \((v, v') \in E_{\text{modal}}\) of directed edges between the vertices in \( V \) representing a call of a modal window and \( E_{\text{modeless}} \) as a relation on \( V \), with elements in \( \{v, v'\} \in E_{\text{modeless}} \) of undirected edges between the vertices in \( V \) representing a call of a modeless window. The dedicated start vertex and final vertex are adapted from the original ESG definition.

The view-ESG as in Figure 7 differs from the original ESG by its semantics. A view-ESG describes the interaction
of the different windows an application has. Each vertex represents therein a window. A window can be either a modal or modeless one. If a parent window evokes a child window which is a modal one the vertex of the parent window connects the vertex of the child window by a directed edge. If the child window is a modeless window the two vertices are connected by an undirected edge.

4. Generating Structural Models for Improving Testability

Similar to strong-connectedness and symmetrical features [6], the modality feature is extremely important for testing since it represents the structure of GUIs which also has to be tested. During the design phase of software development the focus is on the behavior. In fact, most existing and available models for test generation are behavioral models given as state-based or event-based models. Likewise, this work necessarily needs a structural model for testing. For that purpose the introduced structural properties are recognized and extracted from the behavioral model.

The ESG as a behavioral model already includes the structure of the different window modes of a GUI. The structure of which object belongs to which window is given by capturing the corresponding GUI as described in Section 3.1. The result is a window tree. To test the structural view of the GUI these two pieces of information have to be filtered and then used to generate a corresponding structural view model as depicted in Figure 8. This model can be used to derive test cases for predefined test criteria.

The resulting model of the GUI is also an ESG focusing on the windowing. To distinguish the different kinds of ESG the latter one is called view-ESG as introduced in Section 3.3.

4.1. Extracting the Structural Model

For generating the structural model it is assumed that the GUI objects are already captured and hierarchically structured. Additionally, an ESG model is needed that has been created for the behavior of the GUI. The structural model to be extracted is in turn another ESG, introduced as view-ESG in Section 3.3. The generation of a view-ESG is determined by the identification of the window modes.

The identification can be restricted to identifying modal windows. Then, all other windows have to be modeless. If a modal window is found, all objects belonging to this window can be reduced to one vertex in the behavioral model of the GUI. The behavioral model is converted step by step into a structural model of the GUI. Finally, the objects of the remaining non-modal windows have to be changed by one vertex representing the modeless windows. Figure 9 represents a modified ESG, which separates the event.
“Modal Form” taking the modality into account. The original ESG is thereby separated into two ESGs representing the main window and the modal window.

The identification of modal windows is determined by the successor events of the objects belonging to a window. Three different situations can occur as depicted by Figure 10:

1) The simplest one is when an event has only successor events in the same window where the event itself belongs to. Then no other window is evoked and has not to be considered further.

2) The relation is hybrid if there exists a successor event in the same and in another window. The existence of at least one hybrid successor in the relation between the evoked child window and its parent window is then determined as hybrid. Consequently, the child window is a modeless window.

3) If an event has only successor events in other windows the relation is called non-hybrid. If all successor relations of an evoked window are non-hybrid then also the overall relation is non-hybrid and therefore a modal window. Note that for a modal window all relations of successor events have to be non-hybrid. The violation of just one non-hybrid relation could never become a modal window.

4.2. Algorithm

Based on the previous section and Figure 8 Algorithm 1 schematically depicts the extraction of the structural model.

The algorithm takes as input a set of captured GUI elements, the behavioural GUI model represented as ESG and a function $m$. This function maps to each GUI element the corresponding vertex of the ESG. As a result a view-ESG and ESGs for each window are computed.

The set of vertices $W$ of the view-ESG consists of the windows of the GUI. For each window $w$ of the GUI a model $ESG_w$ is created. The vertices of each ESG consist of the relevant GUI objects that belong to window $w$ denoted by the set $GUIO_w$. Edges are created based on behavioral model of the GUI. Finally, the edges of the view-ESG can be computed, i.e., the sets $E_{modal}$ and $E_{modeless}$, by applying the rules of the previous section.

