This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Linschulte, M., On 'Negative' Tests of Web Applications, Annals of Mathematics, Computing & Teleinformatics, Vol. 1, No. 5, pp. 44-56

On "Negative" Tests of Web Applications

Fevzi Belli, Michael Linschulte
University of Paderborn
Warburger Str. 100, 33098 Paderborn, Germany
{belli,linschu}@uni-paderborn.de

***Abstract*—Web applications are of decisive importance in e-commerce. Due to the heterogeneous nature and different quality criteria of system environment, its components and user expectations, new demands emerge for testing of those systems to ensure a high reliability level. This paper introduces a novel approach to testing the functionality of web applications. For this purpose the system under test (SUT) is modeled by structured event sequence graphs (sESG) combined with decision tables. Walks through the ESG form "complete" event sequences and thus define tests. The test process is completed by augmenting the model(s) given for testing the SUT behavior in unexpected, undesirable situations; this will be called "negative testing". Accordingly, "positive tests" check the conformity of SUT behavior with the expected one under regular circumstances. A test algorithm is presented for minimizing the test effort in execution of both positive and negative tests. Commercially available test tools help with reducing the manual work and costs.**

***Index Terms*—testing, testingvalidation, web applications.**

I. Introduction and Related Work

Because of the rising e-commerce demand the number of web applications increased enormously over the past years. Due to the fact that web applications are essential for these business processes and their success, the quality of such systems is crucial. The question is: What are appropriate quality criteria with respect to web applications? A first hint gives the international standard ISO/IEC 9126 that tries to define general quality criteria that are applicable to any kind of software. It classifies software quality in functionality, reliability, usability, efficiency, maintainability and portability. But the standard seems not to be without problems, e.g., it is criticized by Jung et al. [1]. However, the discussion of those quality criteria shall not be the central issue of this paper. This is a much wider issue than the testing aspects discussed and covered here. Much more of interest is to emphasize quality criteria that are applicable to web applications in order to draft key aspects in testing these applications and the position of the presented approach. Therefore some special characteristics should be mentioned: First, web applications allow an open usage to any user. Second, the running environment is very heterogeneous due to different browsers, web-server and operating systems. Furthermore, their components can be implemented by different technologies on different platforms, e.g., ASP, Java or PHP (see also [2]). Last but not least, large amounts of data can be collected by web-based forms; a process, which can even cause dynamic structural changes within the application. According to these special characteristics, quality criteria like usability, performance and reliability, interoperability, safety, topicality and accessibility should be emphasized (see also [3]).

This paper focuses on testing the functionality and therefore the usability of web applications. Generally speaking, web applications belong to the class of interactive systems. Modeling and testing of interactive systems by means of state-based models has a long tradition [4], [5]. These approaches analyze the system under test (SUT) and model the user requirements to achieve sequences of user interaction (UI) which form test cases. In [6] a simplified state-based, graphical model to represent UIs is introduced; this model has been extended in [7] to consider not only desirable situations, but also the undesirable ones.
Most of the models to design and test web applications use graphical means, e.g., extensions of UML to describe web applications [8], [9]. In [10], an UML-model to analyze and test a web application is used, too. It is based on a meta-model which describes the concepts of a web application. According to this, the model of a certain application is an instance of this meta-model. In [11], a method is introduced for test case generation starting with an object-oriented model. The SUT is analyzed from three perspectives, concerning its object(s), structure and behavior. These perspectives are modeled by different diagrams. Based on these diagrams test trees are created which lead to test cases. The approach introduced in [12] addresses black-box-testing of web applications. It considers the state-based behavior of web applications and proposes a specific solution for the state-space-explosion-problem by refinement of nodes with additional models.
This work is on testing the functionality of web ap-

plications. SUT and its graphical user interface (GUI) are modeled by event sequence graphs (ESG), complete paths of which form test cases; a concept first introduced in [7]. However, differing from this concept, the model developed in this paper enables refining and structuring of the model by decision tables. For testing the undesirable behavior ("negative" testing) the starting model for testing the desirable behavior ("positive" testing) is augmented. Thus, introducing decision tables to ESG makes new considerations about augmenting ESGs necessary. Moreover, a new algorithm is given for minimizing the effort of both, positive and negative tests. Finally, a detailed fault classification is achieved by an extension of the fault model given in [7].

The next section summarizes the theoretical background of ESGs. Section 3 extends the notion introduced to enable modeling and testing of web applications and presents the novel algorithm for minimizing the test effort. Furthermore, a fault model for a comprehensive fault classification to analyze web applications is introduced. As such applications are too costly to manually be tested, Section 4 demonstrates how a commercially available tool can be used for test automation. Section 5 sketches and evaluates the approach by means of an example drawn from a large commercial web portal. This case study reveals also the characteristic factors of the approach when applied to complex web applications. Section 6 summarizes the ongoing and future work and concludes the paper.

II. Theoretical Background

As already mentioned, this work uses ESG notion to represent the system behavior from the user's point of view [13]. Basically, an event is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response, punctuating different stages of the system activity.

A. Event Sequence Graphs

Definition 1: An event sequence graph $ESG = (V, E)$ is a directed graph where $V \neq \emptyset$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

$\Xi(ESG), \Gamma(ESG)$ represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex $[\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex $] \notin V$.

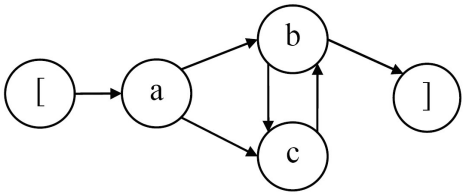

Fig. 1. An ESG with *a* as entry and *b* as exit and pseudo vertices [,]

The semantics of an ESG is as follows: Any $v \in V$ represents an event. For two events $v, v' \in V$, the event v' must be enabled after the execution of v if and only if $(v, v') \in E$. The operations on identifiable components of the GUI are controlled and/or perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of V and lead interactively to a succession of user inputs and expected desirable system outputs.

