This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Budnik, Ch. J., Minimal Spanning Set for Coverage Testing of Interactive Systems, First International Colloquium on Theoretical Aspects and Computing (ICTAC), LNCS, Vol. 3407, pp.220-234, Springer-Verlag

Minimal Spanning Set for Coverage Testing of Interactive Systems

Fevzi Belli, Christof J. Budnik

University of Paderborn, Warburger Str. 100, 33098 Paderborn, Germany
{belli, budnik}@adt.upb.de

Abstract. A model-based approach for minimization of test sets for interactive systems is introduced. Test cases are efficiently generated and selected to cover the behavioral model and the complementary fault model of the *system under test* (*SUT*). Results known from state-based conformance testing and graph theory are used and extended to construct algorithms for minimizing the test sets, considering also structural features of the SUT.

1 Introduction

Testing is the traditional validation method in the software industry. There is no justification, however, for any assessment on the correctness of the SUT based on the success (or failure) of a single test, because there can potentially be an infinite number of test cases, even for very simple programs. To overcome this shortcoming of testing, formal methods have been proposed, which introduce models that represent the relevant features of the SUT. The modeled, relevant features are either functional behavior or the structural issues of the SUT, leading to *specification-oriented testing* or *implementation-oriented testing*, respectively. This paper is on specification-oriented testing; i.e., the underlying model represents the system behavior interacting with the user's actions. The system's behavior and user's actions will be 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.

Based on [3], this paper introduces a novel, graphical representation of both the behavioral model and the fault model of the SUT. Algorithms are introduced for the coverage of these models by a minimal set of test cases (*minimal spanning set for coverage testing*). The next section summarizes the related work before Section 3 introduces the fault model and the test process. The optimization of the test case set is discussed in Section 4. Section 5 considers the structure of the SUT to avoid unnecessary and/or infeasible tests. Supporting tools are introduced in Section 6. Section 7 summarizes the results and sketches the research work planned.

2 Related Work

Methods based on finite-state automata have been used for almost four decades for the specification and testing of system behavior, e.g., for specification of software systems [8], as well as for conformance and software testing [6, 1, 20, 18]. Also, the modeling and testing of interactive systems with a state-based model has a long tradition [19, 13, 21, 25]. These approaches analyze the SUT and model the user requirements to achieve sequences of *user interaction* (*UI*), which then are deployed as test cases. [25] introduced a simplified state-based, graphical model to represent UIs; this model has been extended in [3] 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 [22] to test UIs.

A similar fault model as in [3] is used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using *mutation operations* [10]. This approach has been well understood, is widely used and, thus, has become quite popular. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g., integration testing, state-based testing, etc. [9]. Such operations have also been independently proposed by other authors, e.g., "state control faults" for fault modeling in [7], or for "transition-pair coverage criterion" and "complete sequence criterion" in [18]. However, the latter two notions have been precisely introduced in [3] and [25], respectively, earlier than in [18].

Another state-oriented group of approaches to test case generation and coverage assessment is based on model checking, e.g., the Software Cost Reduction method, as described in [12]. These approaches identify negative and positive scenarios to generate and select test cases automatically from formal requirements specifications. A different approach, especially for *graphical user interface* (*GUI*) testing, has been introduced in [16]; it deploys methods of knowledge engineering to generate test cases, test oracles, etc., and to deal with the test termination problem. All of these approaches use some heuristic methods to cope with the state explosion problem.

This paper also presents a method for test case generation and test case selection. Moreover, it addresses test coverage aspects for test termination, based on [3], which introduced the notion of "minimal spanning set of complete test sequences", similar to "spanning set", that was also discussed in [15]. The present paper considers existing approaches to optimize the round trips, i.e., the Chinese Postman Problem [1], and attempts to determine algorithms of less complexity for the spanning of walks, rather than tours, related to [24, 17].

3 Fault Model and Test Process

This work uses *Event Sequence Graphs (ESG)* for representing 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.

3.1 Preliminaries

Definition 1. An Event Sequence Graph ESG=(V,E) is a directed graph with a finite set of *nodes* (*vertices*) $V \neq \emptyset$ and a finite set of *arcs* (*edges*) $E \subseteq V \times V$.

For representing user-system interactions, the nodes of the ESG are interpreted as events. The operations on identifiable components of the UI are controlled/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 system outputs.

