equipment [8]. This paper is about UI testing, i.e. testing of the software which materializes the UI. To some extent, also analysis aspects will be covered as testing and analysis usually belong together.

Based on finite-state automata (FSA) and regular expressions, the paper will introduce a holistic view: Fault modeling will be carried out as a complementary step to system modeling. Thus, both the desired and undesired behavior of the system will be specified at the same level of the design granularity. Appropriate formal notions will be used to introduce efficient algorithms to systematically generate and select test cases.

While constructing *test cases*, one generally has to produce meaningful *test inputs* and to determine then the expected system *outputs* for this inputs. To generate test cases for a UI, one has to identify first the test objects and the test objectives. *Test objects* are the instruments for the input, e.g. screens, windows, icons, menus, pointers, commands, function/alphanumerical keys, etc. The *test objective* is to generate the expected system behavior *(desired event*). Robust systems possess also a good exception handling mechanism, i.e. they are responsive by behaving good-natured in case of erroneous inputs of the user, generating constructive warnings, or tentative corrections, etc. that navigate the user into the right direction. In order to validate such robust system behavior, one needs systematically generated illegal inputs which entail injection of *undesired events* into the system under test (SUT).

Test inputs of GUI usually represent sequences of GUI objects activities and/or selections that will operate interactively with the objects (*Interaction Sequences – IS* [40], see also [20], *Event Sequences*). Such an interactive sequence is *complete (CIS)*, iff it eventually produces the desired system response. From Knowledge Engineering point of view, the testing of GUI represents a typical *planning* problem that can be solved goal-driven [23]: 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 testing 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*, [22]). 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* [16, 11]). 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, illustrating it with examples lent from real projects, e.g. electronic vending machines which accept electronic and hard money, performing transfers of electronic money to the owner's bank account, etc. The favored methods for modeling will concentrate on finite-state-based techniques, i.e. state transition diagrams and regular events. For the systematical, 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 a simplified interpretation of finite-state machines 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 summarized 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.

The approach was introduced first in [5]; the present paper deepens and extends this previous paper, including following discussions on:

- Usability and limitation of simplified FSA for UI modeling, based on [29].
- Applicability of the approach for other state-based techniques, e.g. state charts, state diagrams of the UML, etc.

Moreover, the previous experiments have been updated and extended, especially the experiment with the UI of the mobile phone Nokia 6110. A technical report with details of all these experiments is available [6].

2 Modeling From User's View: The Desired System Behavior and the Undesired One

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 descrip-

tion, 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.

2.1 Finite-State Modeling of UI

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 [27, 10], protocol conformance of open systems [7], compiler construction [1], but also for UI specification and testing [28, 40]. 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 corresponding regular expressions (RegEx), and v.v. [14, 32]. 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 [32]. 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 (A discussion of these assumptions will be given in the Sect. 4.3):

- 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 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. This simplification has been long ago introduced ("Myhill Graphs", [26]).

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, connected by operations

- *Catenation, or concatenation* (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^+*: at least one occurrence of a*).

Example: $T = (ab(a+c)^+)^*$ indicates that a will be followed by b leading to ab which be followed either by a or c; either one must occur at least once. The entire sequence can occur any number of times. Examples of the generated sequences are: aba, abc, abaaba, abaabc, …, but also λ (which symbolizes the empty word) for 0 (zero) occurrence.

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.

2.2 Terminology and an Elementary Example

Fig. 1 presents a small part of a MS WordPad-like word processing system (see also [22]). 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, interaction sequences (IS) can be generated.

