

# Finite-State Modeling, Analysis and Testing of System Vulnerabilities

Fevzi Belli, Christof J. Budnik

Dept. of Computer Science,  
Electrical Engineering and Mathematics  
University of Paderborn  
Warburger Str. 100  
D-33098 Paderborn  
belli@upb.de  
budnik@adt.upb.de

Nimal Nissanke

School of Computing Science,  
Electrical Engineering and Mathematics  
South Bank University  
103 Borough Road  
UK-London SE1 0AA  
nissanke@sbu.ac.uk

**Abstract:** Man-machine systems have several *desirable* properties, as to user friendliness, reliability, safety, security or other global system attributes. The potential for the lack, or breaches, of any such property constitutes a system vulnerability, which can lead to a situation that is *undesirable* from user's point of view. This undesired situation could be triggered by special events in the form of intended, or unintended, attacks from the system's environment. We view the undesirable system features as the sum of the situations, which are, mathematically speaking, complementary to the desirable ones that must be taken into account from the very beginning of the system development to achieve a stable system behavior and a robust operation. This work is about the modeling, analysis and testing of both desirable and undesirable system features which can be viewed as relations between the system and its operation.

## 1. Critical Features and Vulnerabilities of Embedded Systems

In terms of behavioral patterns, the relationships between the system and its environment, i.e., the user, the natural environment, etc., can be described as proactive, reactive or interactive. In the case of *proactivity*, the system generates the stimuli, which evoke the activity of its environment. In the case of *reactivity*, the system behavior evolves through its responses to stimuli generated by its environment. Most computer-human interfaces are nowadays *interactive* in the sense that the user and the system can be both pro- and reactive. In this particular kind of system, the user interface (UI) can have another environment, that is, the plant, the device or the equipment, which the UI is intended for and embedded in, controlling technical processes.

When categorizing behavioral aspects of the human-computer systems, it is a good principle to start with the *ergonomic* aspects which concern the extent of comfort, or discomfort, experienced by the system and its environment at each other's company. Any vulnerability is always accompanied by *threats*.

In the case of *safety*, the threat originates from within the system due to potential failures and its spill-over effects causing potentially extensive damage to its environment. In the face of such failures, the environment could be a helpless, passive victim. The goal of the system design in this case is to prevent faults that could potentially lead to such failures or, in worst cases, to mitigate the consequences at run-time should such failures ever occur.

In the case of *security*, the system is exposed to external threats originating from the environment, causing losses to the proprietor of the system. In this case, the environment, typically the user, maybe unauthorized, can be malicious or deliberately aggressive. The goal of the system design is then to ensure that it can protect itself against such malicious attacks.

In contrast to safety and security, the lack of *user friendliness* of a UI is a milder form of system vulnerability. A failure to be user-friendly is still a failure to fulfill a system attribute, though it may typically cause an annoyance or an irritant to the user, possibly leading to confusion and complaints. However, disastrous consequences cannot be entirely ruled out under stressed conditions of the user if the UI is intended to mediate between a safety critical application and its operator. Since safety is being considered here as a system vulnerability in its own right, we limit ourselves here to the UIs of modest applications designed to guide and help the user to navigate through, and explore, the system functionality.

Having identified several differences and distinctions between certain system vulnerabilities, let us abstract away from these peculiarities and threat them all in a generic manner in the following section.

Further sections will illustrate our approach through a case study considering user interactions as a special case.

An earlier version of the approach was introduced in [BE01]. The present study generalizes the approach and extends its scope of application.

## 2. Finite State Modeling of System Vulnerabilities

### 2.1 Modeling the System and Its Environment

The functionality  $f$  of an application is described here as a deterministic finite-state automaton (FSA)

$$M_f = (S, E, \alpha, f) \quad (1)$$

- where
- $S$  is a set of states,
  - $E$  is a subset of  $S \times S$ ,
  - $\alpha$  is a set of input signals, and
  - $f$  is a total function from  $E$  to  $\alpha$ .