Definition 2. Let V, E be defined as in Def. 1. Then any sequence of nodes $\langle v_0,...,v_k \rangle$ is called an (legal) *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i=0,...,k-1$.

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial node and end node of an ES, i.e., $\alpha(ES)=v_0$, $\omega(ES)=v_k$. Finally, the function l (*length*) of an ES determines the number of its nodes. In particular, if $l(ES)=1$ then $ES=\langle v_i \rangle$ is an ES of length 1. An $ES=\langle v_i, v_k \rangle$ of length 2 is called an *event pair* (*EP*).

The assumption is made that there is an ES from the single node ε to all other nodes, and from all nodes there is an ES to the single node γ ($\varepsilon, \gamma \notin V$). ε is called the *entry* and γ is called the *exit* of the ESG.

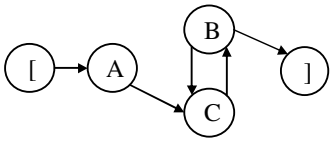

Fig. 1. An ESG with [as entry and] as exit

The entry and exit, represented by '[' and ']', respectively, are not included in V. They enable a simpler representation of the algorithms to construct minimal spanning test case sets (Section 4).

Definition 3. An ES is called a *complete ES* (*Complete Event Sequence*, *CES*), if $\alpha(ES)=\varepsilon$ is the entry and $\omega(ES)=\gamma$ is the exit.

CESs represent *walks* from the entry '[' of the ESG to its exit ']'.

Definition 4. The node *w* is a *successor event* of *v* and the node *v* is a *predecessor event* of *w* if *(v, w)∈ E*. The *difference* of a node *u∈ V diff(u)* is defined as the number of predecessor events reduced by the number of successor events.

Definition 5. Let two ESGs be defined as $ESG_i = (V_i, E_i)$, $i = 1,2$. ESG_1 is a *subgraph* of ESG_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. ESG_1 is an induced subgraph by a set of nodes.

3.2 Fault Model and Test Terminology

Definition 6. For an *ESG=(V, E)*, its *completion* is defined as $\widehat{ESG} = (V, \widehat{E})$ with $\widehat{E} = V \times V$.

Definition 7. The *inverse* (or *complementary*) *ESG* is then defined as $\overline{ESG} = (V, \overline{E})$ with $\overline{E} = \widehat{E} \setminus E$ (\: set difference operation).

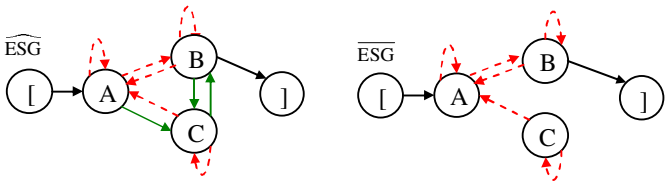

Fig. 2. The completion $\widehat{ESG}$ and inversion ESG of Fig. 1

Note: Entry and exit are not considered while constructing the $\overline{ESG}$.

Definition 8. Any EP of the ESG is a *faulty event pair* (*FEP*) for ESG.

Definition 9. Let $ES=\langle v_0,\ldots,v_k\rangle$ 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 according ESG . The concatenation of the ES and FEP forms then a *faulty event sequence* $FES=\langle v_0,\ldots,v_k, v_m\rangle$.

Definition 10. An *FES* will be called *complete* (*Faulty Complete Event Sequence*, *FCES*) if $\alpha(FES)=\varepsilon$ is the entry. The ES as part of a FCES is called a *starter*.

3.3 Test Process

Definition 11. 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*).

Once a test set has been constructed, tests can be run applying the test cases to the SUT. If it behaves as expected, the SUT *succeeds* the test, otherwise it *fails* the test. 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 proceed it and thus, to succeed the test. Accordingly, if a FCES is used as a test input, a failure

is expected to occur. The latter case represents an exception that must be properly handled by the system, i.e., the SUT is supposed to refuse the proceeding and produce a warning. The test process is sketched in Algorithm 1.

To determine the point in time in which to stop testing, a criterion is necessary to systematize the test process and to judge the efficiency of the test cases. The approach converts this problem into the *coverage of the ES and FES of length k of the* $\widehat{ESG}$.

