

Increasing the Robustness of Human-Machine Interactions

Prof. Dr. Fevzi Belli

University of Paderborn (<http://adt.upb.de>),

Faculty of Computer Science, Electrical Engineering and Mathematics

D-33098 Paderborn, Germany

Phone: +49-5251 60 3447; Fax: +49-5251 60 3246

Abstract

*Robust human-machine interfaces enable to deliver acceptable services in spite of occurrence of faulty user interactions. This paper introduces an approach to analysis and testing of man-machine interfaces in a **holistic** way. The generated complementary view of the desirable system behavior enables to obtain a precise and complete description of **undesirable** situations, leading to a systematic analysis and test of the system under consideration. Examples illustrate and validate the approach.*

Key words: Software engineering for accessibility; modeling, analyzing and testing of the robustness of user interfaces; knowledge engineering.

1. Introduction and Terminology

A key factor for increasing the accessibility to information systems (IS) is the robustness of their user interfaces (UI). For an ideal IS, there is no user error, i.e., upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the anticipated situation that completes the sequence of user interactions and thus delivers a desirable outcome.

There are two distinct types of construction work while developing IS, especially its software:

- Design, i.e., modeling and analysis of the IS.
- Design, i.e., modeling and analysis of its UI.

We assume that UI might be constructed separately, as it requires different skills, and maybe different techniques than construction of common IS. The design part of the development job requires a good understanding of user requirements; the implementation part requires familiarity with the technical equipment, i.e. programming platform, language, etc. Analysis requires both: a good understanding of user requirements, and familiarity with the technical equipment. This paper is about modeling and analyzing of

UI. As a good analysis includes not only static aspects, but also entails dynamic ones, we generally speak of *testing* the system under consideration.

Test cases generally require the determination of meaningful test inputs and expected system outputs for these inputs. Accordingly, to generate test cases for a UI, one has to identify the test objects and test objectives. The test objects are the instruments for the input, e.g. screens, windows, icons, menus, pointers, commands, function keys, alphanumeric keys, etc. The objective of a test is to generate the expected system behavior (desirable event) as an output by means of well-defined test input, or inputs. In a broader sense, the test object is the system under test (SUT); the objective of the test is to gain confidence to the SUT. Robust systems possess also a good exception handling mechanism, i.e. they are responsive not in terms of behaving properly in case of correct, legal inputs, but also by behaving good-natured in case of illegal inputs, generating constructive warnings, or tentative correction trials, etc. that help to navigate the user to move in the right direction. In order to validate such robust behavior, one needs systematically generated erroneous inputs which would usually entail injection of undesirable events into the SUT. Such events would usually transform the IS under test into an illegal state, causing even a system crash, if the program does not possess an appropriate exception handling mechanism.

Test inputs of UI usually represent sequences of UI objects activities, or *events*, and/or selections that will operate interactively with the objects (*Event Sequences – ES*). Such an event sequence is *complete* (*complete event sequence – CES*), if it eventually invokes the desirable system responsibility. From Knowledge Engineering point of the view, the testing of GUI represents a typical planning problem that can be solved goal-driven [MEM_]: Given a set of operators, an initial state and a goal state, the planner is expected to produce a sequence of operators that will change the initial state to the goal state. For the UI analysis and test problem described above, this means we have to con-

struct the test sequences in dependency of both the desirable correct events and the undesirable, faulty events. A major problem is the unique distinction between correct and faulty events (*Oracle Problem*). Our approach will exploit the concept of CES to elegantly handle the Oracle Problem.

The present paper summarizes our research work, depicting it by examples lent from real projects, e.g. electronic vending machines which accept electronic and hard money, “emptying” the machine by transfer of the cashed money to a bank account, etc. Section 2 combines system modeling and fault modeling, considering also the test process. An example to illustrate and validate the approach is given in Section 3. Section 4 discusses the approach, considering related work, and concludes the paper summarizing the results.

2. Integrating the System Modeling with Fault Modeling

As already mentioned, this work uses event sequence graphs (ESG) to represent the system behavior and, moreover, the facilities from the user’s point of view to interact with the system. Basically, an *event* is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response, punctuating different stages of the system activity. It is clear that such a representation disregards the detailed internal behavior of the system, which is given by means of its different states and, hence, an ESG is a more abstract representation compared to, for example, a state transition diagram (STD) or finite-state automaton (FSA) [SALO]. Following, the notions used in the approach are formally introduced.