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

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

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

Furthermore, α(initial) and ω(end) are functions to determine the initial vertex and end vertex of an ES, e.g., for $ES = \langle v_0, \ldots, v_k \rangle$, initial vertex and end vertex are $\alpha(ES) = v_0, \omega(ES) = v_k$, respectively. For a vertex $v \in V, N^+(v)$ denotes the set of all *successors* of v, and $N^-(v)$ denotes the set of all *predecessors* of v. Note that $N^-(v)$ is empty for an entry $\xi \in \Xi$, and $N^+(v)$ is empty for an exit $\gamma \in \Gamma$.

Finally, the function $l(length)$ of an ES determines the number of its vertices. In particular, if $l(ES) = 1$ then $ES = v_i$ is an ES of length 1.

Note that the pseudo vertices [and] are not considered in generating any ESs. Neither are they considered to determine the initial vertex, end vertex, and length of the ESs.

Example 2: For the ESG given in Fig. 1, ***bcbc*** is an ES of length 4 with the initial vertex ***b*** and end vertex ***c***.

Definition 3: An ES is *complete* (or, it is called a *complete event sequence, CES*), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\omega(ES) = \gamma \in \Gamma$ is an exit.

Example 3: ***acb*** is a CES of the ESG given in Fig. 1. CESs represent walks from the entry of the ESG to its exit realized by the form *(initial) user inputs →(interim) system responses → ... →(final) system response*. Note that a CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs and a final system response.

Definition 4: Given an ESG, say $ESG_1 = (V_1, E_1)$, a vertex $v \in V_1$, and an ESG, say $ESG_2 = (V_2, E_2)$. Then replacing v by ESG_2 produces a refinement of ESG_1, say $ESG_3 = (V_3, E_3)$ with $V_3 = V_1 \cup V_2 \backslash v$, and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \backslash E_{1replaced}$ ('\': set difference operation), wherein $E_{pre} = N^-(v) \times \Xi(ESG_2)$ (connections of the predecessors of v with the entry nodes of ESG_2), $E_{post} = \Gamma(ESG_2) \times N^+(v)$ (connections of exit nodes of ESG_2 with the successors of v), and $E_{1replaced} = \{(v_i, v), (v, v_k)\}$ with $v_i \in N^-(v)$, $v_k \in N^+(v)$ and where $(v_i, v), (v, v_k) \in E_1$(replaced arcs of ESG_1).

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

B. Complementary View

The approach assumes that there is no user error, i.e., upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the anticipated desirable situation. Due to this requirement, a complementary view is necessary to consider potential user errors in the modeling of the system (see also [17], [18]).

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

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

Fig. 3 illustrates $\widehat{ESG}$, which can systematically be constructed in three steps:

- Add arcs in the opposite direction wherever only one-way arcs exist.
- Add self-loops to vertices wherever none exist.
- Add two-way arcs between vertices wherever no arcs connect them. Note that they are drawn bi-directional.

$\overline{ESG}$ (the inversion of the ESG) consists of arcs that will be added to the ESG to construct the $\widehat{ESG}$ (completion of the ESG).

Definition 7: Any EP of the $\overline{ESG}$ is a *faulty event pair (FEP)* for ESG.

Definition 8: Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length k + 1 of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0, \ldots, v_k, v_m \rangle$.

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

Example 4: For the ESG given in Fig. 3, the FEP ***ca*** of the $\overline{ESG}$ can be completed to the FCES ***acbca*** by using the ES ***acbc*** as a starter. Note that the [is not included in the FCES as it is a pseudo vertex.

Note that Definition 9 explicitly points out that a FCES does not necessarily finish at an exit, unlike a CES that must finish at an exit. The starter ***acbc*** in Example 4 is arbitrarily chosen, and hence the variation in length of an FCES is always attributable to starters prior to this special FEP under consideration. The result is then FCESs of various lengths. Thus, the "length" in the test process primarily relates to the CESs.

III. Modeling Web Application Interactions by Means of ESG

The ESG model presented in Section 2 was successfully applied to interactive systems [13]. As observed in industrial applications, ESG that model input data and their dependencies get rapidly large. This chapter extends the ESG notion by decision tables for efficiently modeling input data. The interpretation of FEP, that contain decision tables, is also discussed. Additionally, a modified test process algorithm and fault model is presented.

A. Modeling User Inputs with Decision Tables

Usually, data is input to a web application via web-based forms. Such forms consist of *text boxes*, *check boxes*, *radio buttons* and *select boxes*. Text boxes allow the input of any string whereas check boxes, radio buttons and select boxes provide a set of given values. Furthermore, radio buttons allow the selection of exactly one value contrary to check boxes that afford selection of multiple values. Select boxes allow both according to their 'multiple' attribute. Modeling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model. To avoid this, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled.

Definition 10: A Decision Table $DT = \{C, A, R\}$ represents actions that depend on certain constraints where:

