Finite State Testing and Analysis of Graphical User Interfaces

Fevzi Belli (belli@upb.de),
University of Paderborn (http://adt.upb.de), Germany

Abstract

Based on finite-state automata (FSA) and equivalent regular expressions, the paper introduces a holistic view of fault modeling that can be carried out as a complementary step to system modeling, revealing much rationalization potential. Appropriate formal notions will be used to introduce efficient algorithms to systematically generate and select test cases. The completeness of the test can be determined exploiting the link coverage of the state transition diagram of the FSA that models both the desired and undesired behavior of the system under test; this enables a precise scalability of the test and analysis process, leading to a better cost-effectiveness The elements of the approach will be narrated by realistic examples which will be used also to validate the approach.

1. Introduction and Terminology

There are two distinct types of construction work while developing software:

- Design, implementation, and test of the *programs*.
- Design, implementation, and test of the *user interface (UI)*.

We assume that UI might be constructed separately, as it requires different skills, and maybe different techniques than construction of common software. 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. Testing requires both: a good understanding of user requirements, and familiarity with the technical equipment. This paper is about UI testing, i.e. testing of the programs that materialize the UI, taking the design aspects into account. To some extent, also analysis aspects will be covered because testing and analysis usually belong together.

Graphical User Interfaces (GUIs) have become more and more popular and common UIs in computer-based systems. Testing GUIs is, on the other hand, a difficult and challenging task for many reasons: First, the input space possesses a great, potentially indefinite number of combinations of inputs and events that occur as system outputs; external events may interact with these inputs. Second, even simple GUIs possess an enormous number of states which are also due to interaction with the inputs. Last but not least, many complex dependencies may hold between different states of the GUI system, and/or between its states and inputs.

Nevertheless, nowadays it will be taken for granted that most Human-Computer-Interfaces (HCI) will be materialized via GUI. There exists a vast amount of research work for specification of HCI, there has been, however, little well known systematic study in this field, resulting an effective testing strategy which is not only easy to apply, but also *scalable* in sense of stepwise increasing the test complexity and accordingly the test coverage and completeness of the test process, thus also increasing the test costs in accordance with the test budget of the project. This paper presents a strategy to systematically test GUIs, being capable of test case enumeration for a precise test scalability.

Test cases generally require the determination of meaningful test *inputs* and expected system *outputs* for these inputs. Accordingly, to generate test cases for a GUI, 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, alphanumerical keys, etc. The *objective* of a test is to generate the expected system behavior (*desired event*) as an output by means of well-defined test input, or inputs. In a broader sense, the test object is the software 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 *undesired events* into the SUT. Such events would usually transduce the software 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 GUI usually represent sequences of GUI objects activities and/or selections that will operate interactively with the objects (*Interaction Sequences – IS*, [28], see also [15], "Event Sequences"). Such an interactive sequence is *complete (CIS)*, iff it eventually invokes

1071-9458/01 $17.00 © 2001 IEEE

the desired system responsibility. From Knowledge Engineering point of the view, the testing of GUI represents a typical *planning* problem that can be solved goal-driven [16]: 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 GUI test problem described above, this means we have to construct the test sequences in dependency of both the desired, correct events and the undesired, faulty events. A major problem is the unique distinction between correct and faulty events (*Oracle Problem,* [16]). Our approach will exploit the concept of CIS to elegantly handle the Oracle Problem.

Another tough problem while testing is the decision when to stop testing (*Test Termination Problem and Testability* [10, 12]). Exercising a set of test cases, the test results can be satisfactory, but this is limited to these special test cases. Thus, for the quality judgement of the program under test one needs further, rather quantitative arguments, usually materialized by well-defined *coverage criteria*. The most well known coverage criteria base either on special, structural issues of the program to be tested (*implementation orientation/white-box testing*), or its behavioral, functional description (*specification orientation/black-box testing*), or both, if both implementation and specification are available (*hybrid/gray-box testing*).

The present paper will summarize 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. The favored methods for modeling concentrate on finite-state–based techniques, i.e. state transition diagrams and regular events. For the systematically, scalable generating and selection of test sequences, and accordingly, for the test termination, the notion *Edge Coverage* of the state transition diagram will be introduced. Thus, our approach is addressed primarily to the specification-oriented testing. It enables an incremental refinement of the specification which may be at the beginning rough and rudimentary, or even not existing. The approach can be, however, also deployed in implementation-oriented testing, in a refined format, e.g. using the implementation (source code as a concise description of the SUT, i.e. as the ultimate specification) and its control flow diagram as a finite-state machine and as a state transition diagram, respectively.

Section 2 introduces the notion of finite-state modeling and regular expressions which will be used both for modeling the system and the faults through interaction sequences. Cost aspects will be discussed in Section 3; a basic test coverage metric will be introduced to justifiable generate test cases. An optimization model will be introduced to solve the test termination problem. Some potentials of test cost reduction will be discussed; Section 3 includes further rationalization aspects as automatically executing test scripts that have been specified through regular expressions. Further examples and discussion on the validation of the approach will be given in Section 4. Section 5 discusses the approach, considering related work, and concludes the paper summarizing the results.

Putting the different components of the approach together, a holistic way of modeling of software development will be materialized, with the novelty that the complementary view of the desired system behavior enables to obtain the precise and complete description of undesired situations, leading to a systematic, scalable, and complete fault modeling.

