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: Krüger, B., Linschulte, M., Cost Reduction Through Combining Test Sequences With Input Data, Proc. 6th IEEE International Conference on Software Security and Reliability (SERE), pp.207-216

Cost Reduction Through Combining Test Sequences With Input Data

Benedikt Krüger, Michael Linschulte
Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany
{bkrueger, linschulte}@adt.upb.de

Abstract— **Nowadays computer applications mainly depend on input data that bear additional constraints to be considered and evaluated carefully. Very often, this invokes test sequences that describe the way how and where the context in which the input data is to be entered, e.g., while testing graphical user interactions. Model-based testing allows deriving those test sequences from a model, for example, based on event sequence graphs (ESGs) where the nodes represent events. Unfortunately, combining ESG with the great variety of input data considerably inflates them with respect to the number of their nodes. To avoid this event inflation, previous work extended ESGs by decision tables to take this variety and constraints on input data into account. This paper extends the previous work and shows how to derive input data-supplemented test sequences and, at the same time, to considerably reduce effort for test generation and test execution. A case study drawn from a large commercial web portal evaluates the approach and analyzes its characteristics.**

Keywords-component; model-based testing; event sequence graph; decision tables; input data; test sequences.

I. INTRODUCTION

Software testing is the process that aims to increase confidence in the quality of software by checking whether the software does what is expected to do or not. Systematic testing of input data is of decisive importance and implicates high challenges to ensure the quality and reliability of computing systems. The reason is that nowadays computer applications make extensive use of input data, e.g., the possibility of collecting large amounts of data by web-based forms is often used in modern web applications and can even cause dynamic structural changes within the application. Mostly, there exist additional constraints on input data, which have to be evaluated carefully.

Although black-box-based test data generation has gained some attention in literature [1], most approaches are concentrated on code-based input data generation fulfilling criteria like branch coverage to check if a feasible walk through the given program exists [2], [3]. Moreover, they deal with generation of input data on the basis of numerical input values and their constraints but a single input data can also be a sequence of characters fulfilling a predefined format like a number plate of a car or a date.

Very often, testing input data requires test sequences, e.g., in GUI testing. Test sequences describe the path to the place where the input data is to be entered. *Model-based testing* has several advantages. It enables testers to concentrate on the relevant aspect of the given system under consideration (SUC). Furthermore, Model-Based testing may, depending on the underlying model features and the test criterion considered, offer the testers the ability of automatic generation of test sequences even if the source code of SUC is not available. Moreover MBT allows updating and reusing those generated test sequences with evolving requirements.

A previous work introduced event sequence graphs for modeling and testing user-system interaction which have been extended by decision tables to model constraints on input data. This paper continues this work. The novelties and merits are summarized as follows:

- Modeling and systematic evaluation of complex constraint sets considering input data not only for positive testing (data that should be accepted) but also for negative testing (data which should not be accepted).
- Generation of input data-supplemented test sequences with reduced effort for test execution.

A case study demonstrates the applicability of the approach and evaluates it by means of a practical and non-trivial web application that is commercially available.

The next section summarizes related work. Section III introduces the terminology and notion used in our approach. Section IV describes the new test case generation strategy. Section V validates the approach and determines its characteristic features over a case study based on a commercial web-based software system. Furthermore, a test tool supporting the test process is presented. Section VI concludes the paper summarizing the results and outlines our research work planned.

II. RELATED WORK

Edvardsson [2] presented a survey on automatic test data generation concentrated on code-based generation to fulfill criteria like branch coverage to check if a feasible path through the given program exists. Ferguson and Korel [3] divided these

methods into three classes: random, path-oriented and goal-oriented. The path-oriented approach considers a specific statement and selects a path to that statement (manually or automatically) for which test data is generated. The goal-oriented approach is similar to the path-oriented approach but omits the path selection step. There exists also a lot of work on code-based input data generation considering numerical inputs [4], [5], [6], [7], [8]. Some of them describe extensions to other types like enumeration or real data [4] [6] or even strings [6], [7]. Alshraideh and Bottaci [7] described an approach on string data generation considering string equality, string ordering or regular expression matching to cover program branches. But code-based test data generation is not the goal of this paper. Our goal is black-box-based test data generation where input data and given constraints are described by a model and code is not available.