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 function1 TO n DO
  Generate appropriate ESG and ESG
  FOR k:=2 TO length DO                                    //Section 4.3
    Cover all ESs of length k by means of CESs
    subject to minimizing the number and total length of the CES //Section 4.1
  Cover all FEPs of by means of FCESs
  subject to minimizing the total length of the FCESs      //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
```

The test costs are given by the minimized total length of the CESs and FCESs. The length of the ESs can be increased stepwise. This enables a scalability of the test costs which are proportional to the length of the ESs.

4 Minimizing the Spanning Set

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 EPs 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 EPs 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 [24], MSCESs can be constructed as follows:

- Check whether an ideal walk exists.
- If not, check whether entire walks exist. If yes, construct a minimal one.
- If there is no entire walk, construct a set of walks with minimal total length to cover all ES.

4.1 An Algorithm to Determine Minimal Spanning Complete Event Sequence

A similar problem to the determination of MSCESs is the *Directed Chinese Postman Problem* [23]. In the following, some results are summarized that are relevant to determine the test costs and enable scalability of the test process.

Algorithm 2. Determination of the MSCES

Input: *ESG=(V, E)*; ε=[, γ=]
Output: *MSCES*

```
addArc(ESG,(γ,ε));                    //insert arc from ] to [
sets A,B,M,MSCES := ∅                 //empty sets
FOR EACH v∈V DO
  IF (diff(v)>0) THEN
    A := A ∪ {v_i |i∈{1,..,diff(v)}};
  IF (diff(v)<0) THEN
    B := B ∪ {v_i |i∈{1,..,diff(v)}};
m := |A| := |B|;                      //cardinality
D[1..m][1..m];                        //distance matrix D
FOR EACH v∈A DO          //compute all shortest paths from v to all b∈B
  computeShortestPaths(v,B,D);  //shortest distances are saved in D
M := solveAssignmentProblem(D);
 //M = {(i, j) | one-to-one mapping: i∈{1,..,m}→j∈{1,..,m}} by Hungarian method
FOR EACH (i,j)∈M DO
  Path := getShortestPath(i,j);
  FOR EACH e∈Path DO
    addArc(ESG,e);
EulerTourList := computeEulerTour(ESG);  //tour starts in ε
                              //EulerTourList = (ε,...,γ,ε,...,γ,ε,...,γ,ε)
start := 1;
FOR i:=2 TO length(EulerTourList)-1 DO
  IF (getElement(EulerTourList,i) = γ) THEN
    MSCES := MSCES ∪ getPartList(EulerTourList,start,i);
    start := i+1;          //MSCES = {(ε,...,γ),(ε,...,γ),(ε,...,γ),...}
RETURN MSCES;
```

The Algorithm 2 determines a set of walks with the minimal total length to cover all EPs and requires that this graph be strongly connected, which can be done through an additional arc from the final to the entry (Fig. 3). The figures within the nodes in Fig. 3 indicate the calculated differences (Definition 4) of these nodes. These balance values determine the number of additional EPs that will be identified by searching the all-shortest-path and solving the optimization problem by the Hungarian method [14]. The problem can then be transferred to the construction of the Euler tour for this graph [24].

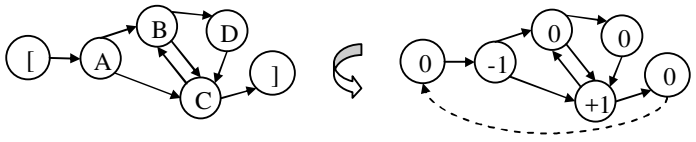

Fig. 3. Transferring walks into tours and balancing the nodes

The function `addArc(ESG,(u,v))` inserts a new arc from the node *u* to the node *v* of the *ESG*. The function `computeShortestPath()` determines all shortest paths from a node *v* to all $b \in B$ with BFS algorithm and stores these shortest distances in the matrix *D* for later usage by the function `getShortestPath()`. The function `solveAssignmentProblem()` returns a one-to-one mapping of the unbalanced nodes. The function `computeEulerTour()` determines the Euler tour of the ESG. The euler tour will be decomposed into subsequences by the function `getPartList()`.

In Algorithm 2, the ESG is represented by its adjacency matrix. The algorithm consists of three sections:

- Determination of all-shortest-paths by Floyds algorithm with the complexity O(|V|3) [2]. However, because the ESG is a non-weighted digraph, the complexity can be decreased by using the Breadth-First-Search down to O(|V|·|E|). This results from the fact that:
  - Breadth-First-Search algorithm determines the shortest path from one node of the ESG to all other ones in $O(|E|)$ as $|E|>|V|+1$.
  - Breadth-First-Search algorithm iterates $|V|$ times to handle all nodes.
- The optimizing problem, which is solved in accordance with [14] by the Hungarian method, with the complexity $O(|V|^3)$.
- Computation of an Euler tour with the complexity of O(|V|·|E|) [24].

To sum up, the MSCES can be solved in $O(|V|^3)$ time. Note that no entire walk exists for the example. Therefore, an ideal walk cannot be constructed.

4.2 Minimal Spanning Set for the Coverage of Faulty Event Sequences

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 exactly $u:=n^2-d$ arcs are FEPs. Thus, at most *u* FCESs of minimal length, i.e., of length 2, are available; those FCESs emerge when the node(s) after entry is (are) followed immediately by a faulty input. 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 [11].

While constructing the MSCESs one can exclude the ESs that are already used to form starters to construct MSFCESs. This can help save costs if the test budget is very limited, as is very often the case in practice.

4.3 Generating Event Sequences with Length > 2

A phenomenon in testing interactive systems most testers are familiar with, is that faults can be frequently detected and reproduced only in some context. This makes a test sequence of a length>2 necessary since repetitive occurrences of some subsequences are needed to cause an error to occur/re-occur.

Consider the following scenario: Based on the ESG given in Fig. 4, the tester assumedly observes that the EP given by ***BC*** always reveals a fault, no matter if executed within *[A**BC**]*, *[ABA**BC**]*, or *[ABDC**BC**]*; i.e., the test cases containing *BC* always detect the fault in any context. In this case, the fault is said to be a *static* one, as it can be detected without a context. Furthermore, the same scenario (so the assumption) demonstrates that the EP ***BA*** reveals another fault, but only in the context of *[ABC**BA**C]*, and never within *[A**BA**C]*, or *[A**BA**CBDC],* etc. In this case the fault is said to be a *dynamic* one.

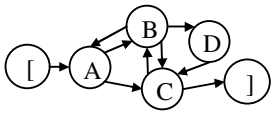

Fig. 4. Static faults vs. dynamic faults

Such observations clearly indicate that the test process must be applied to longer ESs than 2 (EPs).

Therefore an ESG can be transformed into a graph in which the nodes can be used to generate test cases of length > 2, in the same way that the nodes of the original ESG are used to generate EPs and to determine the appropriate MSCES.

Fig. 5 illustrates the generation of ESs of length=3. In this example adjacent nodes of the extended ESG are concatenated, e.g., AB is connected with *BD*, leading to *ABBD*. The shared event, i.e., *B*, occurs only once producing *ABD* as an ES of length=3. In case ESs of length=4 are to be generated, the extended graph must be extended another time using the same algorithm.

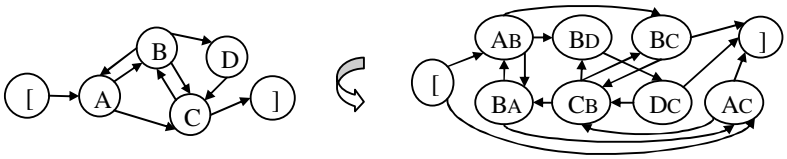

Fig. 5. Extending the ESG for covering ESs of length=3

The common valid of this approach is given by Algorithm 3. Therein the notation `ES(ESG,i)` represents the identifier, e. g., *AB*, of the node *i* of the *ESG*. This identifier can be concatenated with another identifier `ES(ESG,j)` of the node *j*, e.g., *CD*. This is represented by $AB \oplus CD$, or `ES(ESG,i)` $\oplus$ `ES(ESG,j)`, resulting in the new identifier *ABCD*. Note that the identifiers of the newly generated nodes to extend the ESG will be made up using the identifiers of the existing nodes. The function `addNode()` inserts a new ES of length *k*. Following this step, a node *u* is connected with a node *v* if the last *n-1* events that are used in the identifier of *u* are the same as the first *n-1* events that are included in the identifier of *v*. The function `addArc()` inserts an arc, connecting *u* with *v* in the ESG. The pseudo nodes [,] are connected with all the extensions of the nodes with which they were connected before the extension. In order to avoid traversing the entire matrix, arcs which are already considered are to be removed by the function `removeArc()`.

Algorithm 3. Generating ESs and FESs with length > 2

Input: *ESG=(V, E)*; ε =[, γ=], *ESG'=(V', E')* with *V'*=$\varnothing$, ε'=[, γ'=];
Output: *ESG'=(V', E')*, ε'=[, γ'=];

