Test minimization for human-computer interaction

Fevzi Belli · Christof J. Budnik

Abstract This paper introduces a model-based approach for minimization of test sets to validate the interaction of human-computer systems. The novelty of the approach is twofold: (i) Test cases generated and selected holistically cover both the behavioral model and the complementary, fault model of the system under test (SUT). (ii) Methods known from state-based conformance testing and graph theory are extended to construct efficient, heuristic search-based algorithms for minimizing the test sets that are constructed in step (i), considering also structural features. Experience shows that the approach can help to considerably save test costs, up to 60%.

Keywords Fault diagnosis · Model-based reasoning and testing · Test generation/test selection · Chinese postman problem · Event sequence graph · Test coverage · Minimal spanning set

1 Introduction

Testing is the traditional validation method in the software industry. This paper introduces a specification-oriented, model-based testing approach; i.e., the underlying model represents the system behavior interacting with the user’s actions. The system’s behavior and user’s actions are viewed here as events, more precisely, as desirable events if they are in accordance with the user expectations. Moreover, the approach includes modeling of the faults as undesirable events.

as, mathematically spoken, a complementary view of the behavioral model. Once the model is established, it “guides” the test process to generate and select test cases, which form sets of test cases (also called test suites). The selection is usually ruled by an adequacy criterion, which provides a measure of how effective a given set of test cases is in terms of its potential to reveal faults [1]. Most of the existing adequacy criteria are coverage-oriented. The ratio of the portion of the specification or code that is covered by the given test set in relation to the uncovered portion can then be used as a decisive factor in determining the point in time at which to stop testing (test termination).

From Knowledge Engineering point of view, the testing represents a typical planning problem that can be solved goal-driven: 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 test problem described above, this means we have to construct 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).

Based on [2], this paper introduces event sequence graphs (ESG) for representing both the behavioral model and the fault model of interactive systems and their coverage-oriented testing. To solve the oracle problem effectively, the involvement of the user is limited to solely judge final events of the ESGs. The number of the test cases to cover both models primarily determines the test costs. Therefore, test sets are constructed and minimized by heuristic search-based algorithms exploiting of the structural feature of the system under test (SUT) (minimal spanning set for coverage testing). A scalability of the test process is given by the length of events to be covered which can be stepwise increased.

F. Belli · C. J. Budnik (✉)
University of Paderborn, EIM-E, Warburger Str. 100, 33098 Paderborn, Germany
e-mail: budnik@adt.upb.de

F. Belli
e-mail: belll@upb.de
Compared to [2], where it and was first used for the study of user interactions, this work extends and refines it in following respects: (a) The present paper introduces a solid formal background for modeling aspects by means of ESG notion. (b) Based on this background, algorithms are introduced and analyzed for test generation and optimization. (c) The fault model is extended in order to handle a broad variety of malfunctions, qualitatively and quantitatively. (d) A case study validates the approach by means of a lucid, nevertheless non-trivial application and analyzes its deployment, identifying and varying characteristic factors for a justifiable deployment. (e) Tools that are necessary for an effective deployment are introduced and discussed. (f) Lessons learned from intensive and extensive application of the approach to industrial projects are summarized.

The next section summarizes the related work before Section 3 introduces the fault model and the test process. The minimization of the test suite is discussed in Section 4. Section 6 summarizes the results of the case study to validate the approach, using appropriate tools (as described in Section 5) we developed to support the practical work. Section 7 discusses the results of the case study reflecting the lessons learned. Section 8 concludes the paper and sketches the research work planned.

2 Related work

Methods based on state-based and event-based techniques, especially on finite-state automata (FSA), have been used for almost four decades for the specification and testing of system behavior [3], as well as for conformance and software testing [4, 5]. Also, the modeling and testing of interactive systems with a state-based model has a long tradition [6, 7]. These approaches analyze the SUT and model the user requirements to achieve sequences of user interaction (UI), which then are deployed as test cases.

The approach in this paper introduces a simplified state-based, graphical model to represent UIs to consider not only the desirable situations, but also the undesirable ones. This strategy is quite different from the combinatorial ones, e.g., pairwise testing, which requires that for each pair of input parameters of a system, every combination of these parameters' valid values must be covered by at least one test case. It is, in most practical cases, not feasible [8] to test UIs.

A similar fault model as in [2] is used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using mutation operations [9]. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g., system testing, integration testing, etc. [10]. Such operations have also been independently proposed by other authors, e.g., "state control faults" for fault modeling in [11], or for "transition-pair coverage criterion" and "complete sequence criterion" in [5]. However, the latter two notions have already been introduced, considerably earlier in [2].