Grindal et al. [9] present a survey on combination testing strategies and their classification. Combination testing strategies define ways to select values for individual input parameters and combine them to form complete test cases. An important refinement to the category partition method [10] are classification trees introduced by Grochtmann et al. [1]. Classification trees organize parameter into a tree structure where leaves represent choices on the given parameter. However, Grindal et al. [9] also report on remaining issues regarding combination testing strategies. One such issue is the identification of suitable parameters and parameter values. A second issue not adequately investigated deals with constraints among the parameter values. Both issues will be addressed in this paper where sets of constraints are build which have to be solved. Solving sets of constraints leads to the problem of constraint satisfaction where a solution has to be found that satisfies all the given constraints. Russel and Norvig [11] give a detailed overview on constraint satisfaction problems (CSP) and their solution. Bulatov et al. [12] summarize CSPs regarding global cardinality constraints which are additional requirements that prescribe how many variables must be assigned a certain value.

However, all of the mentioned approaches concentrate on the selection of input values. Combining them with test sequences is not considered, e.g., as it is required for testing interactive systems. Modeling and testing of interactive systems by means of state-based models has a long tradition [13], [14]. These approaches analyze the SUC and model the user requirements to achieve sequences of user interaction which form test cases. In [14] a simplified state-based, graphical model to represent UIs is introduced; this model has been extended in [15] to consider not only desirable situations, but also the undesirable ones. The approach introduced in [16] addresses black-box-testing of web applications. It rudimentarily considers data input and their order, but not constraints on input data and how input values are to be selected. Toersel [17] proposed a textual model which is able to consider input data but specific input values need to be externally associated.

III. THEORETICAL BACKGROUND

This section introduces the relevant background used to understand the approach.

A. Event Sequence Graphs

Belli et al. introduced an event-based approach based on modeling with event sequence graphs (ESG) for testing human-computer systems [15], [18]. Vertices of the ESG represent externally observable phenomena (“events”), that is, an environmental or a user stimulus, or a system response, punctuating different stages of system activity. Directed edges connecting two events define allowed sequences among these events.

The semantics of an ESG is as follows. Given two vertices a and b in V, a directed edge ab from a to b indicates that event b follows event a, defining an *event pair* (EP) ab. The remaining pairs $\overline{E}$ that can be constructed by all combinations $\hat{E} = V \times V$ of the nodes given in the alphabet Σ, but not in the ESG, that is, $\overline{E} = \hat{E} \setminus E$, form the set of faulty event pairs (FEP). The set of FEPs constitutes the complement of the given ESG, which is symbolized as $\overline{ESG}$.

As a convention, a dedicated start vertex, e.g., [, remarks the entry vertices Ξ of the ESG whereas a final vertex, e.g.,], represents the exit vertices Γ. Note that [and] are not included in Σ. An example can be seen in Fig. 1.

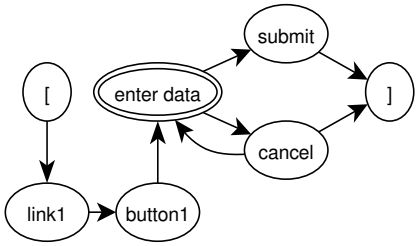

Figure 1. Login ESG

A sequence of $n + 1$ consecutive events that represents the sequence of n edges is called an *event sequence* of length $n + 1$, e.g., an event sequence of length 2 is an EP. An event sequence is complete if it starts at the initial event of the ESG and ends at the final event; in this case it is called a complete event sequence (CES). Occasionally, CESs are also called walks through the ESG given. Accordingly, a *faulty event sequence* of length n consists of $n - 2$ concluding, subsequent EPs and ends with an FEP. An faulty event sequence is complete if it starts at the initial vertex of the ESG; in this case it is called faulty complete event sequence, abbreviated as FCES. An FCES must not end at the final event. Faulty event sequences that are not complete, can be completed by event sequences (starters) that start at the entry and end at the first node of the considered faulty event sequence.

B. Decision Tables

Input data and their constraints will be modeled by decision tables (DT). DTs are popular in information processing [19] and are also used for testing, e.g., in cause and effect graphs [20], [21]. A DT logically links conditions (”if”) with events (”then”) that are to be triggered, depending on combinations of conditions (”rules”). A DT consists of three different areas:

1. Constraints (C)
2. Events (E)
3. Rules (R)

An abstract example can be seen in Table I. In general, constraints can be set to true or false. Rules define combinations of constraints as well as corresponding events that should be successful or not regarding a specific constraint set.