The vertices of the corresponding transition diagrams represent the *states* (as the elements of  $\mathbf{S}$ ); the directed arcs (the elements of  $\mathbf{E}$ ) represent the *state transitions* which are labeled by *input symbols* (the elements of  $\alpha$ ). This notation is slightly different than the common ones used in the literature. We focus here on the state transitions caused by inputs and determined by  $f$  defining, in conjunction with  $E$ , the *next state function*.

$E$  has a distinguished unlabelled element denoting the *entry* from the exterior (*environment*) of  $M$  and another distinguished unlabelled element denoting the exit to the exterior of  $M$ .  $\mathbf{S}$  and  $\mathbf{E}$  are such that the field of  $f$  equals  $\mathbf{S}$ .

The detailed structure of  $M$  in (1) can be suppressed by using a transition diagram of inputs described in terms of a *grammar*  $G$  equivalent to  $M$  based solely on  $\alpha$ , in terms of  $L(M)=L(G)$ , where  $G$  can be represented by a *regular expression*  $R$ . Thus, we can merge inputs and states, leading to considerably more efficient algorithms for analysis and test. This view allow us to focus on input sequences, generated as strings of  $L(G)$ , or  $L(R)$ , instead of input-state sequences, as is necessary in the case of Moore automata. A similar interpretation has been introduced by Myhill [My57] long ago.

## 2.2 Strings to Model the Behavioral Patterns of the System – Analysis and Test Aspects

System functions, as well as the threats to a chosen system vulnerability attribute, may each be described using two disjoint subsets of strings,

- one belonging to the language  $L(M)$  and
- another one not belonging to  $L(M)$ , respectively.

*Legal* state transitions represent desirable events, leading to input sequences belonging to  $L(M)$ , producing system functions; *illegal* transitions represent the undesirable events, leading to input sequences not belonging to  $L(M)$ , causing breaches to vulnerabilities.

Let us denote the

- *system functions* as  $F$  and
- the *vulnerability threats* as  $V$ .

Depending on the chosen system vulnerability attribute, the strings corresponding to vulnerability threats can be grouped in accordance with their length  $n$ . This assumes that all threats can be unequivocally identified by patterns of  $n$  consecutive symbols, i.e. strings, of  $\alpha$ . It is obvious that we can utilize the grammar  $G$ , or its regular expression  $R$ , to generate *test cases* to verify whether

- system functions are fulfilled, and/or
- vulnerability threats occur.

It is important to note that several vulnerability attributes simultaneously apply to a given application. In this case,  $F$  will remain the same in the study of every vulnerability attribute, but  $V$  will vary from one vulnerability attribute to another. To avoid mismatching, the relevant threats to each attribute  $att$  will be identified as  $V_{att}$ .

### 2.3 Ordering the Vulnerability Risk, Countermeasures

The threats constitute only a part of the specification of system vulnerability. The remainder consists of a vulnerability risk ordering relation, or briefly, a *risk ordering relation*  $\Xi$  on  $S \times S$ ; see [ND02]. Given the states  $s_1, s_2 \in S$ , the risk ordering  $\Xi$  is defined such that  $s_1 \Xi s_2$  is true if and only if the risk level of  $s_1$  to the chosen system vulnerability is known to be less than, or equal to, the risk level of  $s_2$ . In other words, risk level quantifies the *degree of the unacceptability* of an event, on the grounds of hazardousness, exposure to breaches of security, the lack of usability or user-friendliness and so forth.

The risk ordering relation  $\Xi$  is intended as a guide to decision making upon the detection of a threat, whether internal or external, and on how to react to it. The required response to breaches of vulnerability needs to be specified in terms of a *defense matrix*  $D$ , which is a partial function from  $S \times V$  to  $S$ . The defense matrix utilizes the risk ordering relation to revert the system state from its current one to a less, or the least, risky state. The actual definition of this matrix is the responsibility of a domain expert specializing the risks to a given vulnerability. Reversion of the system state from its current one to a less risky state is to be implemented in the form of *exception handlers*  $X$ , which can be defined as

$$X \in S \times V \rightarrow S \quad (2)$$

such that

$$\forall s_1, s_2, v \bullet (s_1, v) \in \text{dom } X \wedge X(s_1, v) = s_2 \Rightarrow s_2 \Xi s_1 \quad (3)$$

Finally, let us define the *model of an application defended* against vulnerabilities as

$$M_d = (S, E, \alpha, f; G, F, V, \Xi, X) \quad (4)$$

with  $S, E, \alpha, f; G, F, V, \Xi, X$  are as defined above.