Techniques well-known from artificial intelligence (AI) will traditionally and successfully used for fault diagnosis, especially model-based diagnosis [12–14], also to generate software tests [15, 16]. Soft computing methods, e.g., genetic algorithms, are widely used to introduce adequacy criteria and to solve the oracle problem [17]. All of those approaches deploy methods based on some heuristic algorithms to cope with the state explosion problem while constructing and selecting test suites.

This paper also presents a model-based approach to test case generation and selection. Moreover, it addresses test coverage aspects for test termination, based on [2], which introduced the notion of "minimal spanning set of complete test sequences", similar to "spanning set", that was later discussed also in [18]. The present paper considers existing approaches to optimize the round trips, i.e., the Chinese Postman Problem [4], and attempts to determine algorithms of less complexity for the spanning of walks, rather than tours, related to [19].

3 Event sequence graphs, fault model and test process

Any software-based system can be viewed as interacting with its environment through stimuli-response pairs. In this context, the environment could be one or more human users, a set of service seekers, or any combination thereof, and we use the terms "user" and "environment" interchangeably. An Event Sequence Graph (ESG) is supposed to model a subset of the interactions between a system and its user. The complete set of interactions is captured in terms of a set of ESGs, where each ESG represents a possibly infinite set of event sequences. An event, an externally observable phenomenon, can be a user stimulus or a system response, punctuating different stages of the system activity.

Definition 1. An event sequence graph $ESG = (V, E, \Xi, \Gamma)$ 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 $\forall \xi \in \Xi \text{ there is at least}$

one sequence of vertices $(\xi, v_0, \ldots, v_k)$ from each $\xi \in \Xi$ to $v_k = \gamma$ and

one sequence of vertices $(v_0, \ldots, v_k, \gamma)$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k - 1$ and $v \neq \xi, \gamma$. 
Definition 2. Let \( V, E \) be defined as in Definition 1. Then any sequence of vertices \((v_0, \ldots, v_k)\) is called an event sequence (ES) if \((v_i, v_{i+1}) \in E, \) for \( i = 0, \ldots, k-1.\)

Note that the pseudo vertices \( i, j \) are not included in the ESs. An ES \( (v_i, v_k) \) of length 2 is called an event pair (EP). Accordingly an event triple (ET), event quadruple (EQ), etc. can be defined.

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

Furthermore, \( \alpha(\text{initial}) \) and \( \alpha(\text{end}) \) are functions to determine the initial vertex and end vertex of an ES, e.g., for ES \( (v_0, \ldots, v_k), \) initial vertex and end vertex are \( \alpha(ES) = v_0, \alpha(ES) = v_k, \) 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 \( v. \) 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 \( \text{length} \) of an ES determines the number of its vertices. In particular, if \( \text{length}(ES) = 1 \) then \( \Xi(ES) = s(v) \) is an ES of length 1.

Note that the pseudo vertices \( i \) and \( j \) 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 Fig. 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 \( \alpha(ES) = \gamma \in \Gamma \) is an exit.

Example 3. \( abc \) is a CES of the ESG given in Fig. 1.

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

\[
(\text{initial user inputs} \rightarrow (\text{interim} \text{ system responses}) \rightarrow \cdots \rightarrow (\text{final system response}).
\]

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.

Definition 4. Given an ESG, say \( ESG_1 = (V_1, E_1) \), a vertex \( v \in V_1 \), and an ESG, say \( ESG_2 = (V_2, E_2). \) Then replacing \( v \) by \( ESG_2 \) produces a refinement of \( ESG_1, \) say \( ESG_3 = (V_3, E_3) \) with \( V_3 = V_1 \cup V_2 \setminus \{v\}, \) and \( E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \cup E_{replaced} \) (\( \setminus \) : set difference operation),

\[
E_{pre} = N^- (v) \times \Xi(ESG_2) \quad (\text{connections of the predecessors of } v \text{ with the entry nodes of } ESG_2),
\]

\[
E_{post} = \Gamma(ESG_2) \times N^+(v) \quad (\text{connections of exit nodes of } ESG_2 \text{ with the successors of } v), \) and
\]

\[
E_{replaced} = \{(v_i, \gamma) : v \in N^-(v) \text{ and } v \in N^+(v) \} \quad (\text{replaced arcs of } ESG_1).
\]

As Fig. 2 illustrates, every predecessor of vertex \( v \) of the ESG of higher level abstraction points to the entries of the refined ESG. In analogy, every exit of the refined ESG points to the successors of \( v. \) The refinement of \( v \) in its context within the original ESG of higher level abstraction contains no pseudo vertices \( i \) and \( j \) because they are only needed for the identification of entries and exits of the ESG of a refined vertex.