**Algorithm 1:** Extracting the structural model

**Input:**
1) A set of captured GUI objects $GUIO$
2) $ESG = (V, E)$ (Behavioral GUI model)
3) Mapping $m: GUIO \mapsto V$

**Output:**
1) View-ESG $vESG$
2) An $ESG_w$ for each window $w$

1. $vESG := (W := \emptyset, E_{modal} := \emptyset, E_{modeless} := \emptyset)$
2. $W := \text{extract the set of windows from } GUIO$
3. foreach $w \in W$ do
4.    create $ESG_w := (V_w := \emptyset, E_w := \emptyset)$
5.    foreach $o \in GUIO_w$ do
6.       $V_w := V_w \cup o$
7.    foreach $o_1, o_2 \in GUIO_w$ do
8.       if $(m(o_1), m(o_2)) \in E$ then
9.          $E_w := (o_1, o_2)$
10. Determine the relation between all windows $w \in W$
and compute the sets $E_{modal}, E_{modeless}$ accordingly

4.3. Test Case Generation

The structural view model describes the interaction between windows. Each view-ESG can thereby be used as an input graph for the generation of test cases fulfilling the coverage criteria as introduced in [4]. However, the resulting sequences are abstract and can only be used as descriptions to implement executable test cases for the system under test. Coverage of all relations in the view-ESG represented by its edges can validate if the window controls of the GUI corresponds towards its corresponding requirements.

A more concrete verification is the interpretation of the view-ESG in a Petri net or Janssen net manner [7], [8]. The start window of the view-ESG is therefore annotated with a token. The token represents which window is active.
at a given state. The token is given from one vertex to another vertex whenever a new window is evoked. In case of an undirected edge representing a modeless relation of two interacting windows the token is multiplied to each window that can be evoked and to the parent window itself. An implementation of this idea is a further ongoing work and not realized yet.

Based on this interpretation test cases are created by adapting the test generation approach of covering event pairs and faulty event pairs of the view-ESG as described in [5]. These test cases intend to reveal windowing faults of a GUI by verifying if the correct window modularity is used. In addition, further windowing properties can be defined which have to be tested. These properties concerning especially the structure of a GUI are intended to be user-defined, extensible and flexible. In the following, some of them are listed:

- At any state there should not be opened more than \( n \) windows.
  This is the case if too many modeless windows are used which becomes overwhelming to the user.
- At any state there should not exist more than \( n \) inactive windows.
  That means that the depth of modal windows evoking further modal windows is limited by \( n \) new calls of windows.

Such criteria could be more formally described using temporal logic. Then, it is also possible to apply model checking to verify these properties.

5. Example and Tool Support

The extraction of the structural view model for GUIs can be very cumbersome if done manually. Especially, the behavioral model of very large systems could become complex and have to be analyzed automatically. In this section the approach is conducted for a sample application. The application for the generation of a view-ESG is introduced in short.

![Figure 11. Top-level GUI of WordPad, modal/non-modal windows](image)

5.1. An Example

The approach has been applied to the analysis of the GUIs of different kinds of systems. Figure 11 depicts a small part of the GUI of a MS WordPad-like word processing system. This GUI will usually be active when a text portion is to be loaded from a file, or to be manipulated by cutting, copying, or pasting. The GUI will also be used for saving the text to the current file (or to another one).

At the top level, the GUI has a menu with the items “File”, “Help”, etc. that invoke other components, e.g., the event “File” invokes a pull-down menu with “Open” as an option. Selection of “Open” invokes a modal window. This window has further sub-options, e.g., the option “Select” can select files or invoke sub-directories. The optional events are abbreviated in Figure 11 with capital letters. There are still more window components which will not be explained here further. The window can be closed by selecting either “OK” or “Cancel”. The components described are used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

![Figure 12. Resulting view-ESG (“WordPad”) by identifying window modes and ESGs (“Main”, “Open”, “Help”) for each window](image)

The different window modes are marked by the different triangles. The white triangle evokes a modeless window while the dark triangle represents the call of a modal window. The GUI represented in Figure 11 is transferred to
a view-ESG as shown in Figure 12. The different window types contained are recognized and extracted.