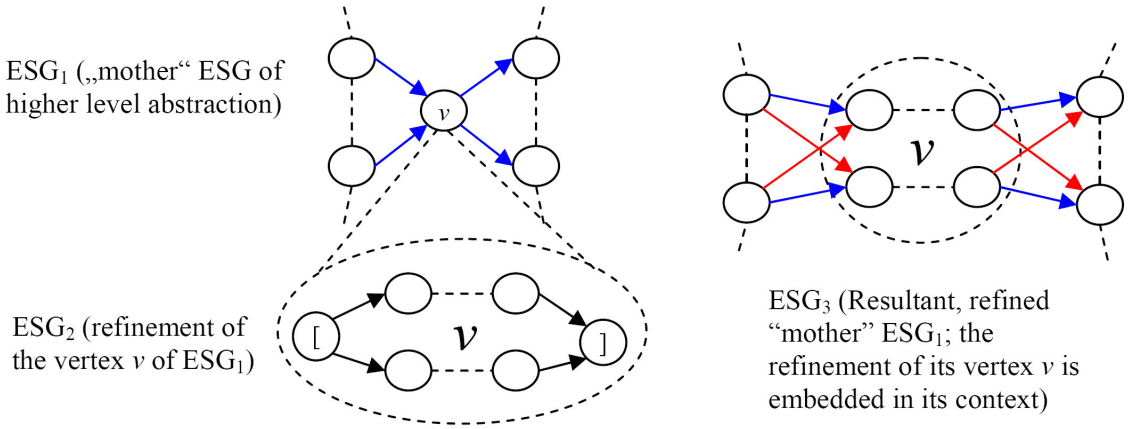

Fig. 2. Refinement of a vertex v and its embedding in the refined ESG

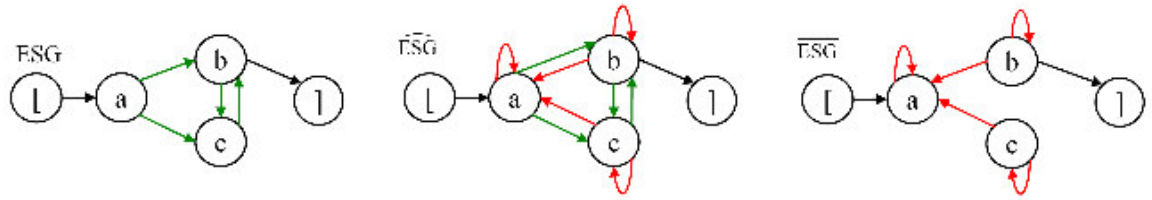

Fig. 3. ESG of Fig. 1, its completion $\widehat{ESG}$ and inversion $\overline{ESG}$ with $\overline{ESG} = \widehat{ESG} \backslash ESG$

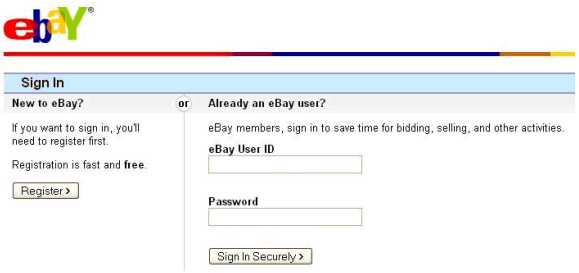

Fig. 4. Input form of eBay

- $C \neq \emptyset$ is the set of constraints
- $A \neq \emptyset$ is the set of actions
- $R \neq \emptyset$ is the set of rules that describe executable actions depending on a certain configuration of constraints

Decision tables [19] are popular in information processing and are also used for testing, e.g., in cause and effect graphs [15], [16]. A decision table logically links conditions ("if") with actions ("then") that are to be triggered, depending on combinations of conditions ("rules").

According to web applications, the set of constraints C is the union of CF and CC. Constraints CF describe the domain D of input fields F. Each constraint of the set CC involves some subset of input fields F and specifies the allowable combinations of values for that subset or additional domain restrictions.

Definition 11: Let R be defined as in definition 1. Then a *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, A_x)$ where:

- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to true
- $C_{False} = C \backslash C_{True}$ is the set of constraints, that have to be resolved to false
- $A_x \subseteq A$ is the set of actions that should be executable if all constraints $t \in C_{True}$ are resolved to true and all constraints $f \in C_{False}$ are resolved to false

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$.

Example 5: For the Decision Table given in Table I: C = {Constraint 1, Constraint 2}, A = {Action 1, Action 2} and R = {R1, R2, R3, R4, R5} with

R1: $C_{True} = \{Constraint1, Constraint2\}, C_{False} = \emptyset, A_x = \{Action1, Action2\}$

R2: $C_{True} = \{Constraint1\}, C_{False} = \{Constraint2\}, A_x = \{Action1\}$

R3: $C_{True} = \{Constraint2\}, C_{False} = \{Constraint1\}, A_x = \{Action2\}$

R4: $C_{True} = \{Constraint1\}, C_{False} =$

TABLE I
A DECISION TABLE WITH 5 RULES, 2 CONSTRAINTS AND 2 ACTIONS

Decision Table	R1	R2	R3	R4	R5
Constraint 1	T	T	F	T	F
Constraint 2	T	F	T	F	T
Action 1	X	X			
Action 2	X		X	X	X

TABLE II
EXAMPLE OF A COMPLETE, REDUNDANCE-FREE AND CONSISTENT DECISION TABLE

Decision Table	R1	R2	R3	R4
Constraint 1	T	T	F	F
Constraint 2	T	F	T	F
Action 1	X	X		
Action 2	X		X	X

$\{Constraint2\}, A_x = \{Action2\}$
R5: $C_{True} = \{Constraint2\}, C_{False} = \{Constraint1\}, A_x = \{Action2\}$

Table I depicts some combinations of constraints and actions. Nevertheless it is not complete because no rule exists, where both constraints are set to false.

Definition 12: A Decision Table $DT = (C, A, R)$ is *complete*, if $P(C) \subseteq S$ with $S = \{x|(x, y, z) \in R\}$ and $P(C) = \{M|M \subseteq C\}$. $P(C)$ is also announced as *Power Set* of C.

Moreover the Decision Table in Table I is not redundance-free as R3 and R5 describe the same combination of conditions and actions.

Definition 13: A Decision Table $DT = (C, A, R)$ is *redundance-free*, if $S = \emptyset$ with $S = \{x|(x, y, z) \in R \wedge (a, b, c) \in R \backslash (x, y, z) \wedge x = a \wedge z = c\}$.

Rule 2 and Rule 4 of Table I describe different possible actions at the same condition combination. Thus, the decision table is also not consistent.

Definition 14: A Decision Table $DT = (C, A, R)$ is *consistent*, if $S = \emptyset$ with $S = \{x|(x, y, z) \in R \wedge (a, b, c) \in R \wedge x = a \wedge z \neq c\}$.

The conclusion that follows from Definition 3 to Definition 5 is that a Decision Table $DT = (C, A, R)$ is *complete*, *redundance-free* and *consistent*, if $P(C) = S$ with $S = \{x|(x, y, z) \in R\}$. Such a decision table has always $2^{|C|}$ rules according to $P(C)$ where $|P(C)| = 2^{|C|}$. A decision table that meets all these criteria can be seen in Table II.

Definition 15: Given a $DT = (C, A, R)$ and two rules $r1 = (C_{True1}, C_{False1}, A_{x1})$ and $r2 = (C_{True2}, C_{False2}, A_{x2})$ with $r1, r2 \in R$. DT can be *consolidated*, if $A_{x1} = A_{x2} \wedge \exists c \in C_{True1} : (C_{True1} \backslash c) = C_{True2} \wedge (C_{False2} \backslash c) \in C_{False1}$.