Example 4. In Fig. 3 the refinement of the vertex \( a \) of \( ESG_1 \) is given as \( ESG_2. \) \( ESG_3 \) is the resulting refinement of \( ESG_1. \) Note that the pseudo vertices \( i \) and \( j \) of \( ESG_2 \) are not included in \( ESG_3. \)

More precisely, \( ESG_1 \) is given as \( V_1 = \{a, b, c\}, E_1 = \{(a, b), (a, c), (b, c), \{c, b\}\}. \) In the refinement, i.e., \( ESG_2 \) of \( a, \) the predecessors and successors are \( N^-(v) = \{\}, N^+(v) = \{b, c\} \) and the refinement of \( ESG_2 \) is given by \( V_3 = \{a, a_2, a_3\}, E_2 = \{(a, a_2), (a_1, a), (a_2, a_3), (a_3, a_2)\}, \Xi(ESG_2) = \{a_1\} \) and \( \gamma(ESG_2) = \{a_2, a_3\}. \) The resultant \( ESG_3 \) is represented by

\[
V_3 = V_1 \cup V_2 \setminus \{v\} = \{a, b, c\} \cup \{a_1, a_2, a_3\} \setminus \{a\} = \{b, c, a_1, a_2, a_3\}.
\]
Fig. 2 Refinement of a vertex $v$ and its embedding in the refined ESG

ESG$_1$ ("mother" ESG of higher level abstraction)

ESG$_2$ (refinement of the vertex $v$ of ESG$_1$)

ESG$_3$ (Resultant, refined "mother" ESG$_1$; the refinement of its vertex $v$ is embedded in its context)

Fig. 3 A Refinement of the vertex $a$ of the ESG given in Fig. 1

$$E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{replaced}$$

$$= \{(a, b), (a, c), (b, c), (c, b)\} \cup \{(a_1, a_2), (a_1, a_3), (a_2, a_3), (a_3, a_2), (a_2, b), (a_2, c), (a_3, b), (a_3, c)\}$$

3.1 Fault model and test terminology

System malfunctions manifest in form of failures, thus affecting the ability of the system to perform its functions. A failure can be realized, or triggered, by a fault as an incorrect, thus undesirable, event during the execution of the system. A fault can eventually be traced back to a human action, or inaction, leading to an incorrect and, thus, an undesirable result. Though there is a cause-effect relationship [20] between them, the terms "error" and "fault" are often synonymously used as the cause of a "failure". The purpose of any test effort is to force an error/fault (lying at the root of the above chain) to show up as a failure, i.e., as a situation that cannot be hidden from the environment, especially from the user of the system.

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 (see also [21, 22]).

**Definition 5.** For an ESG $=(V, E)$, its completion is defined as $\overline{ESG} = (V, \overline{E})$ with $\overline{E} = V \times V$.

**Definition 6.** The inverse (or complementary) ESG is then defined as $\overline{ESG} = (V, \overline{E})$ with $\overline{E} = \overline{E} \setminus E$.

Figure 4 illustrates $\overline{ESG}$, which can systematically be constructed in three steps:

- Add arcs in the opposite direction wherever only one-way arcs exist.

Fig. 4 ESG of Fig. 1, its completion $\overline{ESG}$ and inversion $\overline{ESG}$ with $\overline{ESG} = ESG \setminus ESG$
- 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.

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

**Definition 7.** Any EP of the \( ESG \) is a faulty event pair (FEP) for \( ESG \).

**Example 5.** \( ca \) of the given \( ESG \) in Fig. 4 is a FEP.

**Definition 8.** Let \( ES = (v_1, \ldots, v_k) \) be an event sequence of length \( k + 1 \) of an \( ESG \) and \( FEP = (v_k, v_m) \) a faulty event pair of the corresponding \( ESG \). The concatenation of the ES and FEP then forms a faulty event sequence \( FES = (v_1, \ldots, v_k, v_m) \).

**Example 6.** For the \( ESG \) given in Fig. 4, \( bca \) is an \( FES \) of length 3.

**Definition 9.** An \( FES \) is complete (or, it is called a faulty complete event sequence, FCES) if \( \alpha(FES) = \Xi \) is an entry. The ES as part of a FCES is called a starter.

Note that Definition 9 explicitly points out that a FCES does not finish at an exit, unlike a CES that must finish at an exit.

**Example 7.** For the \( ESG \) given in Fig. 4, the FEP \( ca \) of the \( ESG \) can be completed to the FCES \( acba \) by using the ES \( acba \) as a starter. Note that the \( f \) is not included in the FCES as it is a pseudo vertex.

The starter \( acba \) in Example 7 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.