Definition 1. An *event sequence graph* $ESG = (V, E)$ is a directed graph with

$V \neq \emptyset$: a finite set of *vertices (nodes)*,

$E \subseteq V \times V$: a finite set of *arcs (edges)*,

$\Xi, \Gamma \subseteq V$: finite sets of distinguished vertices $\xi \in \Xi$ and $\gamma \in \Gamma$, called *entry nodes* and *exit nodes*, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \dots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \dots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \dots, k-1$ and $v \neq \xi, \gamma$.

$\Xi(ESG)$, $\Gamma(ESG)$ represents the entry nodes and exit nodes of a given ESG, respectively. To mark the

entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex $[\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex $] \notin V$.

The semantics of an ESG is as follows. Any $v \in V$ represents an event. For two events $v, v' \in V$, the event v' must be enabled after the execution of v if and only if $(v, v') \in E$.

The operations on identifiable components of the UI are controlled and/or perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of V and lead interactively to a succession of user inputs and expected desirable system outputs.

Definition 2. Let V, E be defined as in Definition 1.

Then any sequence of vertices $\langle v_0, \dots, v_k \rangle$ is called an *event sequence (ES)* if $(v_i, v_{i+1}) \in E$, for $i = 0, \dots, k-1$.

Note that the pseudo vertices $[,]$ are not included in the ESs. An $ES = \langle v_i, v_k \rangle$ of length 2 (an arc, or edge of the ESG) is called an *event pair (EP)*. Accordingly an *event triple (ET)*, *event quadruple (EQ)*, etc. can be defined.

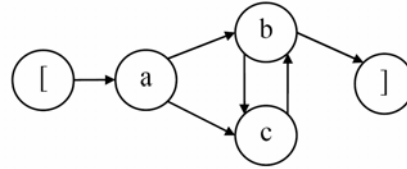


Fig. 1: An ESG with a as entry and b as exit and pseudo vertices $[,]$

Example 1. For the ESG given in Figure 1: $V = \{a, b, c\}$, $\Xi = \{a\}$, $\Gamma = \{b\}$, and $E = \{(a, c), (a, b), (b, c), (c, b)\}$. Note that arcs from pseudo vertex $[$ and to pseudo vertex $]$ are not included in E .

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial vertex and end vertex of an ES, e.g., for $ES = \langle v_0, \dots, v_k \rangle$, initial vertex and end vertex are $\alpha(ES) = v_0$, $\omega(ES) = v_k$, respectively. For a vertex $v \in V$, $N^+(v)$ denotes the set of all *successors* of v , and $N^-(v)$ denotes the set of all *predecessors* of v .

sors of v . Note that $N^-(v)$ is empty for an entry $\xi \in \Xi$, and $N^+(v)$ is empty for an exit $\gamma \in \Gamma$.

Finally, the function $l(\text{length})$ of an ES determines the number of its vertices. In particular, if $l(ES) = 1$ then $ES = \langle v_i \rangle$ is an ES of length 1.

Note that the pseudo vertices l and r are not considered in generating any ESs. Neither are they considered to determine the initial vertex, end vertex, and length of the ESs.

Example 2. For the ESG given in Figure 1, $bcbc$ is an ES of length 4 with the initial vertex b and end vertex c .

Definition 3. An ES is a complete ES (or, it is called a *complete event sequence, CES*), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\omega(ES) = \gamma \in \Gamma$ is an exit.

Example 3. acb is a CES of the ESG given in Figure 1.

CESs represent *walks* from the entry of the ESG to its exit realized by the form

(initial) user inputs \rightarrow (interim) system responses \rightarrow ... \rightarrow (final) system response.

Note that a CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs and a final system response.

2.1 Fault Model

The approach introduced in this paper assumes that there is no user error, i.e., upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the anticipated desirable situation. Due to this requirement, a complementary view is necessary to consider potential user errors in the modeling of the system.

Definition 4. For an $ESG = (V, E)$, its *completion* is defined as $\overline{ESG} = (V, \overline{E})$ with $\overline{E} = V \times V$.

Definition 5. The inverse (or complementary) ESG is then defined as $\overline{\overline{ESG}} = (V, \overline{\overline{E}})$ with $\overline{\overline{E}} = \overline{E} \setminus E$.