2. Integrating System Modeling with Fault Modeling

While developing a system, the construction usually starts with creating a model of the system to be built, in order to better understand its "look and feel" [26], including its overall external behavior, mainly in order to validate the user requirements. Thus, modeling of a system requires the ability of abstraction, extracting the relevant issues and information from the irrelevant ones, taking the present stage of the system development into account. While modeling a GUI, the focus is usually addressed rather to the correct behavior of the system as *desired* situations, triggered by legal inputs. Describing the system behavior in *undesired,* exceptional situations which will be triggered by illegal inputs and other undesired events are likely to be neglected, due to time and cost pressure of the project. The precise description of such undesired situations is, however, of decisive importance for a user-oriented fault handling, because the user has not only a clear understanding how *his* or *her* system functions properly, but also which situations are not in compliance with his or her expectations. In other words, we need a specification to describe the system behavior both in legal and illegal situations, in accordance with the expectations of the user. Once we have such a complete description, we can then also precisely specify our hypotheses to detect undesired situations, and determine the due steps to localize and correct the faults that cause these situations.

Summarizing the discussion about a good modeling, we need a formal specification tool with following capabilities:

- *Generic,* i.e. describing both legal and illegal situations.
- *Recognizing, i.e.* distinguishing between legal and illegal situations.
- *Operable*, i.e. enabling calculations, based on efficient, verifiable algorithms for a quantitative view, e.g. through the enumeration of the generated test cases, assigning them weights in the order of their importance for test coverage, etc. The operability is best given by an algebra, consisting of well-defined operations according to a calculus, an order relation and neutral element(s).

These requirements are fulfilled by Finite-State Automata (FSA) and Regular Expressions (RegEx), having equivalent recognition and generation capabilities, and building an *event algebra* [22].

2.1 Finite-State Modeling of GUI

Deterministic finite-state automata (FSA), also called finite-state, sequential machines have been successfully used for many decades to model sequential systems, e.g. logic design of both combinatorial and sequential circuits [9, 18], protocol conformance of open systems [6], compiler construction [1], but also for UI specification and testing [19, 28]. FSA are broadly accepted for the design and specification of sequential systems for good reasons. First, they have excellent recognition capabilities to effectively distinguish between correct and faulty events/situations. Moreover, efficient algorithms exist for converting FSA into equivalent regular expressions (RegEx), and v.v. [22]. RegEx, on the other hand, are traditional means to generate legal and illegal situations and events systematically.

A FSM can be represented by
- a set of inputs, a set of outputs, and a set of states,
- an output function that maps pairs of inputs and states to outputs,
- a next-state function that maps pairs of inputs and states to next states.

This is rather an informal, but nevertheless sufficiently precise definition which will be used in this paper; for a formal definition, see [22]. For representing GUI, we will interpret the elements of FSA as follows
- Input set: Identifiable objects that can be perceived and controlled by input/output devices, i.e. elements of WIMPs (Windows, Icons, Menus, and Pointers).
- Output set has two distinct subsets
  + Desired events: Outcomes that the user wants to have, i.e. correct, legal responses,
  + Undesired events: Outcomes that the user does not want, i.e. a faulty result, or an unexpected result that surprises the user.

Please note our following assumptions that do not constrain the generality:
- We use FSA and its state transition diagram (STD) synonymously.
- STDs are directed graphs, having an *entry* node and an *exit* node, and there is at least one path from entry to exit (We will use the notions "node" and "vertex" synonymously).
- Outputs are neglected, in the sense of Moore Automata.
- We will merge the inputs and states, assigning them to the vertices of the STD of the FSA.
- Next-state function will be interpreted accordingly, i.e. inducing the next input that will be merged with the due state.

To sum up, we use the notions "state" and "input" on the one side and "state", "system response" and "output" on the other side synonymously, because the user is interested in external behavior of the system, and not its internal states and mechanisms. Thus, we are strongly focusing to the aspects and expectations of the user.

Any chain of edges from one vertex to another one, materialized by sequences of user inputs-states-triggered outputs defines an *interaction sequence (IS)* traversing the FSA from one vertex to another.

To introduce informally, we assume that a Regular Expression RegEx consists of symbols a, b, c, ... of an alphabet which can be connected by operations
- Sequence (usually no explicit operation symbol, e.g. "ab" means "b follows a"),
- Selection ("+", e.g. "a+b" means "a or b"),
- Iteration ("*", Kleene's Star Operation, e.g. "a*" means "a will be repeated arbitrarily"; "a^+" means at least one occurrence of "a").

Example: $T = [(ab(a+c)^+)^*]$

The symbols of the RegEx can be atomic/terminal symbols, or also regular expressions. Accordingly, they can be interpreted as single actions, or an aggregation of actions. An action can represent a command, a system response, etc.