The set of constraints C is the union of C_F and C_C. Constraints C_F describe the domain D of input fields F. Each constraint of the set C_C involves some subset of input fields F and specifies the allowable combinations of values for that subset or additional domain restrictions.

TABLE I. ABSTRACT STRUCTURE OF A DECISION TABLE

		Rules							
		R1	***R2***	***R3***	***R4***	***R5***	***R6***	***R7***	**…**
Constr.	Constraint 1	T	T	T	T	F	F	F	…
	Constraint 2	T	T	F	F	T	T	F	…
	Constraint 3	T	F	T	F	T	F	T	…
	…	…	…	…	…	…	…	…	…
Events	Event 1	X		X		X			
	Event 2		X	X			X		
	Event 3	X	X	X	X	X		X	
	…	…	…	…	…	…	…	…	…

Definition 1: A decision table DT = {C,E,R} represents events that depend on certain constraints where:

- $C \neq \emptyset$ is the set of constraints
- $E \neq \emptyset$ is the set of events
- $R \neq \emptyset$ is the set of rules that describe executable events depending on combinations of constraints ■

DTs refine a node of the ESG (also called as *DT-nodes*). Such refined nodes are double-circled. The successors of such refined vertices represent the events of the DT and vice versa.

Definition 2: Let R be defined as in definition 1. Then a rule $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, E_x)$ 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
- $E_x \subseteq E$ is the set of events 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 ■

Definition 3: A decision table DT = (C,E,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. ■

Definition 4: A decision table DT = (C,E,R) is *redundance-free*, if $S = \emptyset$ with $S = \{x|(x, y, z) \in R \wedge (a, b, c) \in R \backslash (x, y, z) \wedge x = a \wedge z = c\}$. ■

Definition 5: A decision table DT = (C,E,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,E,R) is complete, redundance-free and consistent, if P(C) = S with $S = \{x|(x,y,z) \in R\}$. Such a DT has always $2^{|C|}$ rules according to P(C) where $|P(C)| = 2^{|C|}$.

Definition 6: Given a DT = (C,E,R) and two rules $r_1 = (C_{True1}, C_{False1}, E_{x1})$ and $r_2 = (C_{True2}, C_{False2}, E_{x2})$ with $r_1, r_2 \in R$. DT can be consolidated, if $E_{x1} = E_{x2} \wedge \exists c \in C_{True1} : (C_{True1} \backslash c) = C_{True2} \wedge (C_{False2} \backslash c) \in C_{False1}$. The new decision table is denoted as $DT_{new} = (C,E,R_{new})$ with $R_{new} = (R \backslash \{r_1; r_2\}) \cup r_3$ where $r_3 = (C_{True1} \backslash c, C_{False1}, E_{x1})$. 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 denoted with the *don't care* term '-'. Thus, a constraint that holds a don't care can be resolved to true and false or can even be disregarded.

C. Test Sequence Generation

CESs and FCESs form the test sequences (as test cases). Note that a CES is supposed to lead to the exit vertex. If this is not feasible, the corresponding CES is marked as failed (positive testing). In contrast, an FCES is not supposed to lead to the final event since it ends with an FEP which should not be executable (negative testing). If this is feasible, the corresponding FCES is marked as failed. For a thorough positive testing of ESGs, all EPs of a given ESG are to be covered by CESs of minimal total length and/or minimal number. This problem is related to the *Chinese Postman Problem* that attempts to find the shortest walk or circuit visiting each arc in a directed or even undirected graph [22].

For enabling the solution of the Chinese Postman Problem, the given graph is to be extended by additional edges until it forms an Eulerian graph. A directed graph is Eulerian if it is strongly connected and each of its vertices $v \in V$ has the same number of incoming and outgoing edges; that is, if it is a *balanced* graph. The resulting Eulerian cycle, which can be obtained by a standard algorithm in $O(|V| * |E|)$ time [23], is a minimal solution to the Chinese Postman Problem if the set of added edges is minimal.

Determining the set of minimal edges leads to a solution of the assignment problem [24]. This attempts to answer the question of how to assign n items (agents) to n other items (tasks), incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimized.