### 3.2 Test process

**Definition 10.** A test case is an ordered pair of an input and expected output of the SUT. Any number of test cases can be compounded to a test set (or, a test suite).

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 successfully proceed it and thus, to succeed the test and to trigger a desirable event. Accordingly, if a FCES is used as a test input, a failure is expected to occur which is an undesirable event and thus, to fail the test. Algorithm 1 below sketches the test process.

**Algorithm 1. Test Process**

- n: number of the functional units (modules)
- length: length of the test sequences
- FOR function \( 1 \) TO \( n \) DO
  - Generate appropriate \( ESG \) and \( ESG \)
  - FOR \( k = 2 \) TO length \( \text{DO} \)
    - Cover all ESs of length \( k \) by means of CESs subject to minimizing the number and total length of the CESs // see Section 4.1
    - Cover all FEPs of by means of FCESs subject to minimizing the total length of the FCESs // see Section 4.2
  - Apply the test set to the SUT.
- Observe the system output to determine whether the system response is in compliance with the expectation.

Note that the functional units \( n \) of a system in Algorithm 1 is given by the corresponding \( ESG \)s and their refinements (see Definition 5) that fulfill a well-defined task. To determine the point in time in which to stop testing, the approach converts this problem into the coverage of the ES and CES of length \( k \) of the \( ESG \) whereby \( k \) is a decisive cost factor. Thus, depending on \( k \), the test costs are to be scalable and stepwise increased by the tester in accordance with the quality goal and test budget.

### 4 Minimizing the test sets

The union of the sets of CESs of minimal total length to cover the \( ESs \) of a required length is called Minimal Spanning Set of Complete Event Sequences (MSCES). If a CES contains all ESs at least once, it is called an entire walk. A legal entire walk is minimal if its length cannot be reduced. A minimal legal walk is ideal if it contains all ESs exactly once. Legal walks can easily be generated for a given \( ESG \) as CESs, respectively. It is not, however, always feasible to construct an entire walk or an ideal walk. Using some results of the graph theory [19], MSCESs can be constructed as the next section illustrates.

#### 4.1 An algorithm to determine minimal spanning set of complete event sequences (MSCES)

As mentioned in Section 3, a CES represents a legal walk, traversing the \( ESG \) from its entry to the exit. Given an \( ESG \) \( e \), a complete legal walk contains each EP in \( e \) at least once. A complete legal walk is minimal if its length cannot be reduced without changing it to an incomplete legal walk. A minimal
legal walk is considered ideal when it contains every EP exactly once. Legal walks can be generated easily for a given ESG as CESs. It is not, however, always feasible to construct a complete or an ideal walk. Using results from graph theory [19], MSCESs can be constructed as follows:

- Check whether an ideal walk exists.
- If not, check whether a complete walk exists and, if so, construct a minimal one.
- If there is no complete walk, construct a set of walks such that (a) sum of the lengths of all walks is minimal, and (b) all EPs are covered.

The MSCES problem introduced here is expected to have a lower degree of complexity than the Chinese Postman Problem as the edges of the ESG are not weighted, i.e., the adjacent nodes are equidistant. In the following we summarize results relevant to the calculation of test costs that make the test process scalable. An algorithm described in [23] to solve the CPP determines a minimal tour that covers the edges of a given strongly connected graph. Transformation of an ESG into a strongly connected graph is illustrated in Fig. 6. Addition of a backward edge, indicated as a dashed arrow from the exit to the entry, transforms the ESG in Fig. 5(a) to a strongly connected graph in Fig. 5(b).

The labels of the vertices in Fig. 5(b) indicate the balance of these vertices as the difference between the number of incoming edges and the number of outgoing edges. These balance values determine the number of additional edges that will be identified by searching all-shortest-paths and solving the optimization problem. The problem can then be transformed into the construction of an Euler tour for this graph [19]. This tour may have multiple occurrences of the backward edge indicating the number of walks. For the ESG in Fig. 5(b), based on Fig. 5(a), the minimal set of the legal walks covering the EPs are given in Example 8. Note that no complete walks exist. Therefore, an ideal walk cannot be constructed.