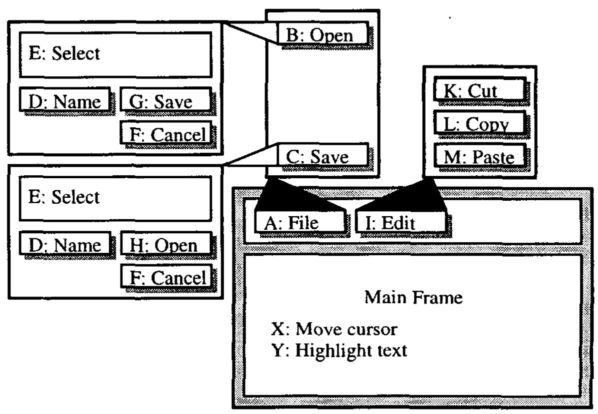


Fig. 1: *Example of a GUI*

Fig. 1 presents a small part of a MS WordPad-like word processing system (see also [16]). This GUI will be usually active when text is to be loaded from a file, or to be manipulated by cutting and pasting, or copying. The GUI will be used also for saving the text in the file (or, in another one). At the top level, the GUI has a pull-down menu with the options `File` and `Edit` that invoke other components, e.g. `File` event opens a sub-menu with `Save As` and `Open` as sub-options. These sub-options have further sub-options. `select` can invoke sub-directories or select files. There are still more window components which will not be described further. The window can be closed by selecting either `Open` or `Cancel`. The described components are used to traverse through the sequences of the menus and sub-menus, creating many different combinations and accordingly, many applications.

Fig. 2 presents the GUI described in the Fig. 1 as a FSA. Again, the terms event, state, and situation will be used here synonymously. Each of the three sub-graphs of the

Fig. 2 presents inputs which interact with the system, leading eventually to events as system responses that are desired situations in compliance with the user's expectation. Based on the sub-graphs, the ICs can be generated.

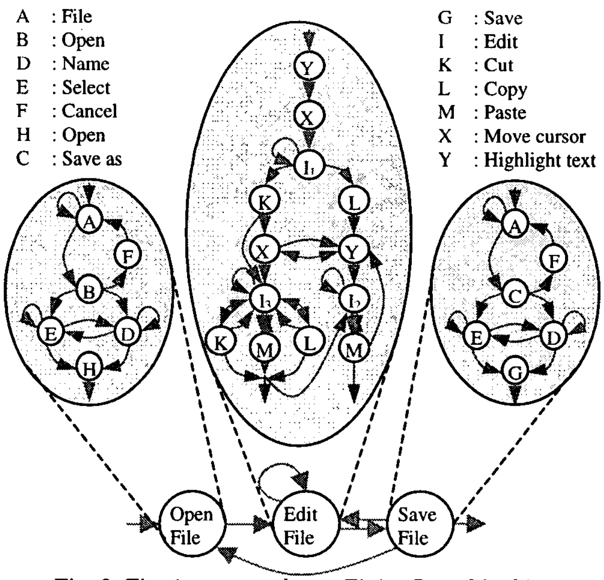


Fig. 2: Fig. 1 *presented as a Finite-State Machine*

The conversion of the Fig. 1 (easy to understand, but informal presentation of the GUI) into Fig. 2 (formal presentation, neglecting some aspects, e.g. the hierarchy) is the most abstract step in our approach that must be done manually, requiring some practical experience and theoretical skill in designing GUIs. As common in modeling process, we choose the events that seem to us most relevant, attempting to adopt the user's view of the picture; there is no algorithmic way to abstract the relevant part from the entire environment. The most of the following job, however, can be carried out at least partly automatically, according to algorithms we describe in this paper.

It cannot be emphasized strongly enough that what we are doing here is an elegant solution of Oracle Problem: Identification of the *Complete Interaction Sequences (CIS)* does present the meaningful, expected system outputs which will be constructed here systematically.

2.2 Interaction Sequences

Once the FSA has been constructed, more information can be gained by means of its state transition graph. First, we can identify now all legal sequences of user-system interactions which may be complete or incomplete, depending on the fact whether they do or do not lead to a well-defined system response that the user expects the system to carry out (Please note that the incomplete interaction sequences are sub-sequences of the complete interaction sequences). Second, we can identify the entire set of the *compatible*, i.e. legal *interaction pairs (IP)* of inputs as the edges of the FSA (Fig. 2b, IPs based on the sub-graph in Fig. 2a). This is key issue of the present approach, as it will enable us to define the *edge coverage* notion as a test termination criterion.

The generation of the CISs and IPs can be based either on the FSA, or more elegantly, on the corresponding RegEx [22], whatever is more convenient for the test engineer (Fig. 2c, RegEx for the sub-graph in Fig. 2a). Finite-state-based techniques have already been successfully used for many years for conformance testing of protocols by many authors [6]. The systematic expansion of the RegEx, as we introduced in [5] is, however, relatively new to generate test cases in a scalable way.

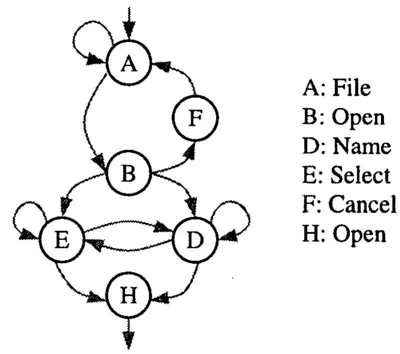


Fig. 2a: *Sub-Graph* `open file`

Sub-Graph	IPs
`Open File`	AA, AB, BD, BE, BF, EH, FA, ED, EE, DD, DE, DH

Fig. 2b: *IPs of the Sub-Graph* `open file`

Sub-Graph	RegEx
`Open File`	$A^{+}B(FA^{+}B)^{*}(E^{+}D^{*}+D^{+}E^{*})^{+}H$

Fig. 2c: *RegEx of the Sub-Graph* `open file`

2.3 Fault Modeling through Interaction Sequences

The causes of faults are mostly:
- The expected behavior of the system has been wrongly specified (*Specification Errors*), or
- the implementation is not in compliance with the specification (*Implementation Errors*).