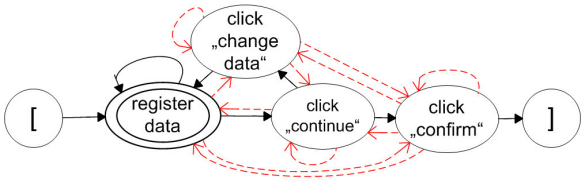

Fig. 6. Completed ESG of Fig. 5

The new decision table is denoted as $DT_{new} = (C, A, R_{new})$ with $R_{new} = (R \backslash \{r1, r2\}) \cup r3$ where $r_3 = (C_{True1} \backslash c, C_{False1}, A_{x1})$. r_3 is also called as *consolidated rule*.

Note that C_{True} conjoint with C_{False} of a consolidated rule now does not equal the set C. The constraints $C^* = C \backslash (C_{True} \cup C_{False})$ are remarked with the *don't care* term '-'. Thus, a constraint that holds a *don't care* can be resolved to true *and* false.

Definition 16: An event $v \in V$ of an ESG is called *data event (DE)* if v is refined by a decision table.

Fig. 5 models the processing of an eBay login form as given in Fig. 4. Table III structures the corresponding decision process as a decision table. The node *login data* of Fig. 5 is refined by the decision table in Table III. *eBay User ID* and *Password* are used here several times as the consecutive events depend on certain values.

To sum up, an ESG consists of nodes that represent either events or other ESGs or decision tables. An ESG visualizes sequences of events and therefore allows detection of discrepancies in the sequential execution of user-system-interaction. Decision tables augment the ESG given to support analyzing of causal dependencies of events. Now walks through the ESG can be generated to produce CESs (see Section 3.4) and the SUT can be trained using them.

B. Faulty Event Pairs in Conjunction with Decision Tables

Testing the desirable behavior of a system is often not sufficient, because faults also arise upon incorrect user operations. We suggest using decision tables to construct such situations. As seen in Table III, the rules which lead to errors in the action part are identified by the sign 'e'. Exercising these rules form *negative tests*. For constructing undesirable behavior of SUT among events, the underlying ESG model is to be completed as described in section 2.2.

Fig. 6 completes the sub-ESG given in Fig. 5. Note that the pseudo nodes [and] are excluded in complementing the given ESG. FEPs (dashed edges) are tested by FCES which start with an entry node and end with a FEP (see Definitions 7 and 9). The test is *passed*, if the final event cannot be reached, otherwise the test

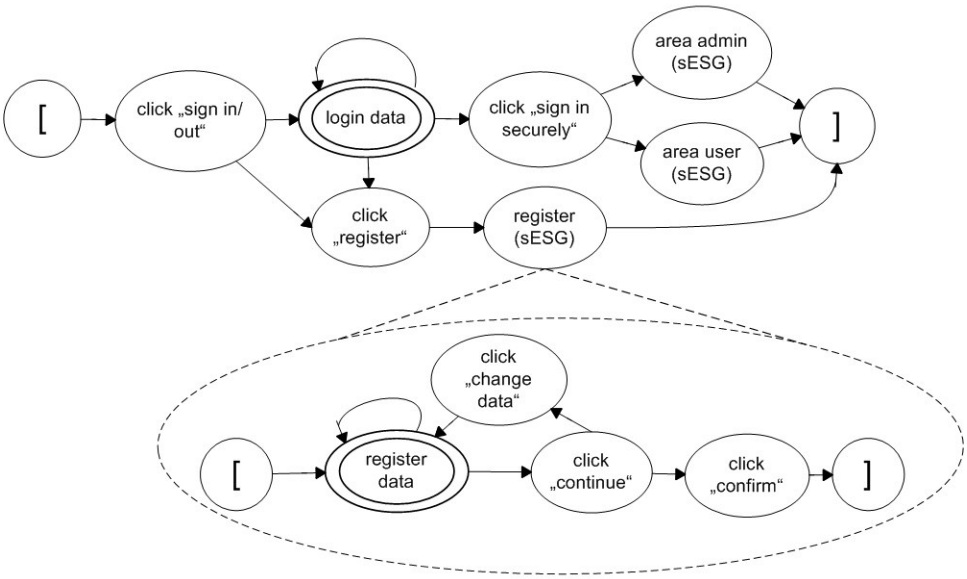

Fig. 5. sESG model for processing the input form given in Fig. 4. Double-circled nodes are refined by decision tables.

TABLE III
DECISION TABLE TO FIG. 4

DT - login data	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**
eBay User ID $\in \sum^*$	T	T	T	T	F	F
Password $\in \sum^*$	T	T	T	F	T	F
$eBayUserID \wedge Password \in admin - login$	T	F	F	-	-	-
$eBayUserID \wedge Password \in user - login$	F	T	F	-	-	-
click "Sign In Securely", area admin	X	e	e	e	e	e
click "Sign In Securely", area user	e	X	e	e	e	e

Legend: 'X': possible actions;'e': error; '-': Don't Care-Term; '?': input alphabet; conditions not in italics: co-domain of input data; conditions in italics: dependencies among input data and/or necessary values related to consecutive actions.

has *failed*. The replacement of decision tables by input data within FEP has to be considered separately. Faulty self loops (Fig. 7) cannot occur if the input form does not change on input of data without pressing a submit-button. In this case the user can correct data without initiating changes on the input form. If there are changes after input of data and without pressing a submit-button, an input to the same input fields should not be possible. It is then sufficient to generate data according to any rule of the decision table, independent of the action following the one under consideration. A test according to faulty self-loops passes, if the input of data is partly or completely not feasible. Faulty outgoing edges (Fig. 8) imply that the following event should not be feasible. Generating data should be performed for every rule of the decision table as the execution of a following event can depend on the selected input data.