The following sections are intended at refining the above model, making it more precise and relating it to practically relevant real-life scenarios.

Objectives of our research is to demonstrate the above ideas using very different key application areas such as user friendliness, safety and security, by adhering to the same principles and, broadly speaking, striving to achieve similar goals. However, this paper presents our results on modeling, analysis and test of user interactions only.

## 2.4 Event Sequence Graphs to Specify the System Functions

FSA (finite-state automata) will be conveniently represented by their state transition diagrams (STD). 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” interchangeably). The edges connect the adjacent nodes and will be labeled by inputs and outputs, in accordance with the next-state function.



Fig. 1: State Transition Diagram (STD) of a Finite-State Automata (FSA)

For representing UI, we will interpret the elements of the input set of the FSA as the operations on identifiable objects that can be perceived and controlled by input/output devices, i.e., elements of WIMPs (Windows, Icons, Menus, and Pointers). Any chain of edges from one vertex of the STD to another, realized by sequences of “initial user inputs – (interim) system responses – (interim) user inputs – ... – (final) system response” defines an *interaction sequence (IS)*. Note that there could also be no interim system responses so that IS can consist of only user inputs except the final system response.

For the user, the system response can be an expected, desirable event. Undesirable events are responses which the user does not want, i.e., a faulty result, or an unexpected result that surprises the user. An IS is *complete* if the (final) response is a desirable event producing a *complete IS (CIS)* ([WA00]). Thus, the set of the CISs specifies the system function  $F$  as introduced in Section 2.2; in other words, the CISs are the legal words of the language defined by the FSA that models the system.

Following this interpretation, we merge the inputs and states, assigning them to the vertices of the STD. The next-state function will be interpreted accordingly, thus inducing the next input that will be merged with the due state. This is in accordance with the conventional interpretation of finite automata as *acceptors* of input sequences, having a single initial state and final (accepting/rejecting) state(s) [Gi62, Sa69]. In other words, we abstract from states and concentrate on inputs. The result is a simplified version of the STD that we call an *Event Sequence Graph (ESG)*. ESG are digraphs and can be compared with the Myhill graphs [My57] which will also be used as computation schemes [Ia58], or as *syntax diagrams*, e.g., to define the syntax of Pascal [Je85, Te02]. The difference between the Myhill graphs and our ESG is that the symbols which label the nodes of an ESG will be interpreted not as symbols and meta-symbols of a language, but as operations of an event set. Fig. 2 represents the ESG of the FSA given in Fig. 1.

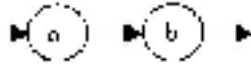


Fig. 2: Event sequence graph of the FSA given in Fig. 1

ESGs have the same computational power as the FSA (*type-3* grammars, producing *regular* languages), having *recognizing* capabilities that we need, e.g., to answer the essential question “Is a (final) event desirable?”, which can be reduced to the *word recognition* problem that is decidable for type-3 grammars.

To sum up, we use the notions “state” and “input” on one hand, and “state” and “output” (as “system response”) on the other hand, synonymously, because the user is interested in the external behavior of the system, and not its internal states and mechanisms. Thus, we are strongly focusing on the needs and expectations of the user.



Fig. 3: Example of a GUI

### 3. Modeling User Interactions

The approach we described here has been used in different environments, including industrial applications. Thus, we could extend and deepen our theoretical view and gain practical insight by examining several applications. Below, we summarize our experiences with the approach. In stead of a full documentation which would exceed the space available in a brief report and exhaust the patience of the reader, we will display some highlights, instantaneously clarifying some relevant aspects and focusing on the fault detection capabilities of the introduced method. We chose this example from a broad variety of applications to emphasize the versatility of the approach.