```
FOR EACH (i,j)∈ E with (i<>ε) AND (j<>γ) DO
  addNode(ESG',(ES(ESG,i) ⊕ ω(ES(ESG,j)));
  removeArc(ESG,(i,j));
FOR EACH i∈ V' with (i<>ε') AND (i<>γ') DO
  FOR EACH j∈ V' with (j<>ε') AND (j<>γ') DO
    IF(ES(ESG',i) ⊕ ω(ES(ESG',j)) =
                    α(ES(ESG',i)) ⊕ (ES(ESG',j)) THEN
      addArc(ESG',(i,j));
  FOR EACH (k,l)∈E with k=ε DO
    IF(ES(ESG',i) = ES(ESG,l) ⊕ω(ES(ESG',i)) THEN
      addArc(ESG',(ε',i));
  FOR EACH (k,l)∈ E with l=γ DO
    IF(ES(ESG',i) = α(ES(ESG',i)) ⊕ ES(ESG,k) THEN
      addArc(ESG',(i,γ'));
RETURN ESG';
```

Apparently, the Algorithm 3 has a complexity of $O(|V|^2)$ because of the nested FOR-loops to determine the arcs in the *ESG'*. A further algorithm to generate FESs of length > 2 is not necessary because such faulty sequences will be constructed through the concatenation of the appropriate starters with the FEPs. Algorithm 2 can be applied to the outcome of the Algorithm 3, i.e., to the extended ESG, to determine the MSCES for *l(ES)* > 2.

5 Exploiting the Structural Features

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. 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.

5.1 A Practical Example

Fig. 6 depicts a small part of the GUI of an 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). The optional events are abbreviated in the Fig. 7 with capital letters. There are still more window components, but they will not be explained here further. The described components are used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

The GUI represented in Fig. 6 is transferred to an ESG (Fig. 7). Fig. 6 is easy to understand, but an informal and imprecise presentation of the GUI, while Fig. 7 is a formal presentation that neglects some aspects, e.g., the hierarchy, while still being precise.

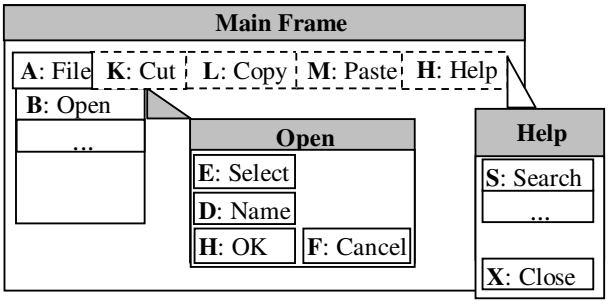

Fig. 6. Top-level GUI of WordPad, modal/modeless windows

The conversion of Fig. 6 into Fig. 7 is the most abstract step of the approach that must be done manually, requiring some practical experience and theoretical skill in designing GUIs. Example 1 lists the FCESs to cover the FEPs of the ESGs `Main/Open` given in Fig. 7.

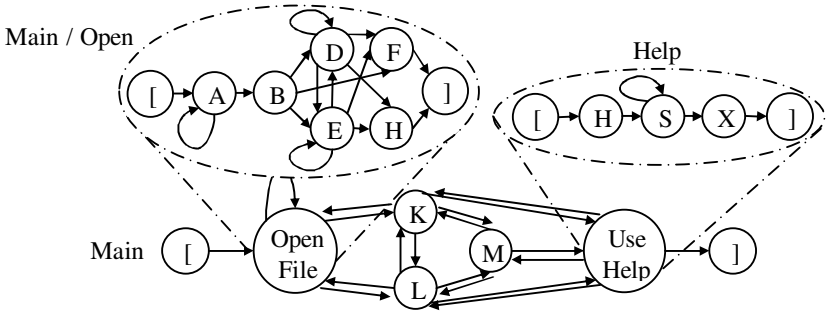

Fig. 7. ESG of the GUI represented in Fig. 6.

Example 1. *AD*, *AE*, *AF*, *AH*, *ABA*, *ABB*, *ABH*, *ABDA*, *ABDB*, *ABEA*, *ABEB*, *ABFB*, *ABFF*, *ABFE*, *ABFD*, *ABFH*, *AB(E+D)HA*, *AB(E+D)HB*, *AB(E+D)HD*, *AB(E+D)HE*, *AB(E+D)HF*, *AB(E+D)HH*

5.2 Modal and Modeless Windows

Analysis of the structure of the GUIs, e.g., the example GUI in Fig. 6, 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).

For the main frame of the WordPad, the child window `Help` is a modeless window; the other child window, `Open`, is a modal one. Fig. 8 represents these windows as a "family tree". In this tree, a unidirectional edge indicates a modal parent-child relationship. A bidirectional edge indicates a modeless one.