In case of faulty incoming edges (Fig. 9), the data input itself should not be possible and generation of

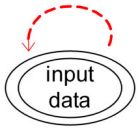

Fig. 7. Faulty self-loop

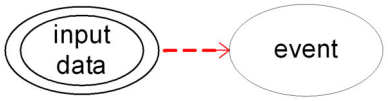

Fig. 8. Faulty outgoing edge

Fig. 9. Faulty incoming edge

data for any rule of the decision table is sufficient (again independent of the following action). Similar to a faulty self-loop, a test passes if the complete data input is not possible.
In practice, the total amount of effort for testing the negative behavior of SUT is expected to be higher than testing the positive behavior: Each of the FEPs has to be tested by an own test sequence which increases the number of test cases (see [13]). Contrary to FEP, one CES ideally can cover all desirable EPs. A solution to reduce the effort of testing FEP is presented in the next section.

C. Modified Fault Model

The approach introduced in [13] uses event sequences, more precisely CES and FCES, as test inputs. The CES of an ESG, as "positive" tests, are supposed to lead to the exit node. If this is not feasible, the corresponding test is cancelled and marked as failed. During a positive test of a web application, an event may not be reachable in certain situations, e.g., if:

- a page was not loadable, although a previous event was executable,
- an error message has to be acknowledged,
- data input delivers a different structure of the program than the expected one,
- different dependencies of valid data have not been considered.

Faults are characterized as *structural sequencing faults* if no input data was involved, otherwise as *data-dependent sequencing faults*. Data-dependent sequencing faults are divided into *valid* and *invalid* ones, in accordance with test cases with valid and invalid input data. FCESs are expected to be not-executable. They are marked as failed, if the corresponding FEP is executable. Failed FEPs are grouped in *data-dependent sequencing faults* (failed FEP with input data) and *structural sequencing faults* (failed FEP without input data), in analogy to FEP with and without input data. Data-dependent sequencing faults need not to be analyzed furthermore as no valid data exist within FEP. *Surprises* are faults that are revealed coincidentally and not by the test case itself, perhaps due to observations of a tester, e.g., a missing error message or an image fault. In short, the fault classes are as follows:

- positive sequencing faults
  - structural sequencing faults (positive)
  - data-dependent sequencing faults (positive)
    - with valid data
    - with invalid data
- negative sequencing faults
  - structural sequencing faults (negative)
  - data-dependent sequencing faults (negative)
- surprises

D. Test Process

For a thorough testing all EP are to be covered by CES of minimal total length. This problem is a derivation of the Chinese Postman Problem (CPP, see [20]). In the following, the algorithm given in [13] will be modified to minimize the effort for testing web applications.

As a first step, structured nodes with decision tables of ESG are replaced by valid and invalid input data generated from those decision tables. The next step resolves nodes that are refined with other ESGs by the CESs of the refinement(s). Table IV depicts an example for input data completion for the eBay example (Fig. 4). The automatic generation of data out of decision tables can be done by a solution of the constraint satisfaction problem (CSP, see [21]). Russell, Norvig [21] describe a CSP as follows: "...a constraint satisfaction problem (or CSP) is defined by a set of variables, $X_1, X_2, \ldots, X_n$, and a set of constraints, $C_1, C_2, \ldots, C_m$. Each variable Xi has a nonempty domain Di of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset."
Each Rule of the decision table represents a CSP. The constraints C_F of the decision table contain the mapping of set X on D. The constraints C_C of the decision table are the constraints of the CSP. Note that constraints have to be set to true or false before submitting to the CSP according to the sets C_{True} and C_{False}. If a rule is consolidated, resolve the constraints that hold a don't care as follows:

- constraints $C = C_C \backslash (C_{True} \cup C_{False})$ can be disregarded
- resolve constraints $C = C_F\ (C_{True} \cup C_{False})$ to true

Definition 17: Given an $EP = \langle v_i, v_{i+1} \rangle$ where v_i is a DE with $DT = (C, A, R)$. Generating data for rules where $v_{i+1} \in A_x$ produces *valid data*. Generating data for rules where $v_{i+1} \neq A_x$ produces invalid data.

The test process requires that each FEP form an own FCES because exercising a FEP as a test case transforms the SUT into an undefined state. Therefore, SUT has to be reset. Events in web applications form elements on the GUI that are executed, e.g., by clicking a hyperlink, button, etc. Nowadays, test tools support validation of element-properties of a GUI (an example of such a test tool is given in Section 4). Instead of executing them, such tools can also check whether they are available and/or (if necessary) enabled. Thus, it is not necessary to execute the faulty event itself; instead,

TABLE IV
EXAMPLE OF A TEST SEQUENCE

No.	sequences without input data	No.	sequences with input data
1	[Click "sign in/out"; **login data**; click "sign in securely"; area user]	1.1	[Click "sign in/out"; **eBay User ID="Seller", Password="alright"**; click "sign in securely"; area user]
		(1.2)	[Click "sign in/out"; **eBay User ID="Admin", Password="admin"**; click "sign in securely"; area user]
		(1.3)	...

Legend:X.Y: Test case with valid data; (X.Y): Test case with invalid data

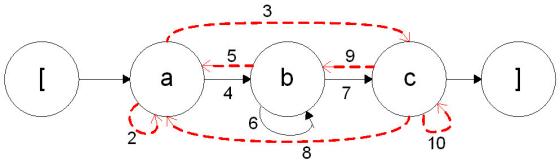

Fig. 10. Test execution

preconditions for its execution can be checked. This helps to reduce test costs. If the FEP is executable, the test is judged as failed, otherwise as passed.

A simple example is shown in Fig. 10. There exists exactly one CES that covers all EPs: [,a,b,b,c,]. The six dashed edges represent FEPs. In analogy to the previous approach six FCES have to be generated (see [13]). Now the FEPs are checked during execution of the CESs. Test execution reads:

1. execute event a
2. check, if event a is executable
3. check, if event c is executable
4. execute event b
5. check, if event a is executable
6. execute event b
7. execute event c
8. check, if event a is executable
9. check, if event b is executable
10. check, if event c is executable.

For this example, instead of executing seven test sequences in total, execution of one test sequence is sufficient. The algorithm, informally represented below (Algorithm 1), sketches the test process. Lines 1 to 17 deal with generation of test sequences, lines 18 to 28 describe their execution.