In our approach, we will exclude the *User Errors*, suggesting that the user is *always* right, i.e. we suggest that there are no user errors. We require that the system must detect all inputs that cannot lead to a desired event, inform the user, and navigate him, or her properly in order to reach a desired situation.

One consequence of this requirement is that we need a view that is complementary to the modeling of the system. This can be done by systematical and stepwise manipulation of the FSA that models the system. For this purpose, we introduce the notion *Faulty/Incompatible Interaction Pairs (FIP)* which consist of inputs that are not legal in sense of the specification. Fig. 3 generates for the sub-graph `open file` of the Fig. 2 the FIP by threefold mani-pulations:

- Add edges in opposite direction wherever only one way edges exists (dotted connections, Fig. 3).
- Add loops to vertices wherever none exists in the specification (dashed-dotted connections, Fig. 3).
- Add edges between vertices wherever none exists (dashed connections, Fig. 3).

Adding all manipulations to the FSA defines the *Completed FSA* (*CFSA*, Fig. 3).

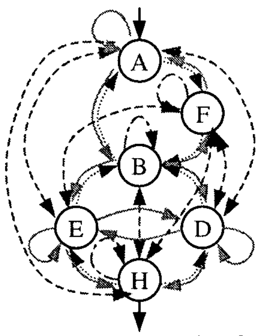


Fig. 3: *CFSA (Completed FSA)*

Now we can construct all potential interaction faults systematically building all illegal combinations of symbols that are not in compliance with the specification (FIPs in Fig. 4a). Once we have generated a FIP, we can extend it through an IS that starts with entry and end with the first symbol of this FIP; we have than a *faulty/illegal complete interaction sequence* (*FCIS*), bringing the system into a faulty situation (Fig. 4b). Please note that the attribute "complete" within the phrase FCIS may not imply that the exit node of the FSA must be necessarily reached; once the system has been conducted into a faulty state, it cannot accept further illegal inputs, in other words, an undesired situation cannot be even more undesired, or a fault cannot be faultier. Prior to further input, the system must recover, i.e. the illegal event must be undone and the system must be conducted into a legal state through a backward or forward recovery mechanism. Please note further that also FIPs which include the entry symbol A, i.e. AD, ..., AF are also CFIPs as they represent executable, faulty interaction sequences of the length two.

Sub-Graph	FIPs
`Open File`	AD, AE, AF, AH, BA, BB, BH, DA, DB, EA, EB, FB, FF, HA, HB, HD, HE, HH

Fig. 4a: *The set of FIPs (Faulty Interaction Pairs)*

Sub-Graph	FCISs
`Open File`	AD, AE, AF, AH, ABA, ABB, ABH, ABDA, ABDB, ABEA, ABEB, ABFB, ABFF, AB(E+D)HA, AB(E+D)HB, AB(E+D)HD, AB(E+D)HE

Fig. 4b: *The set of FCISs (Faulty Complete Interaction Sequences)*

2.4 Test Procedure Using CISs and FCISs

The test process can be summarized now as follows:

1. Construct the complete set of test cases which includes all types of interaction sequences, i.e. all CISs and FCISs to produce the desired system responses and error messages, respectively (*Predictability* of the tests, defining oracles).
2. Input CISs and FCISs to transduce the system into a legal or illegal state, respectively (*Controllability*).
3. Observe the system output that enables a unique decision whether the output leads to a desired system response or an undesired, faulty event occurs which invokes an error message/warning, provided that a good exception handling mechanism has been materialized (*Observability*).

If the steps 1 to 3 can be carried out effectively, we have a *monitoring* capability of testing process that leads to a high grade of testability. Monitoring requires a special structure of software which must be designed carefully, considering the methods and principles of the modern Software Engineering (*Design for Testability*).

2.5 Handling Context Sensitivity

A problem we have to deal with during system modeling stems from the meaningful convenience of using the same commands, or icons for similar operations in different hierarchical levels of the application, e.g. `delete` for deleting a symbol, but also a record, or even a file. Upon the context information, the system can usually carry out the proper action. Our approach eliminates, however, the hierarchy information while abstracting the real system into the model (see the conversion of the Fig. 1 to Fig. 2).

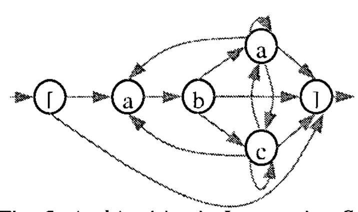


Fig. 5: *Ambiguities in Interaction Sequences*

Fig. 5 depicts an FSA that has two different states which will be initiated, i.e. triggered and identified by the same input `a`. While constructing the IPs and FIPs, and accordingly the CISs and FCISs, we have to differ between the state `a` that leads to `b` and the state `a` that can be reached by `b` and `c`. We have here an ambiguity that can be best resolved by indexing, i.e. `a1` for noting the first appearance of the `a`, and `a2` for the second one.