Example 8. MSCES = \{abcdb, ac\}

Algorithm 2. Generation of MSCES

Input: ESG = (V, E); ε = \{, γ = \}
Output: MSCES
add_arc (ESG, (γ, ε));

 bags A, B, M = \{\}; set MSCES = \{\}; //empty bags & set
FOR all nodes v \in V DO
 IF (diff(v) > 0) THEN FOR
 i := 1 TO diff(v) DO A = A \cup [v];
 IF (diff(v) < 0) THEN FOR
 i := 1 TO diff(v) DO B = B \cup [v];
 m = |A| = |B|; //cardinality
 D[1.. m][1.. m]; //distance matrix D
FOR all nodes v \in A DO
 compute_shortest_paths (v, B, D);
 M = solveAssignmentProblem (D);
FOR all (i, j) \in M DO
 Path = getShortestPath(i, j);
 FOR all arcs e \in Path DO
 add_arc (ESG, e);
 EulerTourList = computeEulerTour (ESG);
 start = 1;
 FOR i = 2 TO length (EulerTourList) - 1
 IF (getElement (EulerTourList, i) = γ) THEN
 MSCES = MSCES U getPartialList (EulerTourList, start, i);
 start = i + 1;
 RETURN MSCES;

Theorem 1. MSCES can be constructed in time O(|V|^3).

Proof (sketched; see also [23]): The shortest paths from one node to all other ones can be determined by a depth-first-search in O(|E| \* |V|), as ESG under consideration is an unweighted graph. Furthermore, because of |E| \gg |V| + 1, the complexity can be approximated to O(|E|). The Hungarian Algorithm that solves the assignment problem has the complexity O(|V|^3) and the algorithm next to determine the
Euler-Tour has the complexity $O(|E| + |V|)$. Thus, the total complexity is determined by $O(|V|^3)$.

In Algorithm 2, $\epsilon$ denotes the entry of the ESG and $\gamma$ its exit, the symbol $\Rightarrow$ the assignment command. Given an event $v \in V$, $\text{diff}(v)$ denotes the number of predecessor events of $v$ minus the number of successor events, which enables the construction of the bags (or multisets) A, B in the FOR-loop. We introduce the notation $\emptyset$ for bags and $\cup$ bag union. They can be defined informally as follows. For instance, if $\text{diff}(v) = 3$ in the first iteration step, assuming that A is initially empty, the bag A will consist of three instances of v, i.e., $A = \ll brv, v, v \rr$, after the assignment there. Note that $\ll v, v, v \rr \neq v$, because the two entities on either side of the inequality sign $\neq$ are of different types; on the LHS is a bag (with three instances of $v$), whereas on the RHS is a singleton set with one element $v$. Turning to $\cup$, note that $\ll v, v, v \rr \cup \ll v \rr = \ll v, v, v \rr$.

4.2 Determination of minimal spanning set for the coverage of faulty complete event sequences (MSFCES)

The union of the sets of FCESs of the minimal total length to cover the FESs of a required length is called Minimal Spanning Set of Faulty Complete Event Sequences (MSFCES). In comparison to the interpretation of the CESs as legal walks, illegal walks are realized by FCESs that never reach the exit. An illegal walk is minimal if its starter cannot be shortened. Assuming that an ESG has $n$ nodes and $d$ arcs as EPS to generate the CESs, then at most $n := n^2 - d$ FCESs of minimal length, i.e., of length 2, are available. Accordingly, the maximal length of an FCES can be $n$; those are subsequences of CESs without their last event that will be replaced by an FEP. Therefore, the number of FCESs is precisely determined by the number of FEPs. FEPs that represent FCES are of constant length 2; thus, they also cannot be shortened. It remains to be noticed that only the starters of the remaining FEPs can be minimized, e.g., using the algorithm given in [24].

Example 9. The minimal set of the illegal walks (MSFCES) for the ESG in Fig. 6: aa, ad, abb, aba, aca, ace, acd, abdb, abdd, abda.

4.3 Exploiting the structural features of SUT for further reduction of test effort

The approach has been applied to the testing and analysis of the GUIs of different kind of systems, leading to a considerable amount of practical experience [2]. A great deal of test effort could be saved considering the structural features of the SUT. Thus, there is further potential for the reduction of the cost of the test process.

Analysis of the structure of the GUIs delivers the following features:

- Windows of commercial systems are nowadays mostly hierarchically structured, i.e., the root window invokes children windows that can invoke further (grand) children, etc.
- Some children 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 (modal windows).

Figure 7 represents these window types as a “family tree.” In this tree, a unidirectional edge indicates a modal parent-child relationship. A bidirectional edge indicates a modeless one.

Because modal windows must be closed before any other window can be invoked, it is not necessary to consider the FESs of the parent and children. This is true only for the FCESs and MSFCES as test inputs considering the structure information might impact the structure of the ESG, but not the number of the CESs and MSCEs as test inputs.

<table>
<thead>
<tr>
<th>root</th>
</tr>
</thead>
<tbody>
<tr>
<td>modal</td>
</tr>
<tr>
<td>modeless</td>
</tr>
<tr>
<td>children</td>
</tr>
<tr>
<td>children</td>
</tr>
</tbody>
</table>

Fig. 7 Modal windows vs. modeless windows and an example of an modal opened window
Fig. 8 Modal windows vs. modeless windows and an example of an modal opened window.