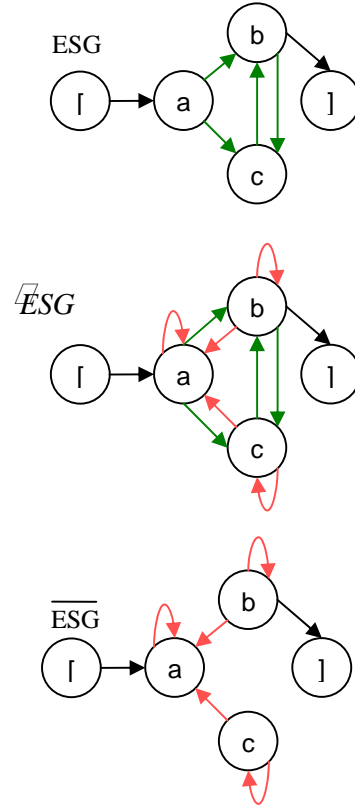


Fig. 2: ESG of Figure 1, its completion \overline{ESG} and inversion $\overline{\overline{ESG}}$ with $\overline{\overline{ESG}} = \overline{ESG} \setminus ESG$

Figure 2 illustrates \overline{ESG} , which can systematically be constructed in three steps:

- Add arcs in the opposite direction wherever only one-way arcs exist.
- Add self-loops to vertices wherever none exist.
- Add two-way arcs between vertices wherever no arcs connect them. Note that they are drawn bi-directional.

$\overline{\overline{ESG}}$ (the inversion of the ESG) consists of arcs that will be added to the ESG to construct the \overline{ESG} (completion of the ESG).

Definition 6. Any EP of the $\overline{\overline{ESG}}$ is a *faulty event pair (FEP)* for ESG.

Example 4. ca of the given $\overline{\overline{ESG}}$ in Figure 2 is a FEP.

Algorithm 1: Test Process.

```

n:= number of the functional units (modules) of the
system that fulfill a well-defined task
length:= required length of the test sequences

FOR function 1 TO n DO
  BEGIN
    Generate the corresponding ESG and  $\overline{ESG}$ 
    FOR k:=2 TO length DO
      BEGIN
        Cover all ESs of length k by means of CESs
      END
      Cover all FEPs by means of FCESs
    END
  Apply the test set to the SUT
  Observe the system output to determine whether a
correct
system response or a faulty event occurs

```

Definition 7. Let $ES = \langle v_0, \dots, v_k \rangle$ be an event sequence of length $k+1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding \overline{ESG} . The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0, \dots, v_k, v_m \rangle$.

Example 5. For the ESG given in Figure 2, bca is an FES of length 3.

Definition 8. An FES is *complete* (or, it is called a *faulty complete event sequence*, *FCES*) if $\alpha(FES) = \xi \in \Xi$ is an entry. The ES as part of a FCES is called a *starter*.

Note that Definition 8 explicitly points out that a FCES does not finish at an exit, unlike a CES that must finish at an exit.

Example 6. For the ESG given in Figure 2, the FEP ca of the \overline{ESG} can be completed to the FCES $acbca$ by using the ES $acbc$ as a starter. Note that the l is not included in the FCES as it is a pseudo vertex.

The starter $acbc$ in Example 6 is arbitrarily chosen, and hence the variation in length of an FCES is always attributable to starters prior to this special FEP under consideration. The result is then FCESs of various lengths. Thus, the “length” in the test process primarily relates to the CESs.

2.2 Test Process

As already mentioned in Section 1, a major problem of testing is the determination of correct and faulty

situations upon inputs (oracle problem]). The approach introduced in this paper uses event sequences, more precisely CES and FCES, as test inputs. If the input is a CES, the SUT is supposed to accept this input and produce a desirable behavior that is well-defined prior to testing; thus the test succeeds. Accordingly, if a FCES is used as a test input, the SUT is supposed to refuse this input and produce a warning. The latter case represents an exception that must be properly handled by the system in order to avoid undesirable failures.

Definition 9. A *test case* is an ordered pair of an input and expected output of the SUT. Any number of test cases can be extended to a *test set*.

The test process is summarized by the given Algorithm 1. The approach assumes that the system is modularly structured, e.g., having n units. Each of these units fulfills a task that can be modeled by an ESG.

The coverage-oriented test process of the approach leads to test cases which exercise the specified functions of the implemented SUT with the goal to cover these functions. This coverage must be, of course, economical in terms of the number of test cases. Therefore, a *stopping rule* of the test case generation is needed. This is given in the Algorithm 1 by the coverage of the event sequences of increasing length (*edge coverage* for length=2), which is determined by analyzing the model.

3. An Example: Vending Machine

For the sake of clear and ease of understanding, we reduce here the full capabilities of the tested modern vending machines considerably. Nowadays, such machines can accept money or credit cards for issuing train or flight tickets, carrying out transfers to and from a control system according to a communication protocol, considering the due security procedures, etc. Following we simplify different components of a vending machine that accept 1 EUR to output a cup of hot chocolate, and 2 EUR for coffee. Further description of the system is included in the legend of the Fig. 3.