Formally, an assignment problem minimizes the objective function (1) for a given $n \times n$ cost matrix c_{ij} which fulfills the given constraints (2), (3) and (4) at the same time.

$$\min \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{ij} = 1 \quad (i = 1,...,n) \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad (j = 1,...,n) \tag{3}$$

$$x_{ij} \in \{0,1\} \quad (i, j = 1,...,n) \tag{4}$$

In our case, c_{ij} defines the number of edges of the shortest path (as costs) between vertex $i \in A$ and vertex $j \in B$ and n the number of elements of set A or B, respectively. After minimization, $x_{ij} = 1$ indicates that edges along the shortest path

from vertex i to vertex j have to be added. Note that set A and B should have the same size since the sum of all degrees in a given graph is zero, i.e., $\sum_{v \in V} \delta(v) = 0$ and, hence, $n = |A| = |B|$.

One of the most well-known solutions of the assignment problem, Hungarian algorithm, provides a solution with $O(n^3)$ time [24]. Other $O(n^3)$ solutions are given by Dinic-Kronrod [24] or Cycle Canceling [25].

D. Test Data Generation

In general, there exist two different approaches to solve a system of constraints: constraint solving and constraint satisfaction. Whereas constraint solving tries to find a solution mathematically, constraint satisfaction is based on search-based algorithms.

The automatic generation of data out of DTs can be done by a solution of the CSP. Russell and Norvig [11] 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 X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset."

CSPs are distinguished by the type of variables and constraints. Variables can be (1) *discrete* with finite or infinite domains or (2) *continuous*. Note that variables with infinite domains can be transformed into variables with finite domains by setting an upper and lower bound to the infinite domain if possible.

Each Rule of the DT represents a CSP. The constraints C_F of the DT contain the mapping of set X on D. The constraints C_C of the DT are the constraints of the CSP. Note that constraints have to be set to true or false according to the sets C_{True} and C_{False} before submitting them to the CSP. If a rule is consolidated, resolve the constraints that hold a 'don't care' as follows:

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

Generating data for rules with an event $e \in E_x$ produces valid data. Generating data for rules with an event $e \notin E_x$ produces invalid data.

But how is this data to be generated? A simple solution is to enumerate all possible combinations and evaluate them on valid solutions. But this is a very inefficient solution and even not possible for variables with infinite or continuous domains.

Another solution is to build a constraint graph $G = \{V, E\}$ where nodes $v \in V$ symbolize the variables and edges $e \in E = \{(v_i, v_j) | v_i, v_j \in V\}$ are annotated with the constraints [11]. An example can be seen in Fig. 2. The underlying constraint set can be defined as follows:

C={A>=2*B; B<C-2; B<=D-1; C=D+5; E>=3*D}

On the basis of this graph, different search algorithms can be run which are based on Depth-First Search or Breadth-First-Search. The advantage of Depth-First-Search algorithms is that they are able to return a single solution faster than Breadth-First-Search algorithms. The basic idea of both variants is to start with a single variable by assigning a value and extending the solution by assigning values to the other variables step-by-step. If the assignment of a value is not possible due to previously selected values, the algorithms go one step back and assign another value to the given variable.

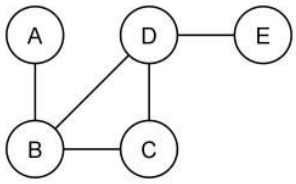

Figure 2. Constraint graph

Further details and techniques to improve this process are also given in [26]. It should be noted that a CSP can be transformed into a Boolean satisfiability-problem [27] (also known as SAT-Problem) as described by Le Berre [28]. But this will be not part of this paper and is not further investigated yet.

IV. Combining Test Sequences And Input Data

The previous section described how to generate test sequences on the one hand and input data on the other hand. The test sequence generation requires covering every EP, the test data generation requires to examine every rule at least once. The result of the input data generation is a set of EPs in which each EP describes some input values (also called as dataset) to be entered as well as the allowed (or not allowed) successor. The next step is to combine test sequences and input data for positive and negative testing. This enables the testers to verify input data- and event-handling of at the same time with a high variation in data sets.

A. Positive Testing

Combining test sequences and input data is simple if the set of test sequences contains as many EPs as needed for combining the input data given by the DTs. Unfortunately, it might occur that the generated test sequences do not contain enough of the given EPs to combine all datasets. A previous work [29] suggested to copy appropriate test sequences in such a case but this might not be minimal.