5.2. Tool Support

For executing the developed tool the .NET framework has to be installed. The required files referring to Section 3 can either be given as text or most common as an intermediate format of XML. A graphical representation of the open source project “Jflap” can be also used as input or as an output to be generated. As shown in Figure 13, one needs only to push “Convert-Button” to start the Model Decomposer. Assuming that the corresponding input files have been selected, the tool decomposes each window by its corresponding ESG. The extracted windows are stored in a separated file while the resulting view-ESG is stored in the original file.

![Figure 13. The model decomposer for generating view-ESGs from behavioral model](image)

The end vertex column can only contain a single vertex as an end vertex. When all objects are mapped, the model can be saved. The generated model can be checked in a frame (down side) where each object is represented by its hash coded ID.

6. Related Work

Several approaches have been proposed for GUI testing using different kinds of models. Shehady and Siewiorek [9] used variable finite state machines to generate test cases. White and Almezen’s [10] approach identifies GUI responsibilities that are used to define complete interaction sequences (CIS) consisting of sequences of GUI objects. A CIS is represented using a finite-state model where user actions, such as OPEN FILE and EDIT, label the states and the edges are unlabeled. Thus, the expected behavior in response to an event is implicit and specified elsewhere in contrast to traditional finite-state models that indicate explicitly the system response to an input as an output label on each transition. The entire system is modeled as a collection of CISs. Test sets are generated based on a finite-state machine for each CIS. An advantage of the CIS-based approach lies in its scalability and intuitiveness. Instead of creating a single composite finite-state model where user actions, such as OPEN FILE and EDIT, label the states and the edges are unlabeled. Thus, the expected behavior in response to an event is implicit and specified elsewhere in contrast to traditional finite-state models that indicate explicitly the system response to an input as an output label on each transition. The entire system is modeled as a collection of CISs. Test sets are generated based on a finite-state machine for each CIS. An advantage of the CIS-based approach lies in its scalability and intuitiveness. Instead of creating a single composite finite-state model, multiple CISs, each representing a user responsibility, are created thereby simplifying the task of model construction and test generation.

Different approaches to testing are introduced in [11], [12] that deploy methods of knowledge engineering, particularly planning, to generate test cases, test oracles [13], etc. In [14], a framework is proposed to meet the special demands of GUI testing. It consists of a GUI model and modules that use this model to perform different tasks of testing.

Approaches that consider structural features are using a tree representation for analysis. However, such a representation cannot be used to derive test cases. These approaches also make use of reverse engineering techniques to create GUI models based on the SUT. In [15], GUI models are created by scanning all windows and its widgets beginning from the root window. The relation of different windows is represented as an integration tree whereas the widgets of a windows are represented by another kind of model (event flow graphs). In contrast to our approach, the modality feature of windows is not computed from an existing behavioral model. Event graphs and timed event graphs are also used in other areas of research such as simulation [16] and automatic control [17]. A different approach uses a reverse engineering tool (REGUI2FM) to extract structural and behavioral information from a GUI [18]. As a result, a skeleton of a state machine is generated that is represented in a formal specification language. The hierarchical structure of the windows of the GUI is also gathered from SUT.

We already mentioned [2] and [1] that give good hints for GUI design and ergonomics, but hardly for increasing its testability. Our approach as presented might start there.
7. Conclusion

When testing GUIs the structural features, especially their windowing system, are seldom taken into account. Thus, the testability of such systems is disadvantageous. The benefit of this work is the generation of a structural model representing the windowing of GUIs. The view-ESG is extracted from a given behavioral model by identifying the different window modes. The algorithm as illustrated can be adapted to any event-based behavioral model because of the completely independent structure which is always contained. Furthermore, the method can be applied as a forward and reverse-engineering technique. This paper uses the algorithm as an reverse-engineering to reveal a structural model for a better testability of the view design of GUIs.

The generation of test cases derived from established test criteria is planned for the future. Additionally, a careful analysis will be necessary how well our approach can be applied to large, complex systems.

References