Fig. 4 displays the ESG which has been completed, i.e., the entire set of the legal and illegal connections between all nodes are visible. We can exploit now Fig. 4 for fault analysis; we skip here, however, the complete analysis, and report only some faults which seem interesting for us, and in some sense surprising in the Table 1.

Discussion of the Example

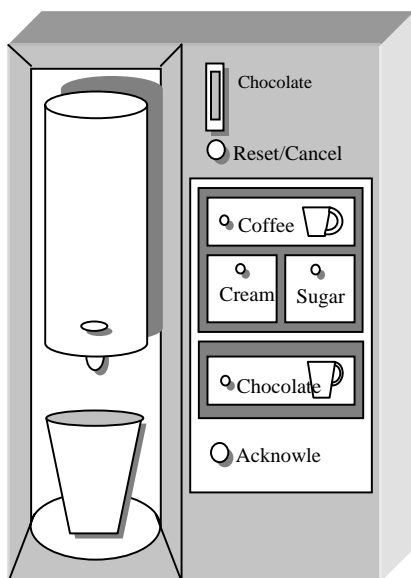
The solid lines in Figure 4 depict the legal sequences of events; the dashed ones are illegal

connections, realizing faulty sequences of events of length 2.

The legal connections visualize the system as it is designed to behave. The test we performed made use of both legal sequences and illegal sequences, i.e., CES and FCES. Those sequences are input as test cases to the system. Some of them revealed faults that are summarized in Table 1.

The specialty of the approach is that the fault detection is reproducible in that the numbers on the arcs of Figure 4 exactly describes how to transform the system into a faulty state, step by step. Thus, the dashed lines clearly indicate the structural flaws of the system that hinder robustness that would tolerate faulty interactions of the user.

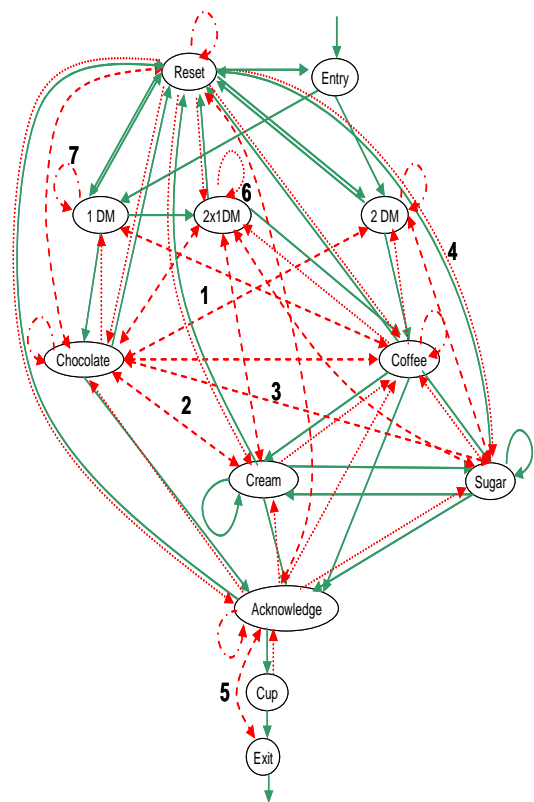
To enable the robustness, a redesign of the system is to take that insufficient behavior into account.



Legend:

- Coffee: 2 EUR.
- Chocolate: 1 EUR.
- The machine accepts only coins 1 EUR and 2 EUR.
- All keys are touch keys; upon activation the control light goes on.
- Acknowledge key starts the vending machine immediately and cannot be interrupted.
- The Reset key brings the machine to the start state and returns the entered coin.

Fig. 3: Vending Machine



Legend:

- > : Interaction Pairs
- - -> : Faulty Interaction Pairs
- X: Fault nb. described in Table 2

Fig. 4: Completed ESG of the Vending Machine (Fig. 3)

4. Related Work and Conclusion

State-based and event-based methods have been used for many decades for specification and testing of software and system behavior, e.g. for Conformance Testing [PARN, BOCH]. Recently, L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [WHIT]. Our work clarifies and extends this approach by taking not only desirable behavior of the software into account, but also undesirable situations. This could be seen as the most important contribution of our present work, i.e., testing GUIs not only through exercising them by means of test cases which show

that GUI is working properly under regular circumstances, but exercising also all potentially illegal events to verify that the GUI behaves satisfactory also in exceptional situations. Thus, we have now a *holistic* view concerning the complete behavior of the system we want to test. Moreover, having an exact terminology and appropriate formal methods, we can now precisely scale the test process, justifying the cumulating costs that must be in compliance with the test budget.