The number of datasets for testing can also be observed directly in the DT. The number of rules, which are denoted as valid to a given event, represents the number of datasets that must be proven for this event. The rules define those datasets according to the specific successor event. As shown in example 1, the number of datasets that can be tested by test sequences depends on the generated CES set.

Example 1: The ESG shown in Fig. 1 represents a typical login procedure. After clicking a specific *link* and *button*, login data must be entered. Depending on the given login data, allowed events are *submit* and *cancel*. Which of those events is

valid or invalid is modeled by the DT given in Table II. The resulting CES set according to Fig. 1 is:

```
CES1: [, link1, button1, enter data, submit, ]

CES2: [, link1, button1, enter data, cancel,
                         enter data, cancel, ]
```

This CES set contains the EP (*enter data*, *submit*) once and the EP (*enter data*, *cancel*) twice. Hence, this CES set allows to check one dataset for vertex *enter data* and the successor event *submit* as well as two datasets for the successor event *cancel*. The DT shown in Table II requires one EP (*enter data*, *submit*) and five EPs (*enter data*, *cancel*). ■

TABLE II. AN EXAMPLE OF A LOGIN FORM

		Rules				
		R1	***R2***	***R3***	***R4***	***R5***
Constr.	username ∈ $\sum^*$	T	T	T	F	F
	password ∈ $\sum^*$	T	T	F	T	F
	(username, password) ∈ Login	T	F	-	-	-
Events	cancel	X	X	X	X	X
	submit	X	-	-	-	-

Considering example 1 reveals following problem: The datasets given by the DT cannot be combined completely with the given CES set. Thus, how do we check the remaining datasets which cannot be combined? A straightforward strategy is to find the shortest sequence covering this EP and to duplicate it as much as needed.

Example 2: Since Table II contains five rules which denote the event "cancel" as valid successor, the EP (*enter data*, *cancel*) is required five times in total. The shortest sequence containing this EP is:

```
[, link1, button1, enter data, cancel, ]
```

Remember: Two datasets, e.g., for rules R1 and R2, are covered already by the generated CES set (see example 1). Hence, this sequence is duplicated 3 times to combine the remaining datasets.

```
[, link1, button1, (enter data)-R3, cancel, ]

[, link1, button1, (enter data)-R4, cancel, ]

[, link1, button1, (enter data)-R5, cancel, ]
```

The EP (*enter data*, *submit*) must be checked once and is already included in the given CES set. The resulting CES set looks as follows:

```
CES1: [, link1, button1, (enter data)-R1, submit, ]

CES2: [, link1, button1, (enter data)-R1, cancel,
                         (enter data)-R2, cancel, ]

CES3: [, link1, button1, (enter data)-R3, cancel, ]

CES4: [, link1, button1, (enter data)-R4, cancel, ]

CES5: [, link1, button1, (enter data)-R5, cancel, ] ■
```

The resulting CES set given in example 2 consists of 22 events. However, this number of events can be reduced any further. *Cycles* allow the reduction of events needed for the total coverage of the given datasets.

Example 3: The ESG given in Fig. 1 contains a reverse edge (*cancel*, *enter data*). Considering this edge for covering the required datasets results in the following CES set:

```
CES1: [, link1, button1, (enter data)-R1, submit, ]

CES2: [, link1, button1, (enter data)-R1, cancel,
                         (enter data)-R2, cancel,
                         (enter data)-R3, cancel,
                         (enter data)-R4, cancel,
                         (enter data)-R5, cancel, ]■
```

The resulting CES set given in example 3 consists of 16 events. But how do we provoke a coverage of the ESG that contains the required EPs as much as needed? As it can be seen in Section III.C., the test sequence generation process derives a CES set that covers all EPs. Remember: EPs are represented by edges in the given ESG. Therefore, the solution is to add additional edges before(!) the ESG gets balanced by solving the assignment problem. Every edge represents a dataset to be combined with the resulting CES set in such a case.