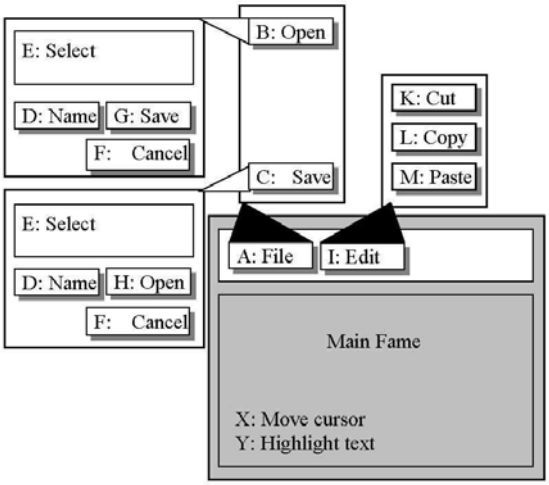

Fig. 1. Example of a GUI

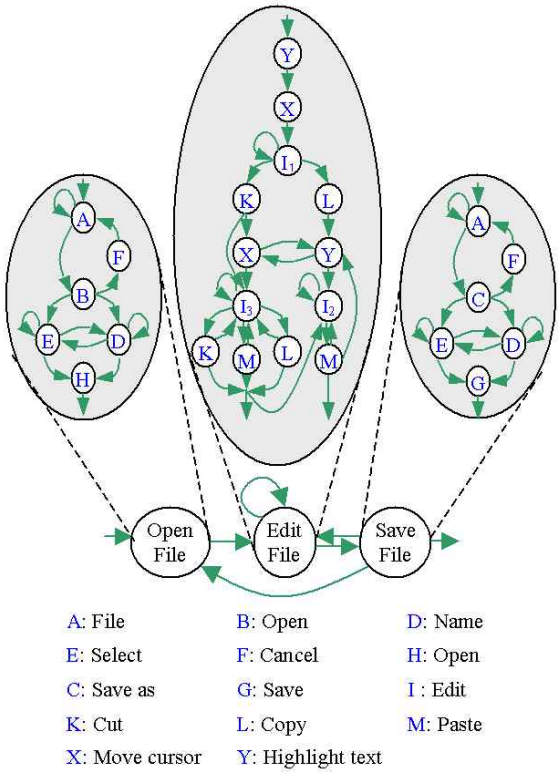

Fig. 2: Fig. 1 presented as a simplified Finite-State Machine

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 or canonical 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.

2.3 Interaction Sequences (IS) and Complete IS (CIS)

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 (Table 1, IPs based on the sub-graph in Fig. 2a). This is key issue of the present approach, as it enables us to define the *edge coverage* notion as a test termination criterion.

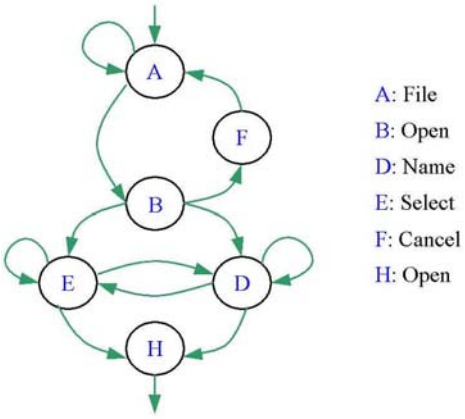

Fig. 2a: *Sub-Graph* open file *of the* Fig. 2

Table 1. *IPs of the Sub-Graph* open file *of the* Fig. 2

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

Table 2. . RegEx of the Sub-Graph open file of the Fig. 2

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

The generation of the CISs and IPs can be based either on the FSA, or more elegantly, on the corresponding RegEx [14, 32], whatever is more convenient for the test engineer (Table 2, 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 [7]. The systematic expansion of the RegEx, as we introduced in [3] is, however, relatively new to scalable generate test cases.

Please note that the Fig. 2a, Table 1 and Table 2 are equivalent; the different representations are supposed to help better understanding.

2.4 Complementing the CIS for a Complete Fault Modeling

The causes of faults in UI are mostly:

- The expected behavior of the system has been wrongly specified (*Specification Errors*).
- 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/In-compatible 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 manipulations:

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

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 Table 3). Once we have generated a FIP, we can extend it through an IS that starts with entry and ends with the first symbol of this FIP; we have than a *faulty/illegal complete interaction sequence* (*FCIS*), bringing the system into a faulty situation (Table 4). 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 also 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.