Thus, similar to the strong-connectedness and symmetrical features [7], the modality feature is extremely important for testing since it avoids unnecessary test efforts. Figure 8 represents the modified ESG, which separates the event “Modal Form” takes the modality into account that avoids unnecessary FEPs.

**Theorem 2.** The separation of one node of an ESG with |V| nodes leads in worst-case to |V|^2 + 1 test cases to cover all legal and illegal event pairs.

**Proof** (sketched): The total number of edges (including the self-loops) of a digraph with |V| nodes is |V|^2 which determines the number of total number of legal and illegal event pairs. Let \( \delta \) be the number of separated nodes with \( 0 < \delta < n \) before the decomposition. After decomposition, the number of nodes of the original ESG is reduced by \( (\delta - 1) \) because a virtual node has to represent the \( (\delta - 1) \) nodes that had been separated. The new ESG possesses then exactly these \( (\delta - 1) \) nodes. Therefore, after the decomposition we have two ESGs: The previous one with \( \delta - 1 \) edges less and a new one with \( \delta \) edges. Thus, the number of test cases is given by:

\[
f(|V|, \delta) = (|V| - (\delta - 1))^2 + \delta^2
\]

Thus, for \( f(|V|, \delta) \) with \( \delta = 1 \) we have:  
\[
f(|V|, \delta)|_{\delta=1} = f(|V|, \delta)|_{|V|=1} = |V|^2 + 1
\]

**Theorem 3.** The separation of \( \frac{|V|}{2} \) nodes of an ESG with |V| nodes leads in best-case to \( \left(\frac{|V|}{2} + \frac{1}{2}\right)^2 + \left(\frac{|V|}{2} + \frac{1}{2}\right)^2 \) test cases to cover all legal and illegal event pairs.

**Proof** (sketched): For the function \( f(|V|, \delta) = (|V| - (\delta - 1))^2 + \delta^2 \) of the Theorem 2 above, there exists a minimum for \( \delta = \frac{n}{2} \), because

\[
\frac{\partial f(|V|, \delta)}{\partial \delta} = -2|V| + 4\delta - 2 \quad \text{with}
\]

\[
-2|V| + 4\delta - 2 = 0 \Rightarrow \delta = \frac{|V|}{2} + \frac{1}{2}
\]

Therefore, the minimum is \( \frac{|V|}{2} \) in case \( \delta \) is even, and \( \frac{|V|}{2} + \frac{1}{2} \) in case \( \delta \) is odd. Thus, we have

\[
f(|V|, \delta)|_{\delta=\frac{|V|}{2}} = \left(\frac{|V|}{2} + \frac{1}{2}\right)^2 + \left(\frac{|V|}{2} + \frac{1}{2}\right)^2
\]

\[\square\]

5 Tool support

The determination of the MSCESs/MSFCESs can be very time consuming when carried out manually. For that purpose appropriate tools are developed by our group. These set of tools can be viewed as add-ons of a commercial capture-playback tool, e.g., WinRunner of Mercury Interactive.

In our tool environment, the GUI objects represent events for the input, e.g., screens, windows, icons, menus, pointers, commands, function keys, alphabetical keys, etc. The capture mode of the tool deployed records properties of the GUI objects of the SUT.

Our add-on tool, the GUI File Parser, summarizes the relevant information on the GUI structure in a file. On its left side, Fig. 9 depicts the GUI File generated by the capture
tool; the GUI file extended by our File Parser is on the right side. It lists all GUI objects as a tree.

For mapping the GUI objects to the ESG, our *Model Generator* opens the GUI file, previously generated by the GUI File Parser. The objects can be “dragged and dropped” (see the left side of the Fig. 10), and inserted as an end or starting node. Beginning with the end node, the starting node column has to be filled by those objects from which the end node can be reached. By pressing the “Add To Model” button, the selected component is integrated into the model. The end node column can only contain a single node as an end node. When all objects are mapped, the model is complete and can be saved.

Once the model of the SUT is generated by the Model Generator, the GUI file containing the structure information is used to decompose all modal windows. Thus, each modal window is represented by its own ESG, and can separately be tested by generating the corresponding MSFCES.

As shown in Fig. 11, one needs only to push “Convert-Button” to generate an ESG for each modal window.

This will be carried out by our add-on tool *Model Decomposer*.

The *Model Tracer* generates test sequences of the MSFCES and MSFCES from the GUI and of the corresponding model represented by an ESG using the algorithms in Section 4. This is shown on the right column in Fig. 12.
The minimal test sequences that are automatically generated must be converted into the test script language of the selected test tool; this enables the execution and analysis in the replay mode (Fig. 13). The conversion is a straightforward translation, and needs only little additional information that is not included in the test sequence generated: Information about the hierarchical structure of the windows, i.e., whether the objects are to be sequentially executed (i.e., they belong to the same window), or another window is to be activated (opened) before the next object is to be executed.