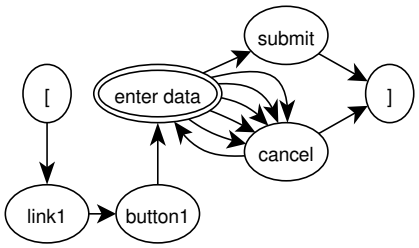

Figure 3. Login ESG with duplicated edges

Example 4: Table II requires the EP (*enter data*, *cancel*) five times. Since this edge exists once in the given graph, it has to be inserted four additional times. As the result, the edge exists five times in the ESG. The edge (*enter data*, *submit*) is required once. Since this edge is already present in the ESG, there is no need to insert additional edges. The result of adding edges is shown in Fig. 3. ■

The process after adding the additional edges is the same as described in Section III.C. First, an edge is added from pseudo vertex] to pseudo vertex [to ensure the entity of strong connectivity. Then, the given graph is to be extended by additional edges until it forms an Eulerian graph.

Example 5: Table III shows the resulting cost matrix to be solved for Fig. 3. According to Table III, following shortest paths (indicated by dark grey boxes) have to be added to the ESG given in Fig. 3 to create a minimal Eulerian cycle: $] \rightarrow enter\ data$, $cancel \rightarrow enter\ data$, $cancel \rightarrow enter\ data, cancel \rightarrow enter\ data$ ■

TABLE III. THE RESULTING COST MATRIX OUT OF FIG. 3

c_{ij}	**enter data**	**enter data**	**enter data**	**enter data**
]	4	4	4	4
cancel	1	1	1	1
cancel	1	1	1	1
cancel	1	1	1	1

After solving the assignment problem, the Eulerian cycle will start and end at pseudo node [. Note: The last vertex [of

the resulting Eulerian cycle does not contribute to the desired result and can be deleted. Furthermore, it might occur that the edge between vertex] and vertex [is traversed more than once in the resulting tour. Thus, the resulting tour is to be split up between the vertices] and [in every place to gain the desired set of CESs.

Example 6: The ESG given in Fig. 4 represents the balanced ESG with duplicated edges as required along the DT given in Table II. The resulting Eulerian cycle is as follows.

```
[, link1, button1, enter data, submit, ], [, link1,
button1, enter data, cancel, enter data, cancel,
enter data, cancel, enter data, cancel, enter data,
cancel, ], [
```

After splitting the cycle into single CESs, we have the following CES set.

```
CES1: [, link1, button1, enter data, submit, ]

CES2: [, link1, button1, enter data, cancel,
                         enter data, cancel,
                         enter data, cancel,
                         enter data, cancel,
                         enter data, cancel, ]
```

The CES set contains the EP (*enter data*, *submit*) once and the EP (*enter data*, *cancel*) five times, as required by Table II. Note: It equals the set of CESs of example 3. In total, this CES set contains 16 events. ■

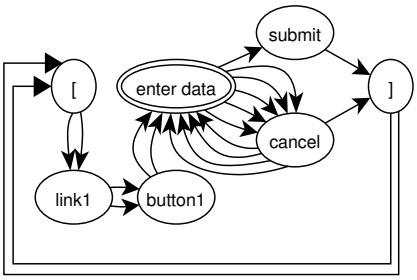

Figure 4. Balanced ESG

B. Negative Testing

Whereas positive tests verify the desired behavior of the SUC, negative tests check the undesired behavior, that is, where the user or system is not working as expected. FEPs test the undesired behavior and are represented by missing edges, which are not part of the given ESG; that is, $\overline{E}$ represents the set of all FEPs. For testing FEPs, the shortest sequence from pseudo node [to the first event of each FEP (also called as *start sequence* or *starter*) must be calculated and concatenated with the FEP, resulting in a FCES. Thus, a FCES consists of the shortest sequence and a FEP where the last event is not expected to be executable or arising.

Example 7: Fig. 5 illustrates the completed $\widehat{ESG}$, 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.