IV. TOOL SUPPORT

THe test suite *Tosca Commander* is a commercial test tool of *Triton Corp. (http://www.triton.at)*. For testing interactive systems, it provides functions as to *Capture*, *Play* and *Reporting*. The SUT is considered as black-box and focus is on inputs of the user and outputs of the underlying system. An interesting facility is the simulation of a user. The SUT is captured by the *Triton Wizard* that analyzes the GUI and controlling functions. The analysis results are available as modules in *Tosca Commander* (see Fig. 11, lower section).

On the basis of these modules, interaction sequences can be put together to form test cases by drag & drop operations, including input data whenever necessary (see Fig. 11, upper section). Furthermore, a single test case can be transformed into a template in order to generate test cases from the one given; those test cases differ from each other in input data values. The input data is entered in an Excel-Sheet and linked with the template. *Tosca* then automatically generates test cases out of the template and the Excel-Sheet. An advantage of *TOSCA Commander* is the identification of controls via technical criteria. Due to this fact, changes of the GUI have no effect on the test cases, except elements of the GUI are deleted. An important feature of our test approach is the validation of GUI-elements and their entities. FEPs need not to be executed because Tosca can check whether an element is available, enabled, etc. (see Fig. 11, circle).

It is a widespread practice in industry to create and execute test cases for testing the most important functionalities. They are constructed using specifications that describe the most important interactions. The TOSCA Commander itself has no functionality to systematically construct test cases. This is a critical disadvantage because the tester is responsible for creating applicable test cases in case that no test cases are given. Here, the approach introduced in this paper can help the tester to systematically construct test cases.

V. CASE STUDY

THe case study summarized in this section has carried out two sets of tests: The first one covers FEPs by means of FCESs, the second one covers FEPs by means of "extended" CESs. This is to show the cost savings by using the algorithm introduced in this paper in comparison to testing by FCESs as described in [13].

The SUT studied is a large commercial web portal with 53.000 LOC (lines of code): ISELTA (*Isik's System for Enterprise-Level Web-Centric Tourist Applications; http://www.iselta.com*). ISELTA enables travel and tourist enterprises, e.g., hotel owners, to create

Algorithm 1. Test process

```
n := number of the functional units of the system which fulfill a well-defined task;
l := required length of the test sequences;
for component 1 to n do
    for k := 2 to l do
        cover all ESs of length k by means of CESs subject to minimizing the number and total length
        of the CES;
        create a set V_FEP containing all FEP;
        foreach DE in CESs do
            generate multiple CES by replacing DE with valid/invalid input data of the Decision Table;
        foreach DE in FEPs do
            if DE is first event then
                generate multiple FEP with generated data for every rule of the Decision Table;
            if DE is second event then
                generate data for one rule of the Dec.Ta. and replace DE
            if DE is first and second event then
                generate data for one rule of the Dec.Ta. and replace DE's
        foreach CES and FEP with sESG do
            generate multiple CES/FEP by replacing the structured nodes with the CES of the sESG (see
            [13]);
        foreach CES do
            m := Length of the CES;
            for i := 2 to m do
                if event i executable then
                    execute event i;
                    foreach event j successor of event i and (i,j)∈ V_FEP do
                        if event j is executable then
                            mark FEP as failed;
                        V_FEP := V_FEP \ (i, j);
                else mark CES as failed and continue with next CES;
            mark CES as failed and continue with next CES;
```

their individual search & service offering masks. These masks can be embedded in the existing homepage of the hotels as an interface between customers and system. Potential customers can then use those masks to select and book hotel rooms and other services. Searching and booking of services are carried out in four successive steps. First, a user can enter his, or her, search criteria as to arrival date and departure date, service level (three stars, four stars, etc.), or type of catering. After clicking "search", the second mask presents the results. Moreover, the user can alter relevant factors to achieve further results. Fig. 12 exemplifies such a result list. The user can also choose "Hotel Details" to get more information about an offer. Additionally, he, or she, can enter individual data for booking a hotel room and services. After clicking "booking" the selections made are summarized; the customer can then acknowledge this, or re-select.

The ESG in Fig. 13 models the process of searching and booking rooms. We selected this ESG as an example out of 39 ESGs that were constructed for modeling and testing ISELTA. The decision table below (Table V) refines the node ***search criteria results*** of Fig. 13.

The modeled ESG possesses 11 nodes (without the pseudo vertices '[' and ']') with 27 valid EPs which leads to 94 FEPs ($= 11^2 - 27$). Three nodes are refined by decision tables so that 49 of the 94 FEPs include data input.

The dotted edges of Fig. 13 are bidirectional faulty edges (FEPs) from/to the nodes which have no legal connections with each other. The dashed edges represent