A good naming policy keeps the Hamming Distance of the identifiers of states with semantic and pragmatic similarity as small as possible, not only in order to enable a good proximity of the associated notions, but also the unambiguous distinction of the corresponding states of the FSA to avoid inconsistencies while producing the IPs, FIPs, CIS, and FCISs. As an example, we could name the operation "`delete`"

- `delete_c`, if we want to delete a character,
- `delete_w`, if we want to delete a word,

etc., or assign different, but associative icons to such operations.

3. Cost Aspects

As already mentioned repeatedly, one of the most difficult decision problems during testing is the determination of the time point when to stop testing. Since the early seventies of the last century, a variety of criteria to generate and to select test cases has been developed. Some of these criteria are formal, i.e. having a mathematical stringency, e.g. based on Predicate Logic ([12], see also Section 5, Related Work). The informal and semi-formal criteria introduce different *test coverage metrics*, e.g. to cover the structure of the software under test (SUT), or to cover its specification ([17, 20], see also Section 1, Introduction).

In our approach, we suggest to cover all combinations of edges which connect the nodes, i.e. to cover all of the IPs and FIPs.

3.1 Test Coverage of Interaction Sequences

With the definition of IPs (interaction pairs) and FIPs (faulty/illegal interaction pairs) that are minimal, i.e. of length two, sub-sequences of CISs (Complete Interaction Sequences) and FCISs (Faulty Complete Interaction Sequences), we have all elements we need for the optimization of the test process that must have monitoring capability:

- Cover all IPs of the CFSA (Complete Finite State Automata) by means of CISs.
- Cover all FIPs of the CFSA by means of FCISs.

Subject to

- Keep the number and total length of the CISs minimal.
- Keep the number and total length of the FCISs minimal.

In other words, we are seeking for a minimal set of CISs and FCISs to cover all prototypes of legal and illegal user-system interactions, revealing all appearances of system behavior, i.e. triggering all desired and undesired events. If we succeed this, we have a complete and minimal set of test cases to exercise the SUT. As we constructed the FSA according to the user expectations, the user himself, or herself acted as an Oracle at the most superior level. Thus, as test inputs we have CISs and FCISs; test outputs are desired and undesired events, as they will be determined during the construction of the FSA, resolving the Oracle Problem. Therefore, our approach delivers not only meaningful test cases, but it can also effectively select an optimal set of test cases to reach a well-defined coverage.

A more formal presentation and solution of the optimization problem is given in [4]. Following we informally summarize some of the results we recently achieved.

The set of CISs and FCISs as solution of these problems will be called *Minimal Spanning of Complete Interaction Sequences (MSCIS)* which can be constructed in two steps:

- *Legal Walks*: Construct CISs that traverse the FSA from entry to exit and contain all IPs, forming sequences of edges as *walks* through the FSA. An *entire walk* contains all IPs at least once. An entire walk is a *minimal walk* if its length cannot be reduced; an *ideal walk* contains all IPs exactly once.
- *Illegal Walks* are the FCISs, they do not necessarily start at the entry and end at the exit.

As demonstrated in the examples (Fig. 3 and 4), legal and illegal walks can be easily constructed for a given FSA. It is evident, that an entire walk exists only for legal walks. It is not, however, always possible to construct a minimal walk [4].

A similar problem is the Chinese Postman Problem [1] which has been studied thoroughly by A.V. Aho, T. Dahbura, Ü. Uyar et al., introducing the notion of "Multiple Unique Input Output Sequences" [1, 22]. Our MSCIS problem is expected to have less complexity, as the edges of the FSA are not weighted, i.e. the adjacent vertices are equidistant; therefore, we assume that the edges have all the length one. Further, we are not interested in tours, but walks through the graph, beginning in a start node (entry) and finishing in an end node (exit). Following, we include following some more results of [4] that are relevant to calculate the test costs and enable a scalability of the test process.

If the CFSA has n vertices, there are maximal n^2 edges (IPs and FIPs) that connect each of the n vertices with all of the other vertices. Assuming that FSA has d edges as legal IPs to present the desired CISs, exactly $u=n^2-d$ edges are illegal FIP. Thus, we can have at most u FCISs of minimal length, i.e. 2 (the entry input will be followed immediately by an illegal input); accordingly, the maximal length of an FCIS can be n (we have a CIS except the last input, i.e. the illegal input occurs just before and instead of the exit).

The minimal length of the CISs can be $n-1$ (inducing an ideal walk as a linear sequence); the maximum length of the CISs increases with n^2. The sum of the maximum lengths of CISs and FCISs increases also with the order n^2. We are working, however, on algorithms that are less costly, approximating to minimal walks [4].

3.2 Merging the States for Test Costs Reduction

Taking the "Divide and Conquer"-Principle into account, the test complexity can be reduced considerably if a sub-component of FSA (Finite State Automata) can be exercised separately; this component can than be reduced to a single vertex. We assume that we have sub-components $S1, S2, ..$ of the FSA. We assume further that these sub-components have a total number $s1, s2, ..$ of CISs and FCISs. If these sub-components $S1, S2, ..$ can be replaced by the vertices $N1, N2, ...$, the total number of test cases increases then additive, i.e. as $s1+s2+..$ instead of multiplicative, i.e. $s1*s2*..$ As L. White mentions in [28], following special structures of FSA enable such reductions.