Below, some of the resulting FCESs are presented as negative test cases.

```
FCES1: [, link1, button1, enter data, submit, cancel

FCES2: [, link1, button1, enter data, link1

FCES3: [, link1, enter data
```

In order to cover the whole ESG, a FCES must be generated for each FEP. ■

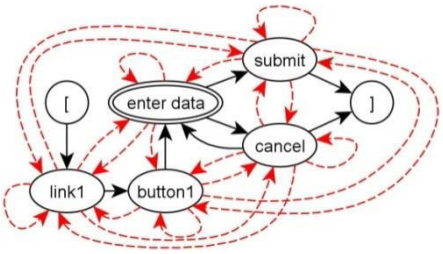

Figure 5. FEPs in sample ESG

Input data selection for a DT-node in a start sequence must result in a valid dataset for the specific successor event (as explained for CES). DT-nodes in an FEP need to be considered separately since they are to be enriched with input values. In general, following types of FEPs can be distinguished:

1. DT ∉ FEP
2. DT ∈ FEP
    a. DT is first element of FEP
    b. DT is last element of FEP

DT-nodes within FEPs contain no specific rule for data selection since the FEP is not modeled in the corresponding ESG. Therefore input values according to any rule may get selected. To achieve a better chance in detecting a fault, the data selection process may choose that rule, which is marked with the most valid successor events.

Example 8: FCES1 of example 7 testing the FEP (*submit*, *cancel*) contains the DT-node *enter data*. Since this DT-node is not part of the FEP, a valid dataset according to its successor event *submit* has to be selected. Note: To test as many different datasets as possible in the whole testing process, a new dataset can be selected for rule R1 of Table II. Thus, FCES1 will be supplemented as follows:

```
FCES1: [, link1, button1, (enter data)-R1, submit,
          cancel
```

In contrast, the FEP (*enter data*, *link1*) as in FCES2 contains the DT- node *enter data*. This data node has no rule defined for the successor event *link1* in Table II. Since rule R1 of this DT is marked as valid most frequently, one dataset according to this rule is selected. Thus, FCES2 will be supplemented as follows:

```
FCES2: [, link1, button1, (enter data)-R1, link1
```

The DT-node contained in FEP (*link1*, *enter data*) as given in FCES3 represents a situation where the input of data itself should not be possible. Nevertheless, input values are selected according to the rule with the most valid successor events. Similar to the FEP above, rule R1 is selected. The resulting FCES for this FEP is as follows:

```
FCES3: [, link1, (enter data)-R1
```

■

Using DT-nodes in ESGs provides an additional way of negative testing by means of *faulty decision table complete event sequences* (FDTCES). Consider an EP consisting of a DT-node and a successor event that is only valid if the correct input values are chosen. Selecting input values according to rules which mark the EP as invalid should result in an error; that is, the valid EP turns into an FEP.

Example 9: Consider the EP (*enter data*, *submit*) which should only be valid if the correct input values are chosen. The DT defined in Table II specifies four rules R2 to R5 for which the successor event *submit* is denoted as invalid. Thus, four FDTCES will be generated using the ESG shown in Fig. 1:

```
FDTCES1: [, link1, button1, (enter data)-R2, submit
FDTCES2: [, link1, button1, (enter data)-R3, submit
FDTCES3: [, link1, button1, (enter data)-R4, submit
FDTCES4: [, link1, button1, (enter data)-R5, submit
```

In the given login scenario, the user should not be able to login to the system with an invalid (*username*, *password*) combination. ■

V. CASE STUDY

This chapter analyzes and validates the testing approach presented in Section 4. SUC is a large commercial web portal with 53K lines of code: ISELTA ("Isik's System for Enterprise-Level Web-Centric Tourist Applications"). ISELTA enables travel and tourist enterprises e.g., hotel owners, to create their individual search & service offering masks and to integrate these masks into an existing homepage as an interface between customers and system. Potential end users can then use those masks to inform about services as well as to book them, e.g., hotel rooms, rental cars, etc. A screenshot of ISELTA is shown in Fig. 6. ISELTA makes extensive use of input forms and is therefore predestinated for evaluating the new approach.

For testing a large sub-system of ISELTA, 4 Scenarios have been selected that contain large amounts of input fields. The selected scenarios enable to manage:

- S1: user profile,
- S2: prices,
- S3: hotel entities,
- S4: special offers and additional services.

S1 enables to change data like contact information or password. S2 enables to enter and change standard prices for booking the hotel. S3 contains information like address of the hotel, type of the hotel, payment information and so on. S4 provides a facility to enter special offers like weekend-trips or additional services like bike rental.

Table IV gives an overview of the size of the models and the generated tests. Fig. 7 shows the ESG for testing scenario 4 and Table V shows one of the corresponding DTs.