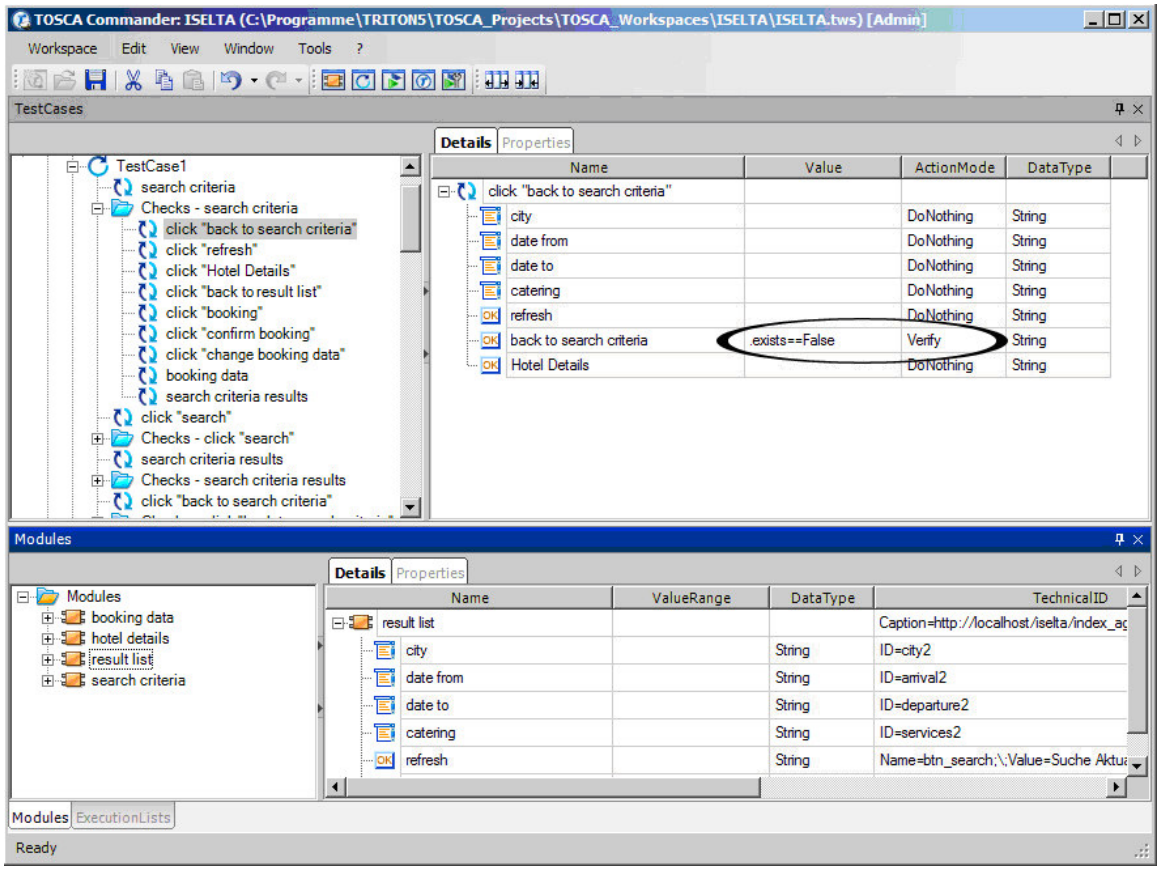

Fig. 11. Tosca Commander

TABLE V
DECISION TABLE TO NODE *search criteria results*

Decision Table - search criteria results	**R1**	**R2**
cities $\in$ {all cities, Bielefeld, Paderborn}	-	-
date from $\in$ {TT.MM.JJJJ}	-	-
date to $\in$ {TT.MM.JJJJ}	-	-
catering $\in$ { —, breakfast, half-board, full-board, all inclusive}	-	-
$datefrom < dateto$	T	F
click "refresh"	X	e

complementary (faulty) edges to existing (legal) ones, or faulty self-loops. The 27 legal edges (EPs) can be completely covered by a single ES. For testing the system behavior with respect to invalid data, i.e., for negative testing, 7 additional test cases were constructed. The system has to be restarted after each run as the test sequence leads the system into an undefined, faulty state. A *restart* is realized by loading the first web page. For a complete negative testing by means of FCESs, 115 additional test sequences were necessary to thoroughly run the FEPs. The second run of the case study saves these additional test cases by adding the mentioned checks to the test sequence with valid data.

Note that there are only 8 test sequences in total to be executed contrary to 123 test sequences in the first run (see Fig. 14). Expected results of the rerun are the confirmation of the revealed faults and time savings in execution. These expectations have been fulfilled: The case study revealed the same faults as in the first run and took less time. First of all the effort in preparing the test tool (see Section 5) took 17 hours in the first run and 8 hours in the second run. The automated test run with the test tool took 73 minutes, whereas the second run only took 9,5 minutes. The saved time to the test run amounts to 63,5 minutes or took only 13 % of the first runtime. The savings can easily be explained as

Fig. 12. Screenshot of ISELTA's result list

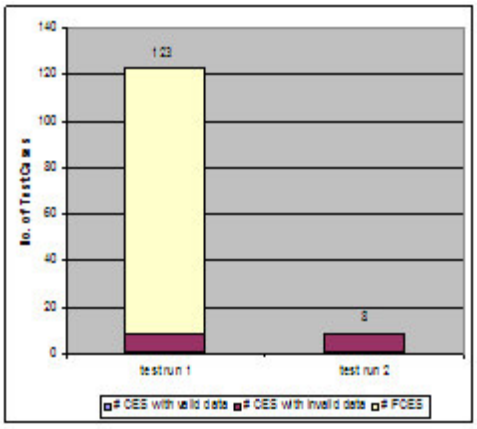

Fig. 14. Overview of the executed test cases

there are a great number of test cases and thus events that need not to be prepared and executed. Additionally, the system does not have to be restarted after each run of a test case during negative testing by a FEP. Table VI excerpts some of the faults that have been revealed during the test process.

VI. Conclusions

This paper introduced an approach for systematic test case generation for web applications by a structured event-based model. It allows a simple, nevertheless powerful modeling. Input data are structured and analyzed by decision tables. The option to test the undesirable behavior of SUT completes the approach. A case study briefly demonstrates the usage of the approach and validates the test process; test effort saved by the approach is roughly quantified. To avoid costly and error-prone manual test effort, a commercial test tool is used to support the introduced approach. Automated generation of data out of the decision tables is the subject of ongoing work. Future work aims at further reduction of the manual test effort and a better self-adaptability of the model due to changes in applications. Furthermore, dealing with data leads to the necessity of checking the data entered. Thus, one of planned future tasks is the extension for data checking.

References

[1] Jung, H.-W., Kim, S.-G., Chung, C.S., "Measuring Software Product Quality: A Survey of ISO/IEC 9126",IEEE Software, vol. 21(5), pp. 88–92, 2004.

[2] Di Lucca, G.A. and Fasolino, A.R., "Testing Web-based applications: The state of the art and future trends", Information and Software Technology, vol. 48(12), pp. 1172–1186, 2006.