### 3.1 Refining the Terminology and an Introductory Example

Fig. 3 represents the main menu of the RealJukebox of the RealNetworks. At the top level, the GUI has a pull-down menu with the options *File*, *Edit*, etc., that invoke other components. These sub-options have further options and sub-options, etc. There are still more window components which can be used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

The event sequence graph (ESG) in Fig. 4 presents at the top level the GUI to produce desired event “Play and Record a CD or Track” via the main menu given in Fig. 3. Each of these desired events defines a *system responsibility* which will be refined in further levels. Again, the terms event, state, and situation will be used here interchangeably.

Each of the nodes of Fig. 4 represents inputs which interact with the system, leading eventually to events as system responses that are desirable situations in compliance with the user’s expectation. Based on the sub-graphs, due interaction sequences (IS) can be generated. Traversing the ESG from the entry to the exit produces a *complete walk* through the ESG which can be seen as a complete IS (CIS) as defined in the section 3.1.



Fig. 4: The responsibility Play and Record a CD or Track represented as an ESG

Fig. 3 is easy to understand, though it is an informal and imprecise presentation of the GUI, while Fig. 4 is a formal presentation that neglects some aspects, e.g. the hierarchy, while still being precise. The conversion of Fig. 3 into Fig. 4 is the most abstract step in our approach that must be done manually, requiring some practical experience and theoretical skill in designing GUIs. As is common in modeling processes, we choose and name such events in the diagram that seem to us most relevant, attempting to adopt the user’s view of the picture; there is no algorithmic or canonic way to abstract the relevant part from the entire environment. The majority of the task that follows the modeling can, however, be carried out automatically, according to algorithms described in [Be01] and summarized in this paper.

The determination of the CISs should ideally be carried out during the definition of the user requirements, long before the system is implemented. They are then a part of the systems specification and test specification. CISs can also be incrementally produced, however, at any later time, even during the test stage, in order to discipline the test process.

It cannot be emphasized strongly enough that what we are doing here contributes to an elegant solution of the Oracle Problem: Identification of the Complete Interaction Sequences (CIS) does present the meaningful, expected system outputs which will be constructed here systematically. What we need is the construction and analysis of the test inputs in terms of sequences of user interactions.

### 3.2 Interaction Pairs as Atomic Elements of System Function

Once the FSA has been constructed, more information can be gained by means of its ESG. First, we can now identify all legal sequences of user-system interactions which may be complete or incomplete, depending on whether they lead to a well-defined system response in accordance with the user's expectation. Second, we can identify the entire set of the *compatible*, i.e., legal *interaction pairs (IP)* of inputs as the edges of the ESG. This is a key feature of the present approach, as it will enable us to introduce and optimize the *edge coverage* as a test termination criterion ([Be01]).

As an example, Fig. 5.1 lists all IPs of Fig. 4.

LS, LR, SP, SM, SR, PS, PP, PR, PM, MP, MS, MM, MR, RL, RM

Fig. 5.1: Interaction Pairs (IPs) of Fig. 4

As a next step of the test case construction, we generate CISs as test inputs. These are sequences of UIs which start at the entry and reach the exit of the ESG, traversing this graph, edge by edge, or IP by IP. Fig. 5.2 gives some CIS for the example sketched in Fig. 4. Note that these CISs are supposed to transfer the system into a final, desirable state which fulfills the user's expectations. Thus, we now have complete test cases consisting of CISs as inputs and the predefined desirable events are the expected outputs.

LSR, LR, LSPR, LSMR, LSR, LSPSR, LSPPR, LSPR, LSPMR, LSMPR, LSMSR,  
LSMMR, LSMR, LRLR, LRMR

Fig. 5.2: Some Complete Interaction Sequences (CISs) as test inputs of Fig. 4

The generation of the CISs and IPs can be based either on the ESG or on the corresponding regular expression [Gl63, Sa66], whichever is more convenient for the system analyst and/or the test engineer. The information given in figures 5.1 and 5.2 are actually included in Fig. 4.