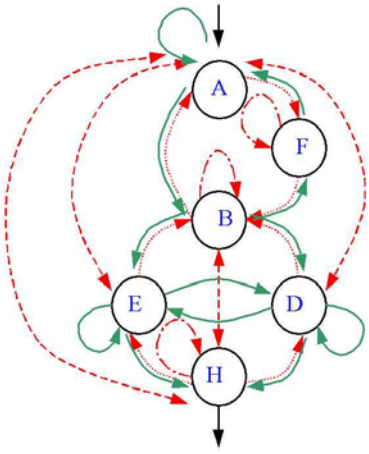

Fig. 3: CFSA (Completed FSA

Table 3. The set of FIPs (Faulty Interaction Pairs)

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

Table 4. The set of FCISs (Faulty Complete Interaction Sequences) which transduce the system into a faulty state

Sub-Graph	FCISs
File Open	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

3 Scaling and Optimizing the Test Process

3.1Defining and Scripting the Test Procedure

Using the notions we introduced, the test process can be summarized now as follow [41]:

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 an exception handling mechanism [15] has been materialized (*Observability*).

We already mentioned in the Sections 1 and 2 that a finite-state automaton FSA can be converted to a corresponding 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 corresponding RegEx of an FSA, we can use well-known algorithms to 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, 4].

In many cases, the corresponding RegEx for an FSA can be constructed intuitively; efficient algorithms, e.g. developed by W.M. Gluschkow [14] or A. Salomaa [32], can be, however, executed automatically, as implemented by H. Troebner [38]. Having once converted the FSA into a RegEx, we can also use the Event Algebra [32], 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 many 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.

3.2 Test Coverage and 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 [21]. 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 ([13]). The informal and semi-formal criteria introduce different *test coverage metrics*, e.g. to cover the structure of the SUT, or to cover its specification ([24, 25, 30], 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.

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. 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 contains 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 Table 3, Table 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.

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, 31]. 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 some

more results 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 Validation of the Approach

4.1 A Selected Experiment: UI of Nokia 6110

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. Following, we summarize our experiences with the approach; instead of a full documentation which would run out space available in a brief report and the patience of the reader, we rather display some spots, instantaneously enlightening some relevant aspects, focusing on the fault detection capabilities of the introduced method. We chose examples from a broad variety of applications to emphasize the versatility of the approach.

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. We chose the handling of Short Message Services which is very popular (Fig. 4 and 5). Table 5 extracts some faults we could detect applying our approach. The STD of the underlying CFSA is given in Fig. 6.

A comparison between the Nokia 6110 and the older model 5110 shows that both models have the same flaws, even if the 6110 has slightly different external features for handling.