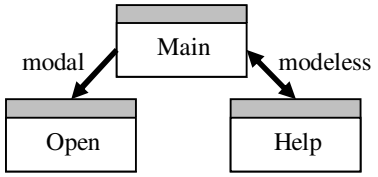

Fig. 8. Modal windows vs. modeless windows

Modal windows must be closed before any other window can be invoked. Therefore, modal windows can be tested without taking the other windows into account, i.e., it is not necessary to consider the combinations of the ESs and FESs of the parent and children. Thus, similar to the strong-connectedness and symmetrical features

[21], the modality feature is extremely important for testing since it avoids unnecessary test efforts. Note that this is true only for the FCESs and MSFCESs as test inputs considering the structure information might impact the structure of the ESG, but not the number of the CESs and MSCESs as test inputs. Fig. 9 represents the modified ESG of the WordPad. The modification, which separates the events *A* and *B* from `Open`, takes the modality into account that avoids unnecessary combinations of EPs and FEPs. Example 2 lists the MSFCESs to cover FEPs of the sub-graph `Open` given in Fig. 9.

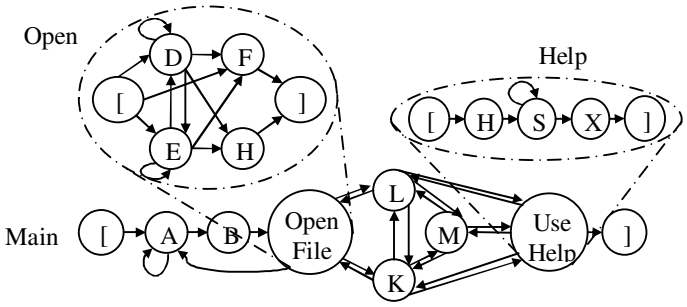

Fig. 9. Modified ESG of the GUI in Fig. 8, taking the modality feature into account.

Example 2. *(E+D)FD*, *(E+D)FE*, *(E+D)FF*, *(E+D)FH*, *(E+D)HF*, *(E+D)HD*, *(E+D)HF*, *(E+D)HH*

Already this example, i.e., the comparison of Example 1 (22 FEPs) with Example 2 (8 FEPs), demonstrates the efficiency increase through the exploitation of the structural features of the SUT.

6 Tool Support

The determination of the MSCESs/MSFCESs can be very time consuming when carried out manually. Also, the gaining of the structural information that is necessary to reduce the number of MSFCESs is frequently a rather costly process. Thus, tools of different categories are necessary for both purposes.

A good software engineering practice ensures that the system behavior has been modeled during the system design. Otherwise the model has to be constructed manually afterwards, according to the specification.

6.1 Test Case Generation

For the generation of test cases the tool *GenPath* [5] has been developed to accept the adjacency matrix of the ESG as input. The user can, however, input several ESGs which can also be subgraphs of the vertices of the ESG itself under consideration.

Fig. 10 represents the GUI of GenPath which generates MSCESs for ESs of required length. Moreover, it represents the ESG under consideration and marks its EPs with the underlying algorithm traces.

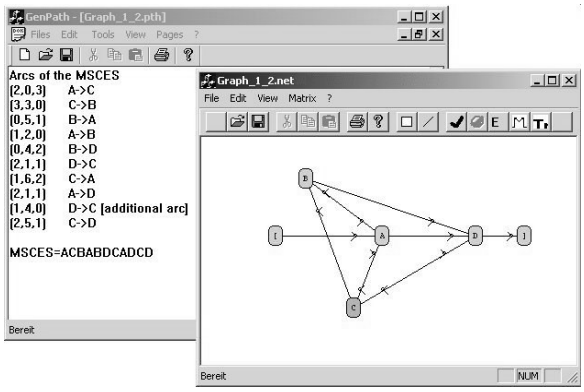

Fig. 10. GenPath to generate MSCES

6.2 Generation of GUI Structure

Section 5 explained the necessity to consider the specific information on the structure of the SUT in order to reduce the number of test cases. This structural information can be obtained with a commercially available Capture-Playback facility, as to WinRunner of Mercury Interactive [26] delivers.