Note that the IPs represent the shortest (*atomic*) interactions, they are also shortest sub-sequences of CISs. Generally spoken, we call the sub-sequences of the complete interaction sequences *incomplete* or *partial* interaction sequences – *PIS*. Thus, PISs of length  $n$  define *n-tupels of interaction*.

### 3.3 Faulty Interaction Pairs to Represent the Vulnerability Threats

The faults in UIs arise mostly from the following:

- 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 *user errors*, i.e., we suggest that there are no user errors. In other words, we suggest that the user is *always right*. 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 the desirable situation.

One consequence of this requirement (*Input Completeness*, as mentioned in the Introduction) is that we need a view which is complementary to the modeling of the system. This will be done by systematic and stepwise manipulation of the ESG 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 the sense of the specification. For our example *Play* and *Record* a CD or *Track* given in Fig. 4, Fig. 6 generates all FIPs by threefold manipulations, producing the *completed ESG (CESG)*:

- Add edges in opposite direction wherever only one-way edges exist (*reversing*, dotted connections, (1) in Fig. 6).
- Add iterations to vertices wherever none exist in the specification (*looping*, dashed-dotted connections, (2) in Fig. 6).
- Add two-way edges between vertices wherever no edges exist (*networking*, dashed connections, (3) in Fig. 6).

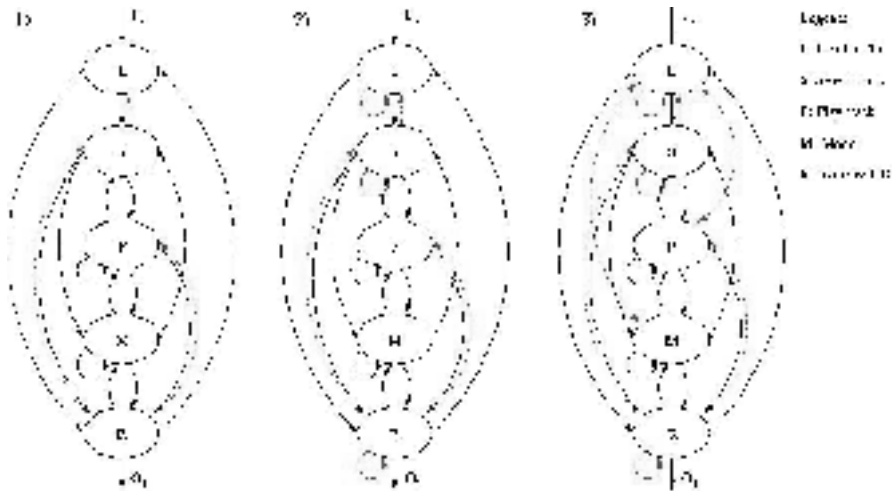


Fig. 6: CESG (Completed ESG) of Fig. 4

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. 6.1). Our objective is to generate test cases to check the system behavior in case of faulty inputs sequences, i.e., undesirable events. For this purpose, we use the FIPs to construct test inputs twofold:

- FIPs that start at the entry are complete test inputs to trigger undesirable situations.
- FIPs which do not start at the entry are not executable. To exercise them, corresponding ISs that start at the entry and end at the first symbol of such FIPs will be generated as their prefixes. These prefixes will be called *starters*. Thus, we construct executable extensions of FIPs to, and transfer the system into, undesirable situations.

LL, LP, LM, SL, SS, PL, ML, RP, RR, RS
--

Fig. 6.1: The set of FIPs (Faulty Interaction Pairs) of Fig. 4

LL, LP, LM, LSL, LSS, LSPL, LSML, LRP, LRR, LRS
---

Fig. 6.2: The set of FCISs (Faulty Complete Interaction Sequences) of Fig. 4

We then have *faulty* (or *illegal*), *complete interaction sequences (FCIS)* of different length, which are executable. In our example (Fig. 6.2), FIPs which start with the entry symbol L, i.e., LL, LP, LM are complete and do not need any prefixes as they are executable; in other words, they are FCIS of length two. They conduct the system into a faulty situation. Note that the attribute “complete” within the phrase FCIS expresses only the fact that the FIP might have been “completed” by means of an IS as a prefix to make this FIP executable (otherwise it is not complete, i.e., not executable). Thus, a *faulty complete interaction pair (FCIP)* can be used as a test sequence which will conduct the system into an undesirable, faulty situation with the goal of revealing and activating its exception handling mechanism (*Fault Injection*).