Strongly Connected Components

D.P. Sieviorek et al. found out [25] that multiple exercising a strongly connected sub-graph, starting from different initial states, does not necessarily expose more faults (A graph is strongly connected if any node can be reached by any other one). In Fig. 6a, we have a strongly connected sub-graph consisting of the nodes `A, B, F`.

Structural Symmetric Components

Symmetric paths in a graph start and end at the same nodes, e.g. at the entry and exit and have the same sub-structure. The paths `ABEH` and `ABDH` of the sub-graph `open file` in Fig. 2 are symmetric (Fig. 6b).

C.N. Ip et al. [13] found out that multiple exercising symmetric components of a sub-graph, starting from different initial states, does not necessarily expose more faults.

These results are very important for our approach, because no matter what shape the initial FSA has, the completed CFSA is strongly connected, and has symmetric components, as it contains all edges (see Fig. 3). Thus, the completions through the initial FIP create sub-graphs that can be reduced to a single node. Clearly expressed, this means we can replace a sub-graph by a single node after we have tested this sub-graph thoroughly. The "inner life" of this node can be then neglected.

Fig. 6c demonstrates merging of states in sub-graphs that are strongly connected and have structural symmetric sub-components, leading to a considerably simpler graph.

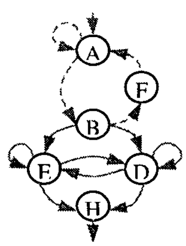


Fig. 6a: *Strongly Connected Sub-graph of* Fig. 2a

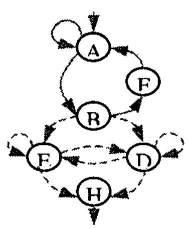


Fig. 6b: *Symmetric Sub-graphs of* Fig. 2a

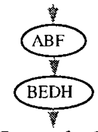


Fig. 6c: M*erging Strongly Connected Sub-graphs and Symmetric Sub-graphs of* Fig. 2a

3.3 Regular Expressions for Scaling and Scripting the Test Process

We already mentioned in the Sections 1 and 2 that a finite-state automaton FSA can be converted to an equivalent regular expression RegEx. Although the test case generation through FSA can be carried out efficiently, RegExs have some essential advantages over FSA concerning scalability. Once we construct the equivalent RegEx of an FSA, we can generate test case sets the cost of which can be determined exactly in terms of the length and number of test cases, as proposed by F. Belli, J. Dreyer and K.-E. Grosspietsch [3, 5]. In many cases the corresponding RegEx for an FSA can be constructed intuitively; efficient algorithms, e.g. developed by W.M. Gluschkow or A. Saloma [22], can be, however, executed automatically, as implemented by H. Troebner [27]. Having once converted the FSA into a RegEx, we can also use the Event Algebra [22], using well-known algorithms to reduce the complexity of the RegEx, keeping its generating capacity equivalent. The event algebra helps also to check similarities and equivalencies of RegExs.

Another advantage of operating with the regular expressions instead of its FSM is that the expression can be used as a *test script*, i.e. as test program that can be semi-automatically expanded by commercially available test tools, e.g. Visual State of IAR, or WinRunner of Mercury (Some test planning and coding effort is necessary). The scaling work can then be carried out by the tool; the test engineer has to specify solely the maximum length of the interaction sequences which are to be generated and exercised automatically according to the scripted test plan. Apart from test tools, also state-based design and specification tools, as to STATEMATE are potential candidates to deploy our approach for a flexible and effective fault handling.

4. Validation and Discussion of the Approach

The approach we described here has been used in different environments, i.e. we could extend and deepen our theoretical view interactively along practical insight during several applications. Instead of a full documentation which would run out space available in a brief report and the patience of the reader, we will in this section

- summarize an experiment to validate the approach,
- summarize the results of various applications, and
- discuss these results.

4.1 Validation

Mobile telephones will be widely used by a broad variety of types of end users. For the marketing success, the UIs of these devices become a decisive factor beside, perhaps before their size, weight and format. For our experiment, we chose the handling of Short Message Services which is very popular (Fig. 7 and 8). Table. 1 extracts some faults we could detect applying our approach. The STD of the underlying CFSA is partially given in the Appendix.

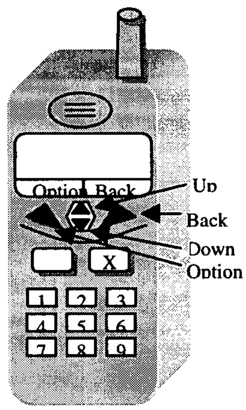


Fig. 7: *Prototype of Nokia Mobile Phone 6110*

1.	While writing and editing a short message (SM), menu can be resumed before the message has been saved.
2.	After sending an SM, the entire message must be deleted before an upper level menu can be reached.
3.	The access to the list of the received messages cannot be followed by a saving step of a selected message.
4.	Even the displayed message cannot be saved.
5	Without invoking the main menu, the received messages cannot be deleted selectively, or in groups.
6.	Fault #5 is valid also for SMs sent.
7.	Received messages cannot be forwarded.
8.	After deleting a message, the exit to the upper level menu follows only via the list of the SMs sent.
9.	Fault #1 is valid also for SMs that have been sent.