A different approach for GUI testing has been recently published by A. Memon et al. [MEM_]. The authors deploy methods of Knowledge Engineering, to generate test cases, test oracles, etc. to handle also the Test Termination Problem. Both approaches, i.e. of A. Memon et al., and L. White, use some heuristic methods to cope with the state explosion problem. We also introduced in the present paper methods for test case selection; moreover we handled test coverage aspects for termination of GUI testing, based on theoretical knowledge that is well-known in Conformance Testing and validated in the practice of protocol validation for decades. The advantage of our approach stems from its simplicity that causes a broad acceptance in the practice. Furthermore, the results of our work enable efficient algorithms to generate and select test cases in sense of a meaningful criterion, i.e., edge coverage.

The introduced holistic approach bases on [BeGr], unifies the modeling of both the desirable and undesirable features of the system to be developed and enables the adoption of the concept “Design for Testability” in IS design; this concept was initially introduced in the seventies [WILL, GOOD] for hardware. We hope that further research will enable the adoption of our approach in more recent and powerful modeling tools as to State Charts [HORR], UML [KIM], etc.

Last, but not least, we hope that this paper could demonstrate the necessity and feasibility of designing robust interfaces for human-machine interactions. In many cases, as demonstrated in the example here, the faulty actions of the user can be transformed to correct ones. In such cases the machine control has to anticipate user’s intension and carry out the operation initiated by the user.

5. Acknowledgments

My thanks are due to Dr. Christof Budnik, my former assistant, for working out the example used in this paper.

References

[BeGr] F. Belli, K.-E. Grosspietsch, “Specification of Fault-

Tolerant System Issues by Predicate/Transition Nets and Regular Expressions – Approach and Case Study”, *IEEE Trans. On Softw. Eng.* 17/6, pp. 513-526, 1991

- [GOOD] J.B. Goodenough, “Exception Handling – Issues and a Proposed Notation”, *Comm. ACM* 18/12, pp. 683-696 (1975)
- [HAML] D. Hamlet, “Foundation of Software Testing: Dependability Theory”, *Proc. Of ISSA '96*, pp. 84-91, 1994
- [HORR] I. Horrocks, “*Constructing the User Interface with Statecharts*”, Addison-Wesley, MA, 1999
- [KIM] Y. G. Kim, H. S. Hong, D.H Bae and S.D. Cha, “Test Cases Generation from UML State Diagrams”, *IEE Proc.-Softw.* Vol. 146, pp. 187-192, Aug. 1999
- [MEM_] A. M. Memon, M. E. Pollack and M. L. Soffa, “Hierarchical GUI Test Case Generation Using Automated Planning”, *IEEE Trans. Softw. Eng.* 27/2, pp. 144-155, 2001
- [PARN] D.L. Parnas, “On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System”, *Proc. 24th ACM Nat’l. Conf.*, pp. 379-385, 1969
- [SHNE] B. Shneiderman, “*Designing the User Interface*”, Addison Wesley Longman, 1998
- [WHIT] L. White and H. Almezen, “Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences”, in *Proc. Int. Symposium on Softw. Reliability Engineering ISSRE 2000*, IEEE Comp. Press, pp. 110-119, 2000
- [WILL] T. W. Williams and K. P. Parker, “Design for Testability - A Survey”, *IEEE Trans. Comp.* 31, pp. 2-15, 1982

Table 1: Faults detected in vending machine (Fig. 3)

Fault 1:	The construction forgot the case that chocolate should be obtained also when 2 EUR has been inserted. The user should receive then either two consecutive cups of chocolate, or 1 EUR returned, depending on an additional selection that is missing in the design here.
Faults 2, 3:	The machine should warn the user in case he/she selects clumsily chocolate with cream (FEP on arc 2) and/or with sugar (FEP on arc 3), without knowing that chocolate includes already sugar and cream. The result is often ruined taste and flavor.
Fault 4:	The selection keys should be kept locked before coin has been inserted. A display should inform the user appropriately.
Fault 5:	An additional sensor should ensure that a cup has been inserted before filling process starts. The sensor should also stop the filling if the cup will be removed before the process concludes.
Faults 6, 7:	A lock should exclude the multiple insertions of coins 1 EUR and 2 EUR, except twice EUR 1 which can be alternatively input instead of one single EUR 2.