6 Case study

We now present a case study to determine the effectiveness of the tests generated using the algorithms mentioned in the previous section. The study was conducted using an industrial application: the display unit of a truck to control a set of installments. We did not have access to the source code and any specification of the application, other than its user manual. Hence all ESGs required for test generation were derived from the application GUI.

6.1 System under test (SUT) and event sequence graphs (ESG)

The SUT we use in the examples in this paper is a control terminal of a marginal strip mower (Fig. 14) which controls a marginal strip mower (RSM 13) of a special, heavy-duty vehicle (Unimog of Mercedes-Benz). This display unit takes the optimum advantage of mowing around guide poles, road signs and trees, etc. Operation is effected either by the power hydraulic of a light truck, or by the front power take-off. Further buttons on the control desk (Fig. 14) simplify the operation, so that, e.g., the mower head returns to working position or to transport position when a button is pressed.

The ESGs of the SUT are produced and incrementally extended, starting at a very rudimentary level. Thus, SUT was carefully studied and the set of its ESGs refined in accordance with Definition 4. Thereby, each of the desirable events defines a system function that must be well understood and precisely represented in a corresponding ESG at an appropriate level of granularity.

The ESG in Fig. 15 represents the GUI of the display unit depicted in Fig. 14 which enables the interaction during the working position (work pos.) of the mower. Therein the head of the mower can be shifted left or right depending on the pressure (pres.) being on or off to keep the mower head on the bottom. The pressure is to be activated before the cutter can be started; otherwise a damage is likely on objects that are close to the vehicle. Upon completely carrying out the cutting process the cutter has to be switched off to move the mower into the transport position (trans. pos.).
The vertices of the ESG in Fig. 15 represent user inputs which interact with the system, leading eventually to events as system responses that are expected, i.e., they describe correct, legal situations. Thus, each edge of the ESG represents a pair of subsequent legal events which was defined as an event pair and has to be covered by adequate MSCES. Example 10 lists the results applying the Algorithm 2.

Example 10. MSCES = \{RSM13 < = ; RSM13 work_pos. shift.left shift.left trans.pos. work_pos. shift.right shift.right shift.left shift.right pres.ON pres.OFF pres.ON cutter.OFF cutter.ON cutter.OFF pres.OFF shift.left pres.ON pres.OFF shift.right trans.pos. work_pos. pres.ON pres.OFF trans.pos. < = \}

Furthermore, each FEP has to be covered by an according MSFCES, resulting from completion of the ESG given in Fig. 15. This leads to a set of 75 test cases that is minimized by the sequences which has to be traversed through the ESG completion to reach a FEP. Considering the structural feature of the SUT the number of test cases can be further reduced to 59 test cases for the MSFCES (Fig. 16).

6.2 Results and their analysis

For a comprehensive testing, several strategies have been developed with varying characteristics of the test inputs, i.e., stepwise and scalable increasing and/or changing the length and number of the test sequences, and the type of the test sequences, i.e., CES- and FCES-based, and their
combinations. Following could be observed: The test cases of the length 4 were more effective in revealing dynamic, intricate faults than the test cases of the lengths 2 and 3. Even though more expensive to be constructed and exercised, they are more efficient in terms of costs per detected fault. Further on the CES-based test cases as well as the FCES-based cases were effective in detecting faults.

Due to the lack of space, the experiences with the approach are here very briefly summarized. This can be, however, found in [2] and [25]. To sum up the test process, one student tester carried out 826 tests semi-automatically and detected a total of 39 faults, including some severe ones (Table 1).

In a second stage, the results of the research work for minimizing the spanning set of the test cases (MSCES and MSFCES) have been applied to the testing of the margin strip mower. Table 2 demonstrates that the minimization algorithm (Section 4.2) could save in average about 65% of the total test costs, while the exploitation of the structural information (Section 4.3) of the SUT could further save up to almost 30%.

### 7 Lessons learned

The ESG-based approach has been applied to the testing of the GUIs of different industrial applications; e.g., the GUIs of a mobile telephone device, a ticketing machine, etc. [2]. In addition, the approach has also been used to validate requirements definitions and to verify and design specifications, both mainly represented by ESGs. While some of the results of the analysis of the detected faults were in compliance with the expectations, other results were surprising, and are summarized below.