Table 1: *Excerpt from the Fault Analysis of Nokia 6610*

4.2 Results of the Experiments

For our validation experiments, we chose systems from a broad variety of applications to emphasize the versatility of the approach, among others:

- WordPad, already used in the present paper to depict examples.
- Vending Machine that supplies soft drinks.
- CD Player.

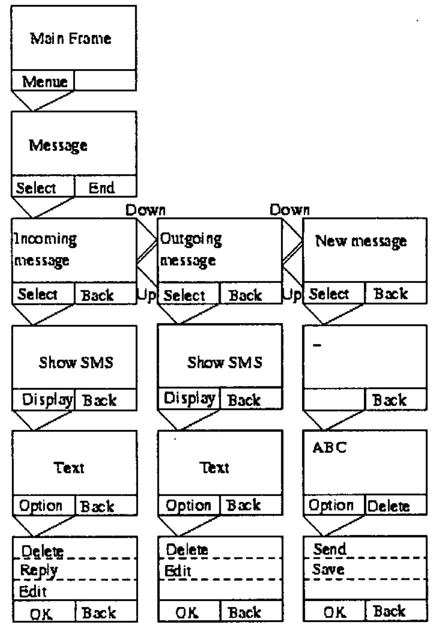


Fig. 8: *Short Messages Service (SMS)*

Table 2 summarizes our experiments, covering only the results of the top level of our models, including CISs and FCISs.

SUT	# Nodes	# FIPs	# Faults
WordPad	26	26	6
Mobile Phone	33	9	12
Vending Machine	18	39	5
CD Player	12	44	8

Table 2: *Numbers of nodes and FIPs vs. number of faults detected*

Please note that the systems we tested and analyzed are products that have been introduced long ago into the market and are being used for many years, thus having been steadily improved. In spite of their maturity, there is apparently still some more improvement potential. We could not, however, determine a direct correlation between the number of faults detected and number of FIPs that can be constructed.

4.3 Discussion of the Results

While some of the results of the fault analyses are in compliance with our expectations, some other results are surprising. Instead of listing long columns of statistical data, we summarize following directly the results of the analysis of these data.

- *Incomplete Exception Handling*: The initial concept for handling the undesired events, i.e. exceptions was in most cases strongly incomplete. The number of the exceptions could be increased in average about 70%. This result was expected: Our approach was originally founded to help the routinization of the exception handling.
- *Conceptual flaws*: Not as often as the forgotten undesired events, we found that also some major elements of the modeled system were missing, because the developer simply forgot them. In other words, the FSA was not lack of the edges, but vertices (Remember: vertices present inputs and states that merge). Thus, the initial concept of the developer(s) was seriously defect, having forgotten, or corrupted some vital components. The number of the vertices could be increased in average about 20%. This result was not expected: The approach helped to accelerate the conceptual maturation process considerably, supporting the creative mental activities.
- Another unexpected result was the *willingness of the end user to participate at the design process*. Even the user without any knowledge in Automata Theory and Formal Languages could understand the approach intuitively and very fast, especially the Transition Diagrams (They called them "Bubble Diagrams" which they could operate skillfully with). The participation of the user helped to complete the exception handling (they contributed to find about half of the forgotten exceptions), but also to detect the conceptual flaws (about 30% of them).

We recommend to use the approach incrementally, i.e. start very early, even with a rudimentary model of the system which should then be completed, adding the illegal connections to determine the faulty interaction pairs (FIP, see Section 2.2). The discussion of these FIPs is very often the most fruitful part of the modeling, leading to detect conceptual defects, and systematically completing the diagram not only by edges, but also by vertices. During this process, the test cases will be also systematically and scalable collected.

5. Related Work and Conclusion

FSA-based methods and RegEx have been used since almost four decades for specification and testing of software and system behavior, e.g. for Conformance Testing [6, 8]. Recently, L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [28]. Our work is intended to extend L. White's approach by taking not only desired behavior of the software into account, but also undesired 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.

Beside L. White's pioneer work, another state-oriented approach, based on the traditional method SCR (Software Cost Reduction) is described by C. Heitmeyer et al. in [11]. This approach uses model checking to generate test cases, using well-known coverage metrics for test case selection. For expressing conditioned events in temporal-logic formulae, the authors propose to use modal-logic abbreviations which requires some skill with this kind of formalism. A different approach for GUI testing has been recently published by A. Memon et al. [16], as already mentioned in Section 1. 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 C. Heitmeyer et al., 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. We showed that the approach of Dahbura, Aho et al. to handle the Chinese Postman Problem [1] in its original version might not be appropriate to handle GUI testing problems, because the complexity of our optimization problem is considerable lower, as summarized in Section 3.1. Thus, the results of our work enables efficient algorithms to generate and select test cases in sense of a meaningful criterion, i.e. edge coverage.

Converting the FSA into a RegEx enables us to work out the GUI testing problem more comfortable, applying algebraic methods instead of graphical operations. A similar approach was introduced 1979 by R. David and P. Thevenod-Fosse for generating test patterns for sequential circuits using regular expressions [9]. Regular expressions have been also proposed for software design and specification [24] which we strongly favor in our approach.