### 3.4 Risk Ordering, Exception Handling and Fault Injection

FCISs are user interaction sequences which start at the entry of the ESG (like the CISs), but will not reach its exit (unlike the CISs): Once the system is in a faulty state, it cannot accept any further legal or illegal inputs, because an undesirable situation can neither be conducted to a desirable one, nor can it be transferred into an even more undesirable one (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 [Go75, Ra78].

The construction of the FCISs guarantee that only the last two symbols (as an FIP) of them are incompatible, i.e., for the determination of the position in which a correction can take place we need a backtracking by at most two symbols. The costs of a correction depends also on the number of the different symbols which can transfer the system into a correct state. As an example, we say the FIP LL of Fig. 6.1 brings the system into a less risky position than the FIP SL because

- LL can be transferred to the only two IPs LS and LR, backtracking by and changing of only one symbol,

- while SL can be transferred to SP, SM and SR

when backtracking by only one symbol.

The risk ordering relation can be determined by using the conventional parsing algorithms known in compiler construction.

In most cases a forward recovery might be more convenient, e.g. alerting immediately the user when detected an FIP.

The FCISs listed in Fig. 6.2 are supposed to transfer the system into a faulty, undesirable situation. Thus, we have complete test cases consisting of FCISs as inputs; expected outputs are by definition always undesirable situations. Additionally, we need a criterion to judge the efficiency of the test cases that will determine when to stop testing. This problem of *Minimal Coverage of the User Interaction* can be transferred to the coverage of the edges and has been handled in [Be01]; in this paper we concentrated on generalization of the approach with the goal to apply it to any critical systems feature, as to safety and/or security.

#### 4. Related Work and Conclusion

Since E.W. Dijkstra's criticism "Program testing can be used to show the presence of bugs, but never to show their absence" [Di72], we in some part of the academic community have almost a religious belief that testing should never be used as a verification method [Di81, BS81]. Nevertheless, testing is the most accepted and widely used method in industrial software development, not only for verification of the software correctness, but also its validation, especially concerning the user requirements (See [Bo81] for the precise definitions of the terms "Verification" and "Validation"). Taking this fact into account, many researchers in software engineering started very early to form a mathematically sound foundation for a systematic testing. One of the pioneer works that made software testing become a solid discipline is the "Fundamental Test Theorem" of S. Gerhart and J.B. Goodenough which was published in 1975: "We prove that properly structured tests are capable of demonstrating the absence of errors in a program" [GG75]. By means of their test selection criteria, based on predicate logic, Gerhart and Goodenough found numerous major errors in a program of P. Naur which was published in textbook and repeatedly quoted by other renowned authors, also used in an example for correctness proof. Since the Fundamental Test Theorem, a vast amount of further research work enabled *systematic testing* to become more and more recognized and also accepted in academia, e.g. through the work of E.W. Howden: "It is possible to use testing to formally prove the correctness of programs" [Ho78]. Also worth mentioning is the work of L. Bouge, who made valuable contributions to the Software Test Theory [Bo85].

As a result of these efforts, we now have a glimmer of hope that systematic testing will be accepted as a sound method by the computer science community (see also [Pe01], Chapters 9 and 10). However, *systematic testing* still has no entry in the ACM CR Categories and testing will be used in there interchangeably with debugging, thus in the sense of a “trial and error” method.