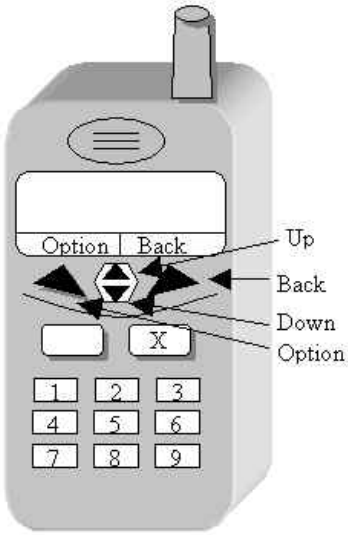

Fig. 4: Prototype of Nokia Mobile Phone 6110

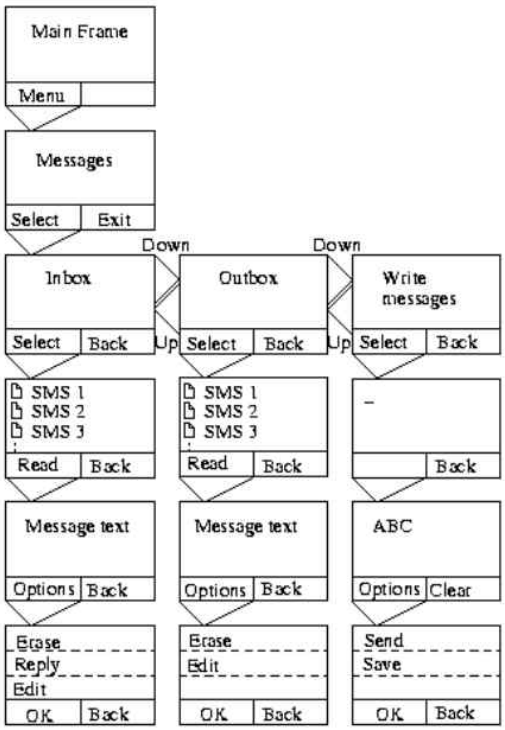

Fig. 5: Short Message Service Handling of the Nokia 6610

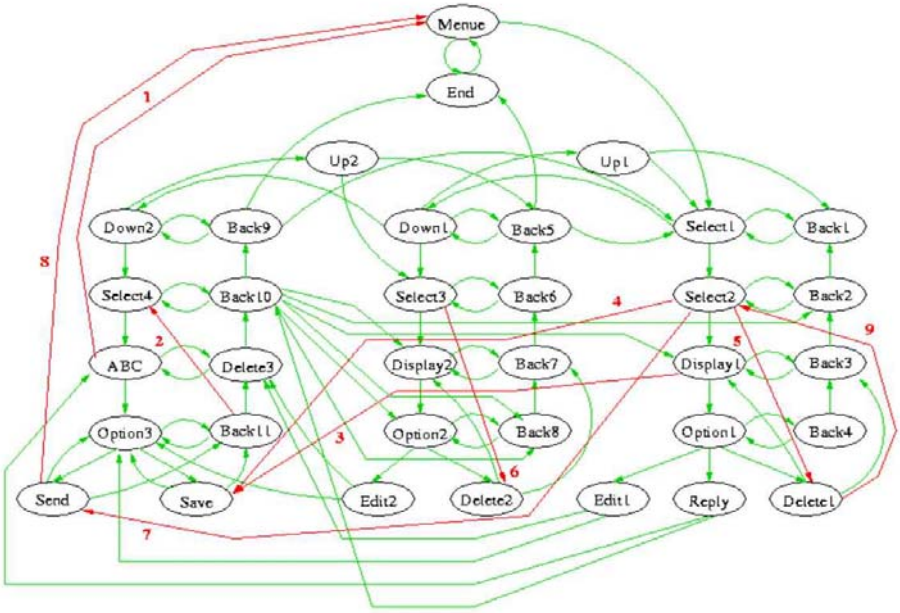

Fig. 6: *CFSA of the SMS Handling* (Legend: Numbers on lines (FIPs) refer to faults in Table 5; lines without numbers: IPs)

Table 5. Excerpt from Fault Analysis of the Nokia 6110

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

4.2 Results of the Experiments and Discussion

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

- WordPad, already described in Section 2,
- the mobile phone Nokia 6110, as described above,
- a commercial vending machine that supplies soft drinks,
- a commercial CD Player.

Table 6 summarizes our experiments, covering only the results of the top level of our models, including CISs and FCISs.

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.

Table 6. Numbers of nodes and FIPs vs. number of faults detected

SUT	# Nodes	# FIPs	# Faults
WordPad	26	26	6
Mobile Phone	33	127	9
Vending Machine	18	39	5
CD Player	12	44	8

While some of the results of the fault analysis 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 users to participate at the design process*. Even the users 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 users 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 2.3). 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 produced.

4.3 Usability and Limitation of the Approach

For modeling the UI via FSA, we merge the states and inputs/out-puts (Sect. 2.3). This assumption simplifies the complementing the STD considerably. Following example depicts this simplification (see Fig. 7a and 7b).

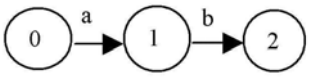

Fig. 7a: A Moore FSA

In Fig. 7a, the input a merges with the state 1; b with 2. State 0 becomes the entry. The result is given in Fig. 7b.