The introduced holistic approach, unifying the modeling of both the desired and undesired features of the system to be developed enables the adoption of the concept "Design for Testability" in software design; this concept was initially introduced in the seventies [29] for hardware. We hope that further research will enable the adoption of our approach in more recent modeling tools as to State Charts [2], UML [14], etc. There are, however, some severe theoretical barriers, necessitating further research to make the due extension of the algorithms we developed in the FSA/RegEx environment, mostly caused by the explosion of states when taking concurrency into account [23].

Acknowledgments

Several people have contributed important ideas to this work. I would like to thank Christof J. Budnik who worked out most of the experiments during his Master Thesis; my special thanks are also due to Walter Gutjahr (University of Vienna) and Andreas Ulrich (Siemens AG, Munich), and Stefan Gossens (Univ. of Erlangen-Nuremberg), and anonymous reviewers for their insightful technical comments.

References

[1] A. V. Aho, A. T. Dahbura, D. Lee and 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] D. Ahrel, A. Namaad, "The STATEMATE Semantics of Statecharts", *ACM Trans. Softw. Eng. Meth.* 5, pp. 293-333, 1996

[3] 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

[4] F. Belli, Ch. Budnik, "Minimal Spanning of Complete Interaction Sequences for GUI Testing", Technical Report 2001/3, Softw. Eng. FB14, Univ. Paderborn, 2001

[5] F. Belli, J. Dreyer, "Program Segmentation for Controlling Test Coverage", *Proc. 8th ISSRE*, pp. 72-83, 1997

[6] G. V. Bochmann, A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing", *Softw. Eng. Notes, ACM SIGSOFT*, pp. 109-124, 1994

[7] B. Boehm, "*Characteristics of Software Quality*", North Holland, 1981

[8] Tsun S. Chow, "Testing Software Designed Modeled by Finite-State Machines", *IEEE Trans. Softw. Eng.* 4, pp. 178-187, 1978

[9] R. David and P. Thevenod-Fosse, "Detecting Transition Sequences: Application to Random Testing of Sequential Circuits", in *Proc. Int. Symp. Fault-Tolerant Computing FTCS-9*, pp. 121-124, 1979,

[10] M.A. Friedman, J. Voas, "*Software Assessment*", John Wiley & Sons, New York, 1995

[11] A. Gargantini, C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specification", *Proc. ESEC/FSE '99*, ACM SIGSOFT, pp. 146-162, 1999

[12] D. Hamlet, "Foundation of Software Testing: Dependability Theory", *Proc. Of ISSTA '96*, pp. 84-91, 1994

[13] C. Noris, D. Dill, "Better Verification Through Symmetry", *Formal Methods in System Design*, Vol. 9, pp. 42-75, 1996

[14] 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

[15] B. Korel, "Automated Test Data Generation for Programs with Procedures", *Proc. ISSTA '96*, pp. 209-215, 1996

[16] 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

[17] J. Musa, "*Software Reliability Engineering – Faster Development and Testing*", McGraw-Hill, New York etc., 1999

[18] S. Naito, M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours", *Proc. FTCS*, pp. 238-243, 1981

[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] S. Rapps, E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Softw. Eng.*, pp. 367-375, 1985

[21] K. Sabnani and A. Dahbura, "A Protocol Test Generation Procedure", *Computer Networks and ISDN Systems 15*, North-Holland, pp. 285-297, 1998

[22] A. Salomaa, "*Theory of Automata*", Pergamon Press, New York, 1969

[23] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys* 22, pp. 299-319, 1990

[24] A.C. Shaw, "Software Specification Languages Based on Regular Expressions", in "*Software Development Tools*", ed. W.E. Riddle, R.E. Fairley, Springer, Berlin, pp. 148-176, 1980

[25] 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

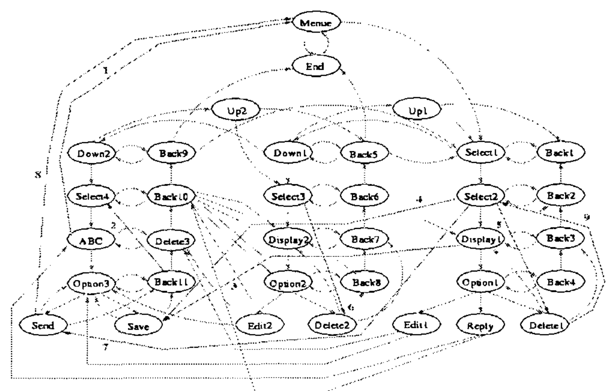
[26] B. Shneiderman, "*Designing the User Interface*", Addison Wesley Longman, 1998

[27] H. Troebner, "*Implementierung eines Verfahrens zur syntaktischen Behandlung der Kommunikationsfehler mittels regulärer Ausdrücke*", Master Thesis and Technical Report 1986/10, Hochschule Bremerhaven, FB 2, 1986

[28] 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

[29] T. W. Williams and K. P. Parker, "Design for Testability - A Survey", *IEEE Trans. Comp.* 31, pp. 2-15, 1982

Appendix



Partial CFSA of the SMS Handling of Nokia 6610 (Top Level)

(Legend: Solid lines: IPs; dashed lines: FIPs – with fault number refering to Table 1)