FSA-based methods and regular expression have been used for almost four decades for specification and testing of software and system behavior, e.g. for Conformance Testing [BP94, Ch78]. Recently, R.K. Shedy, D.P. Siewiorek [SS97] and L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [WA00]. Based on [EB84, BG91], our work is intended to extend and refine L. White’s approach by taking not only desired behavior, but also undesired situations, of the software into account. 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 also exercising all potentially illegal events to verify that the GUI behaves satisfactory 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 cumulative costs that must be in compliance with the test budget. This strategy is quite different from the combinatorial ones, e.g. *pairwise testing*, which requires that for each pair of input parameters of a system, every combination of these parameters’ valid values must be covered by at least one test case. This is, in most practical cases, not feasible [Co97, TL02].

Another state-oriented approach, based on the traditional SCR (Software Cost Reduction) method, is described by C. Heitmeyer et al. in [GH99]. This approach uses model checking (which can identify negative and positive scenarios) to generate test cases automatically from formal requirements specifications, using well-known coverage metrics for test case selection. A different approach for GUI testing has been recently published by A. Memon et al. [MPS00, MPS01]. The authors deploy methods of knowledge engineering to generate test cases, test oracles, etc., and to handle the Test Termination Problem. Both approaches, i.e. those 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 has been deemed appropriate in the practice of protocol validation for decades. The advantage of our approach stems from its simplicity that enables a broad acceptance in practice. We showed that the approach of Dahbura, Aho et al. to handle the Chinese Postman Problem [AhDa, SLD92] in its original version might not be appropriate to handle GUI testing problems, because the complexity of our optimization problem is considerably lower, as summarized in Section 4.2. Thus, the results of our work enable efficient algorithms to generate and select test cases in the sense of a meaningful criterion, i.e. edge coverage.

Converting the ESG into a regular expression enables us to work out the GUI testing problem more comfortably, 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 [DT79]. Regular expressions have been also proposed for software design and specification [Sh80, RS94] 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 1970s for hardware [WP82]. We hope that further research will enable the adoption of our approach in more recent modeling tools such as State Charts [AN96, Ho99], UML [Ki99, BL02], as well as Safecharts [DN99]. There are, however, some severe theoretical barriers, necessitating further research to make the due extension of the algorithms we developed in the FSA/regular expression environment, mostly caused by the explosion of additional vertices and states when taking concurrency into account [Sc90].

To sum up, in this work, we viewed the undesirable system features as the sum of the situations, which are complementary to the desirable ones that must be taken into account from the very beginning of the system development to achieve a stable system behavior and a robust operation. As an example, we analyzed user interactions that are central part of any system.

Our further work concentrates on safety and security aspects, considering also costs and their optimization subject to the mutual coverage of several critical features.

## Literaturverzeichnis

- [AN96] D. Ahrel, A. Namaad, “The STATEMATE Semantics of Statecharts”, *ACM Trans. Softw. Eng. Meth.* 5, pp. 293-333, 1996
- [Be01] F. Belli, “Finite-State Testing and Analysis of Graphical User Interfaces”, *Proc. 12<sup>th</sup> ISSRE*, pp. 34-43, 2001
- [BG91] 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
- [BL02] L. Briand, Y. Labiche, “A UML-Based Approach to System Testing”, *Software and System Modeling* 1, pp.10-42, 2002
- [Bo81] B. Boehm, “*Characteristics of Software Quality*”, North Holland, 1981
- [Bo85] L. Bouge, “A Contribution to the Theory of Program Testing”, *Theoretical Computer Science*, pp. 151-181. 1985
- [BP94] G. V. Bochmann, A. Petrenko, “Protocol Testing: Review of Methods and Relevance for Software Testing”, *Softw. Eng. Notes, ACM SIGSOFT*, pp. 109-124, 1994