```
Open:
{class: window,
 label: Open,
 enabled: 1,
 module_name: "C:\\Windows\\Tool\\WordPad.exe",
 nchildren: 17
}
  { rtree_state: open,
    ltree_state: open,
    lrn_app_stat: done,
    parent_win: "WordPad - [Document 1]",
    opened_by: "menu_select_item(\"Open... Strg+O\");"
  }
```

Fig. 11. Excerpt out of the WinRunner file with information on a GUI structure

This tool can identify all available windows of a GUI-application and generates automatically information on the windows hierarchy that can be assembled to determine modal/modeless windows of the SUT. Fig. 11 represents a part of WordPad that the test environment has traced. The keyword `opened_by` identifies the child window `Open`. The parent window can be traced via the keyword `menu_select_item()`.

7 Validation

A separate study has applied the proposed approach to a selected significant function of the personal music management system RealJukebox (RJB), Version 2, of RealNetworks. This function enables the user to load a CD, select a track, and play it. The user can then change the mode, replay the track, or remove the CD, load another one, etc.

For a comprehensive testing, several strategies have been developed with varying characteristics of the test inputs, i.e.,

- the length and number of the test sequences, and
- the type of the test sequences, i.e., CES- and FCESs-based.

This study delivered following findings:

- The test cases of the length 4 were more effective in revealing dynamic faults than the test cases of the lengths 2 and 3. They were, however, considerably more expensive in terms of costs per detected fault.
- The CES-based test cases as well as the FCES-based cases were effective in detecting faults.

To summarize the test process, one student tester, who acted also as oracle, carried out 1166 tests semi-automatically over a period of 2 days, working, on average, 8 hours per day, thus spending a total of 78560 seconds. These figures result in approximately 67 seconds per test. A total of 32 faults were detected.

Table 1. Reducing the number of test cases

Length	#CES	#MSCES	Cost Reduction ES
2	40	15	62.5 %
3	183	62	66.1 %
4	849	181	78.7 %
Sum	1072	258	76.0 %
Length	#MSFCES without structural information	#MSFCES with structural information	Cost Reduction MSFCES
2	75	58	22.7 %
3	339	218	35.7 %
4	1587	632	60.2 %
Sum	2001	908	54.6 %

The results of the research for minimizing the spanning set of the test cases (MSCES and MSFCES), as described in Section 4, has been applied to the testing of the selected significant function. Table 1 summarizes that the algorithmic minimization (Section 4.1 and 4.2) could save about 75 % of the test costs, while the exploitation of the structural information of the SUT could save up to almost 50%!

A more detailed discussion about the benefits, e.g., concerning the number of detected errors in dependency of the length of the test cases, is given in [4].

8 Conclusion and Future Work

This paper has introduced an integrated approach to coverage testing of interactive 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 (CES) if it produces desirable, well-defined and safe system functionality. The notion of complete faulty event sequences mathematically complements this view.

The objective of testing is the construction of a set of CESs of minimal total length that covers all ESs of a required length. A similar optimization problem arises for the validation of the SUT under exceptional, undesirable situations which are modeled by faulty event sequences (FESs) and complete FESs (FCESs). The paper applied and modified some algorithms known from graph theory to these problems. Furthermore, it was shown how the structure of interactive systems can be algorithmically exploited by a commercial test tool to reduce the test sets by infeasible and/or unnecessary test cases.