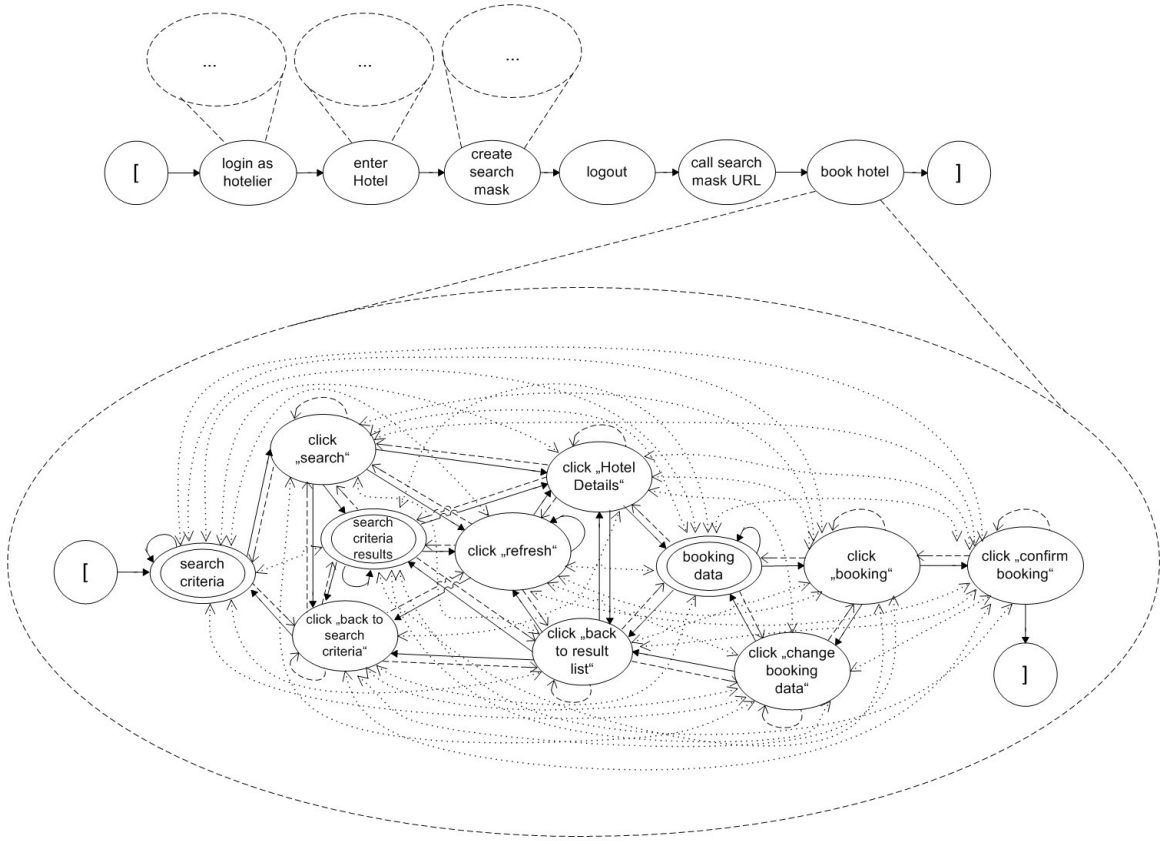

Fig. 13. ESG-model of component *search&booking*

TABLE VI
SOME EXAMPLES OF FAULTS REVEALED

No.	fault
1	after clicking ”Hotel Details”, ”refresh” could be executed
1	after clicking ”Hotel Details”, data input of search criteria results could be entered
1	”street” was no mandatory field in booking data
1	”house number” was no mandatory field in booking data
1	characters as ”phone number” possible

[3] Huberty, D. and Meyerhoff, D. “Test von Web-Anwendungen - Ein Praxisbericht”, Schriften zum Software-Qualittsmanagment. Erfahrungs- und Forschungsberichte, pp.15–35, 2001.
[4] Parnas, D.L., “On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System”, ACM Nat’l. Conf., pp. 379–385. ACM Press, New York, 1969.
[5] Shehady, R.K. and Siewiorek, D.P., “A Method to Automate User Interface Testing Using Finite State Machines”, Int. Symp. Fault-Tolerant Comp., pp. 80–88. IEEE Press, Washington D.C., 1997.
[6] White, L. and Almezen, H., “Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences”, Int. Symp. on Softw. Reliability and Eng., pp. 110–119. IEEE Press, Washington D.C., 2000.
[7] Belli, F., “Finite-State Testing and Analysis of Graphical User Interfaces”, Int. Symp. on Softw. Reliability and Eng., pp. 34–43. IEEE Press, Washington D.C., 2001.
[8] Conallen, J., “Building Web Applications with UML”, Addison-Wesley, Boston, 1999.
[9] Hennicker, R. and Koch, N., “Modeling the User Interface of Web Applications with UML”, UML 2001. LNI, vol. 7, pp.158–172. GI, 2001.
[10] Ricca, F. and Tonella, P., “Analysis and testing of Web applications”, Int. Conf. on Softw. Eng., pp. 25–34. IEEE Press, Washington D.C., 2001.
[11] Kung, D.C., Liu, C.-H. and Hsia, C.-T., “An Object-Oriented Web Test Model for Testing Web Applications”, Asia-Pacific Conf. on QS, pp. 111, IEEE Press, Washington D.C., 2000.
[12] Andrews, A.A., Offutt, J. and Alexander, R.T., “Testing Web applications by modelling with FSMs”, Software and System Modeling, vol. 4(3), pp. 326–345, 2005.
[13] Belli, F., Budnik, C.J. and White, L., “Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study”, The Journal of Software Testing, Verification and Reliability, vol. 16(3), pp. 3–32, 2006.
[14] Goodenough, B., and Gerhart, S.L., “Toward a Theory of Test

Data Selection", International Conference on Reliable Software, pp.493–510, ACM Press, New York, 1975.
[15] Myers, G.J., "The Art of Software Testing", John Wiley & Sons, New York, 1979.
[16] Binder, R.V., "Testing Object-Oriented Systems", Addison-Wesley, Boston, 2000.
[17] Gaudel, M., "Testing can be formal, too", TAPSOFT '95: Theory and Practice of Software Development. LNCS, vol. 915, pp. 82–96, Springer, Heidelberg, 1995.
[18] Koufareva, I., Petrenko, A. and Yevtushenko, N., "Test generation driven by user-defined fault models", IFIP 12th International Workshop on Testing of Communicating Systems, pp. 215–236. Kluwer, Deventer, 1999.
[19] Information processing – Specification of single-hit decision tables, ISO 5806:1984.
[20] Kwan, M.-K., "Graphic Programming Using Odd or Even Points", Chinese Math., vol. 1, pp. 273–277, 1962.
[21] Russell, S. and Norvig, P., "Artificial Intelligence"', Prentice Hall Series in Artificial Intelligence, pp. 137–160, Englewood Cliffs, New Jersey, 2003