- [BS81] R.S. Boyer, J. Strother-Moore (eds.), “*The Correctness Problem in Computer Science*”, Academic Press, London, 1981
- [Ch78] Tsun S. Chow, “Testing Software Designed Modeled by Finite-State Machines”, *IEEE Trans. Softw. Eng.* 4, pp. 178-187, 1978
- [Co97] D.M. Cohen, et al., “The AETG System: An Approach to Testing Based on Combinatorial Design“, *IEEE Trans. Software Eng.* 23/7, pp. 437-443, 1997
- [Di72] E.W. Dijkstra, “Notes on Structured Programming”, in O.J. Dahl et al. (ed.), “*Structured Programming*”, Academic Press, London, pp. 1-82, 1972
- [Di81] E.W. Dijkstra, “Why Correctness Must Be a Mathematical Concern?”, in [BS81], 1981
- [DN99] H. Dammag, N. Nissanke, „Safecharts for Specifying and Designing Safety Critical Systems”, *Proc. 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne., 1999
- [DT79] 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
- [EB84] B. Eggers, F. Belli, “A Theory on Analysis and Construction of Fault-Tolerant Systems” (in German), *Informatik-Fachberichte 84*, pp. 139-149, 1984
- [GG75] S. Gerhart, J.B. Goodenough, “Toward A Theory of Test Data Selection”, *IEEE Trans. On Softw. Eng.*, pp. 156-173, 1975
- [GH99] A. Gargantini, C. Heitmeyer, “Using Model Checking To Generate Tests from Requirements Specification”, *Proc. ESEC/FSE '99, ACM SIGSOFT*, pp. 146-162, 1999
- [Gi62] A. Gill, „*Introduction to the Theory of Finite-State Machines*“, McGraw-Hill, 1962
- [Gl63] W.M. Gluschkow, “*Theorie der Abstrakten Automaten*”, VEB Verlag der Wissensch., Berlin, 1963
- [Go75] J.B. Goodenough, “Exception Handling – Issues and a Proposed Notation”, *Comm. ACM* 18/12, pp. 683-696, 1975
- [Ho78] W.E. Howden, “Theoretical and Empirical Studies of Program Testing”, *IEEE Trans. Softw. Eng.*, pp. 293-298, 1978
- [Ho99] I. Horrocks, “*Constructing the User Interface with Statecharts*”, Addison-Wesley, MA, 1999
- [Ia58] Ianow, J.I., “Logic Schemes of Algorithms”, *Problems of Cybernetics (in Russian) I*, pp. 87-144, 1958
- [Je85] Jensen, K., Wirth, N., „Pascal User Manual and Report“, Springer, 1985
- [Ki99] 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, 1999
- [MPS00] A. M. Memon, M. E. Pollack and M. L. Soffa, "Automated Test Oracles for GUIs“, *SIGSOFT 2000*, pp. 30-39, 2000

- [MPS01] 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
- [My57] Myhill, J., "Finite Automata and the Representation of Events", Wright Air Devel. Command, TR 57-624, pp. 112-137, 1957
- [ND02] N. Nissanke, H. Dammag, „Design for Safety in Safecharts With Risk Ordering of States”, *Safety Science* Vol. 40, pp. 753-763, 2002
- [Pe01] D.A. Peled, "Software Reliability Methods", Texts in Computer Science, Springer, 2001
- [Ra78] B. Randell, "Reliability Issues in Computing System Design", *ACM Comp. Surveys* 10/2, pp. 123-165, 1978
- [Sa66] A. Salomaa, "Two Complete Axiom Systems for the Algebra of Regular Events", *J. ACM* 13, pp. 158-169, 1966
- [Sa69] A. Salomaa, "Theory of Automata ", Pergamon Press, Oxford, etc., 1969
- [Sc90] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys* 22, pp. 299-319, 1990
- [Sh80] 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
- [SLD92] Y.-N. Shen, F. Lombardi and A.T. Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences", *IEEE Trans. Commun.* 40, pp. 1282-1287, 1992
- [SS97] 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
- [RS94] S.C.V. Raju, A. Shaw, "A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines", *Software – Practice and Experience* 24/2, pp. 175-195, 1994
- [Te02] R.D. Tennent, *Specifying Software*, Cambridge University Press, Cambridge, UK, 2002
- [TL02] K. Tai, Y. Lei, "A Test Generation Strategy for Pairwise Testing", *IEEE Trans. On Softw. Eng.* 28/1, pp. 109-111, 2002
- [WA00] 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
- [WP82] T. W. Williams and K. P. Parker, "Design for Testability - A Survey", *IEEE Trans. Comp.* 31, pp. 2-15, 1982