In the case of *safety*, the threat originates from within the system due to potential failures and its spillover effects causing potentially extensive damage to its environment. The goal for future work is to design defense actions, which is an appropriately enforced sequence of events, to prevent faults that could potentially lead to such failures. Further future work concerns cost reduction through automatic, or semiautomatic modification of a given ESG in order to consider modality of interaction structures.

References

1. A. V. Aho, A. T. Dahbura, D. Lee, M. Ü. Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours", *IEEE Trans. Commun.* 39, pp. 1604-1615, 1991
2. R. K. Ahuja, T. L. Magnanti, J. B. Orlin, "Network Flows-Theory, Algorithms and Applications", Prentice Hall, 1993.
3. F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces", Proc. 12th ISSRE, pp. 34-43, 2001

4. F. Belli, N. Nissanke, Ch. J. Budnik, “A Holistic, Event-Based Approach to Modeling, Analysis and Testing of System Vulnerabilities”; Technical Report TR 2004/7, Univ. Paderborn, 2004
5. Ch. J. Budnik, A. Hollmann, R. Moge, „GenPath – A Tool to Generate Paths of different Lengths of an Event Sequence Graph”, Technical Report TR 2004/9, Univ. Paderborn, 2004
6. R.V. Binder, “*Testing Object-Oriented Systems*”, Addison-Wesley, 2000
7. G. V. Bochmann, A. Petrenko, “Protocol Testing: Review of Methods and Relevance for Software Testing”, *Softw. Eng. Notes, ACM SIGSOFT*, pp. 109-124, 1994
8. Tsun S. Chow, “Testing Software Designed Modeled by Finite-State Machines”, *IEEE Trans. Softw. Eng*. 4, pp. 178-187, 1978
9. M.E. Delamaro, J.C. Maldonado, A. Mathur, “Interface Mutation: An Approach for Integration Testing”, *IEEE Trans. on Softw. Eng.* 27/3, pp. 228-247, 2001
10. R.A. DeMillo, R.J. Lipton, F.G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, *Computer* 11/4, pp. 34-41, 1978
11. Edsger. W. Dijkstra, “A note on two problems in connexion with graphs.”, Journal of Numerische Mathematik, Vol. 1, pp. 269-271, 1959
12. A. Gargantini, C. Heitmeyer, „Using Model Checking to Generate Tests from Requirements Specification”, *Proc. ESEC/FSE ’99*, ACM SIGSOFT, pp. 146-162, 1999
13. J. Jorge, N.J. Nunes, J.F. Cunha (Eds.), “Interactive Systems – Design, Specification, and Verification”, LNCS 2844, Springer-Verlag, 2003
14. D.E. Knuth, “The Stanford GraphBase”, Addison-Wesley, 1993
15. M. Marré, A. Bertolino, “Using Spanning Sets for Coverage Testing”, *IEEE Trans. on Softw. Eng.* 29/11, pp. 974-984, 2003
16. A. M. Memon, M. E. Pollack and M. L. Soffa, “Automated Test Oracles for GUIs”, *SIGSOFT 2000*, pp. 30-39, 2000
17. S. Naito, M. Tsunoyama, “Fault Detection for Sequential Machines by Transition Tours”, *Proc. FTCS*, pp. 238-243, 1981
18. J. Offutt, L. Shaoying, A. Abdurazik, and Paul Ammann, “Generating Test Data From State-Based Specifications”, *The Journal of Software Testing, Verification and Reliability, 13(1):25-53, Medgeh 2003.*
19. 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
20. B. Sarikaya, “Conformance Testing: Architectures and Test Sequences”, *Computer Networks and ISDN Systems* 17, North-Holland, pp. 111-126, 1989
21. R. K. Shehady and D. P. Siewiorek, “A Method to Automate User Interface Testing Using Finite State Machines”, in *Proc. Int. Symp. Fault-Tolerant Computing FTCS-27*, pp. 80-88, 1997
22. K. Tai, Y. Lei, “A Test Generation Strategy for Pairwise Testing”, *IEEE Trans. On Softw. Eng.* 28/1, pp. 109-111, 2002
23. H. Thimbleby “The Directed Chinese Postman Problem”, School of Computing Science, Middlesex University, London
24. D.B. West, “Introduction to Graph Theory”, Prentice Hall, 1996
25. L. White and H. Almezen, “Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences”, in Proc ISSRE, IEEE Comp. Press, pp. 110-119, 2000
26. WinRunner, Mercury Interactive, `http://www.mercuryinteractive.com`