Fig. 7b: Simplified FSA

This simplification is appropriate if the inputs and outputs of the system are not substantially different, e.g. if an output can really act also as an input, or can be viewed as a part of the next input. This is true for GUI, as the expected system output usually emerges upon a sequence of inputs which invoke interim outputs as interim system reactions. The sequence of user-system interactions will be completed when the user expectation has been eventually fulfilled. The desired system response can act, in the course of a potentially endless user/system interaction process, as an input for the next interaction sequence.

Our simplification can cause some problems while modeling UI of reactive systems the inputs of which are substantially different than the outputs, e.g. a soft drink vending machine [39]. Here the sequence of inputs, e.g. inserting the coin, pressing a button, etc. can be modeled easily. The bottle with the drink that will be expected at the end is, however, substantially different as a "genuine" output that might not be viewed as an interim input – except the vending machine can accept this bottle immediately as an input to return the deposit on it. In such cases one needs some extra effort for modeling and finding errors [6].

The reason why the merging states with inputs/outputs is very important for the approach will be understood while complementing the STDs. During the simplified STD of the above example with two states a,b and no inputs (Fig. 7b) needs only three additional connections, the one with three states and an input/output alphabet of two symbols a,b (Fig. 7a) needs 16 additional edges (arcs connecting the vertices)! Fig. 8a and 8b demonstrate this; for the sake of simplicity, some arcs are bi-directional, or will be associated with both inputs a,b. Generally, a Moore FSA with n states and an input alphabet of the cardinality m has totally $m.n^2$ edges (Please note that this FSA is non-deterministic). The simplified one has only n^2 edges.

It is evident that a complete test procedure will not be always affordable; it makes also no sense to exercise all FIPs. Therefore, one need efficient heuristics for a non-perfect, but meaningful test selection strategy.

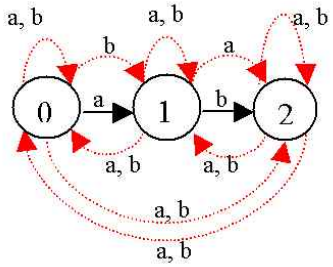

Fig. 8a: FCIS of Fig. 7a

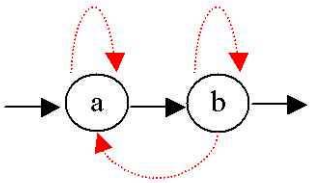

Fig. 8b: FCIS of Fig. 7b

The best way is to work with regular expressions. Based on well-known algorithms [14, 32] for conversion of regular expressions to FSA, [4] describes an efficient algorithm how to produce a simplified FSA for a given Moore-FSA. The resulting simplified FSA (states and inputs/outputs are merged) has more states than the original one, but is much more economic for a complete analysis as the examples above demonstrate clearly.

Another problem with the approach arises if the input cannot be determined completely, because some illegal interactions cannot be excluded, e.g. the likeliness that the user presses a button, or more severely, switches off the vending machine by mistake while it performs a task which was supposed to be atomic. This is, however, a problem of the analysis, and not the modeling tool. Commercial reactive systems in the practice are well-protected against such conditions by means of electronic-mechanical lock-in mechanisms.

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 [7, 9]. Recently, R.K. Sheady, D.P. Siewiorek [35] and L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [40]. 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.

Another state-oriented approach, based on the traditional method SCR (Software Cost Reduction) is described by C. Heitmeyer et al. in [12]. 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. [22, 23], 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, 36] 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 3.2. 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. R. David and P. Thevenod-Fosse for generating test patterns for sequential circuits using regular expressions [10]. Regular expressions have been also proposed for software design and specification [34] 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 [41] for hardware. We hope that further research will enable the

adoption of our approach in more recent modeling tools as to State Charts [2, 17, 18], UML [19], 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 additional vertices while completing the STD, and states when taking concurrency into account [33].

6 Acknowledgment

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).