<table>
<thead>
<tr>
<th>No.</th>
<th>Faults detected by the FCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The cutting unit can be activated without having any pressure on the bottom, which is very dangerous if pedestrians approach the working area (According to the dashed (faulty) arc from “pressure OFF” to “cutter ON” in Fig. 15).</td>
</tr>
<tr>
<td>2.</td>
<td>Keeping the button for shifting the mower head pushed and changing to another screen causes control problems of shifting: The mower head with the cutting unit cannot immediately be stopped in an emergency case.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2 Reducing the number of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length</th>
<th># MSCES without structural information</th>
<th># MSCES with structural information</th>
<th>Cost reduction MSFCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>75</td>
<td>58</td>
<td>22.7%</td>
</tr>
<tr>
<td>3</td>
<td>167</td>
<td>218</td>
<td>35.7%</td>
</tr>
<tr>
<td>4</td>
<td>487</td>
<td>292</td>
<td>40.0%</td>
</tr>
<tr>
<td>Sum</td>
<td>729</td>
<td>568</td>
<td>32.8%</td>
</tr>
</tbody>
</table>
Lesson 1. Start small, but as early as possible

The determination and specification of the CESSs and FCESs should ideally be carried out during the definition of the user requirements, much before the system is implemented; the availability of a prototype would be helpful in this task. They are then a part of the system and the test specification. However, CESSs and FCESs can also be produced incrementally at a later time, even during the test stage, in order to discipline the test process.

As a strategy, one starts with the CESSs and FCESs that cover all event pairs. Test results and quality targets determine how to proceed further, i.e., whether to consider testing with event triples and quadruples.

Lesson 2. Good exception handling is not necessarily expensive but rare

Most GUIs subjected to tests do not consider the handling of the faulty events. They have only a rudimentary, if any, exception handling mechanism, realized by a “panic mode” [26] that mostly leads to a crash, or ignores the faulty events. The number of the exceptions that should be handled systematically, but have not been considered at all by the GUIs of the commercial systems is presumed to be on an average about 80%.

Lesson 3. Analysis prior to testing can reveal conceptual flaws

The analysis of ESGs of the GUIs of some commercial systems has revealed several conceptual flaws: absence of edges, indicating incomplete exception handling, and missing vertices or events (approximately 20%). This amounts to defective components in the final product, highlighting the flaws in the initial concept and the process of product development. In this connection, the proposed approach offers an important unexpected benefit: it provides a framework for the accelerated maturation of the product and for exercising the creativity of the developers.

8 Conclusion and future work

This paper has introduced an integrated approach to coverage testing of human-computer systems, incorporating modeling of the system behavior with fault modeling and minimizing the test sets for the coverage of these models. The framework is based on the concept of “event sequence graphs (ESG)”. Event sequences (ES) represent the human-computer interactions. An ES is complete (CESS) if it produces desirable, well-defined and safe system functionality. An ESG is constructed to reflect the user expectations, the user himself/herself acted as an oracle of a high level of trustworthiness, de facto resolving the oracle problem.

The objective of testing is the construction of a set of CESSs of minimal total length that covers all ESs of a required length. A similar optimization problem arises for the validation of the SUT under undesirable situations. To model the latter problem, faulty event sequences (FESs) are considered. These optimizing problems have been called determination of Minimal Spanning Sets of CESSs and FCESs, respectively. The paper applied and modified some algorithms known from graph theory and conformance testing to the above mentioned problems. The research has shown that the complexity of algorithms that are necessary to solve them is unexpectedly less than the complexity of similar problems, e.g., Chinese Postman Problem, since the vertices of ESGs are equidistant and its edges have no attributes and weights. The next step is to apply the approach to analyze and test safety features; in this case the risks originate from within the system due to potential failures and its spillover effects causing potentially extensive damage to its environment. Another goal for future work is to design a defense action, which is an appropriately enforced sequence of events, to prevent faults that could potentially lead to such failures.

References


Fevzi Belli received the M.S., Ph.D., and Habilitation degrees in electrical engineering and computer science from the Berlin Technical University. He is presently a Professor of Software Engineering in the Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Paderborn, Germany. Prior to this, he headed several projects at a software house in Munich, was a Professor of Computing Science at the Hochschule Bremenhaven and a faculty member of the University of Maryland, European Division. He chaired several international conferences, e.g., ISSRE 1998 and is author and co-author of more than 100 papers published in scientific journals and conference proceedings. His research interests are in testing/fault tolerance/reliability of software and programming techniques.

Christof J. Budnik received the MS degree in electrical engineering and computer science in 2001 from the University of Paderborn. In 2002, he joined the Department of Computer Science, Electrical Engineering and Mathematics at the same University where he is currently a faculty member. His research interests are in the areas of software quality, testing of interactive systems and safety-critical user interfaces.