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, J. Dreyer, "Program Segmentation for Controlling Test Coverage", *Proc. 8th ISSRE*, pp. 72-83, 1997
4. 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
5. F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces", *Proc. 12th ISSRE*, pp. 34-43, 2001
6. F. Belli, "Finite-State Testing and Analysis of User Interactions", Technical Report 2001/4, Univ. Paderborn, FB 14
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. B. Boehm, "Characteristics of Software Quality", North Holland, 1981
9. Tsun S. Chow, "Testing Software Designed Modeled by Finite-State Machines", *IEEE Trans. Softw. Eng.* 4, pp. 178-187, 1978
10. 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
11. M. A. Friedman, J. Voas, "*Software Assessment*", John Wiley & Sons, New York, 1995
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. S. Gerhart, J.B. Goodenough, "Toward a Theory of Test Data Selection", *IEEE Trans. On Softw. Eng.*, pp. 156-173, 1975
14. W.M. Gluschkow, "*Theorie der Abstrakten Automaten*", VEB Verlag der Wissensch., Berlin, 1963
15. J.B. Goodenough, "Exception Handling – Issues and a Proposed Notation", *Comm. ACM* 18/12, pp. 683-696, 1975
16. D. Hamlet, "Foundation of Software Testing: Dependability Theory", *Proc. Of ISSTA '96*, pp. 84-91, 1994

17. D. Harel, “Statecharts: A Visual Formalism for Complex Systems”, *Science of Comp. Programming* 8, 231-274, 1987
18. I. Horrocks, “*Constructing the User Interfaces with Statecharts*”, Addison-Wesley, MA, 1999
19. Y. G. Kim, H.S. Hong, D.H. Bae, S.D. Cha, “Test Case Generastion from UML State Diagrams”, IEEE Proc. Softw., Vol. 146, pp. 187-192, Aug. 1999
20. B. Korel, “Automated Test Data Generation for Programs with Procedures”, *Proc. ISSTA ’96*, pp. 209-215, 1996
21. B. Littlewood, D. Wright, “Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software”, Trans. Softw. Eng., 23/11, pp. 673-683, 1997
22. A. M. Memon, M. E. Pollack and M. L. Soffa, "Automated Test Oracles for GUIs“, *SIGSOFT 2000*, pp. 30-39, 2000
23. 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
24. E. Miller, “Program Testing – An Overview”, *Infotech State of the Art Report on Softw. Testing* 2, Maindenhead, 1979
25. J. Musa, “Software Reliability Engineering – Faster Development and Testing”, McGraw-Hill, New York etc., 1999
26. J. Myhill, “*Finite Automata and the Representation of Events*”, Wright Air Devel. Command, TR 57-624, pp. 112-137, 1957
27. S. Naito, M. Tsunoyama, “Fault Detection for Sequential Machines by Transition Tours”, *Proc. FTCS*, pp. 238-243, 1981
28. 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
29. A. Petrenko, Private Communication, 2001
30. S. Rapps, E.J. Weyuker, “Selecting Software Test Data Using Data Flow Information”, *IEEE Trans. Softw. Eng.*, pp. 367-375, 1985
31. K. Sabnani and A. Dahbura, "A Protocol Test Generation Procedure", *Computer Networks and ISDN Systems 15*, North-Holland, pp. 285-297, 1998
32. A. Salomaa, “Two Complete Axiom Systems for the Algebra of Regular Events ”, *J. ACM* 13, pp. 158-169, 1966
33. F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys* 22, pp. 299-319, 1990
34. 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
35. R. K. Sheady 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
36. Y.-N. Shenn, F. Lombardi and A.T. Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences", *IEEE Trans. Commun*. 40, pp. 1282-1287, 1992
37. B. Schneiderman, “*Designing the User Interface*”, Addison Wesley Longman, 1998
38. H. Troebner, “*Implementierung eines Verfications zur syntaktischen Behandlung der Kommunikationsfehler mittels regulärer Ausdrücke*”, Master Thesis and Technical Report 1986/10, Hochschule Bremerhaven, FB 2, 1986
39. A. Ulrich, Private Communication, 2001
40. 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
41. T. W. Williams and K. P. Parker, "Design for Testability - A Survey", *IEEE Trans. Comp*. 31, pp. 2-15, 1982