TABLE IV. ESG OVERVIEW

	#nodes (DT)	#edges	#CES	#FCES	#FDTCES	Total
S1	11 (2)	35	4	66	4	74
S2	12 (5)	48	1	69	41	111
S3	10 (4)	75	27	22	62	111
S4	18 (5)	150	5	210	32	247
Total	51 (16)	308	37	367	139	543

TABLE V. DECISION TABLE FOR EDITING PERIODS

		Rules				
		R1	***R2***	***R3***	***R4***	***R5***
Constr.	date_from	T	T	F	T	F
	date_to	T	T	T	F	F
	period_name	T	T	T	T	T
	date_from < date_to	T	F	-	-	-
Events	Button:add	X	-	-	-	-
	Button:back	X	X	X	X	X
	DT:period	X	X	X	X	X

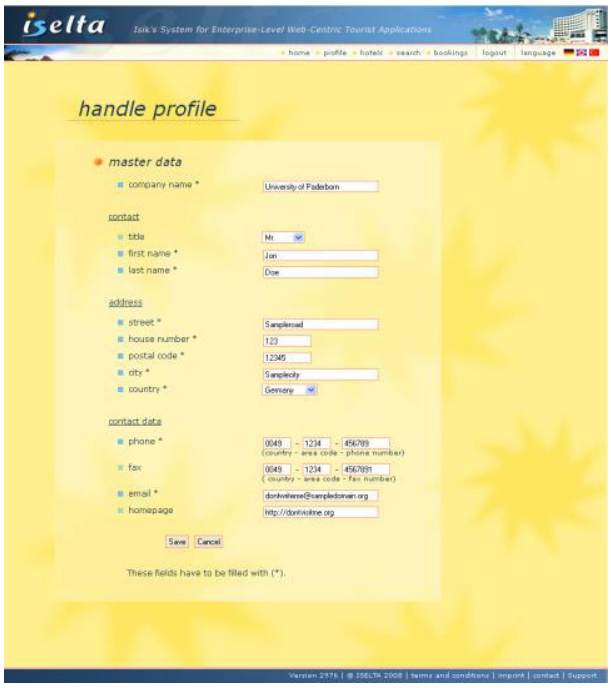

Figure 6. ISELTA screenshot

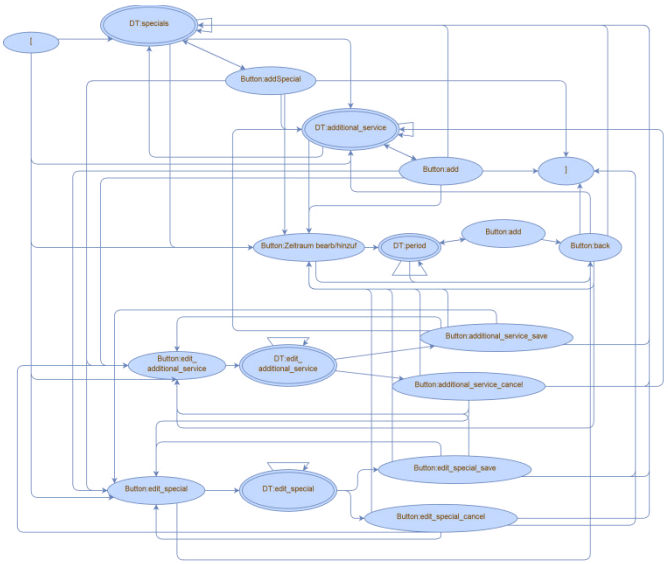

Figure 7. ESG for testing scenario 4

TABLE VI. NUMBER OF FAULTS DETECTED

	# faults		
	CES	***FCES***	***FDTCES***
S1	0	0	5
S2	0	0	0
S3	0	0	3
S4	0	0	3
Total	11		

TABLE VII. LIST OF FAULTS

no.	scenario	description
1	S1	invalid homepage address accepted
2	S1	invalid email-adress accepted
3	S1	invalid international area code accepted
4	S1	invalid area code accepted
5	S1	invalid phone number accepted
6	S3	button "upload" did not show an error message when no image was selected
7	S3	invalid homepage address accepted
8	S3	hotel can be created without providing information on the accepted payment method
9	S4	uploading textfile as image did not show an error message
10	S4	using past date allowed
11	S4	wrong error message when the departure date is earlier than the arrival date

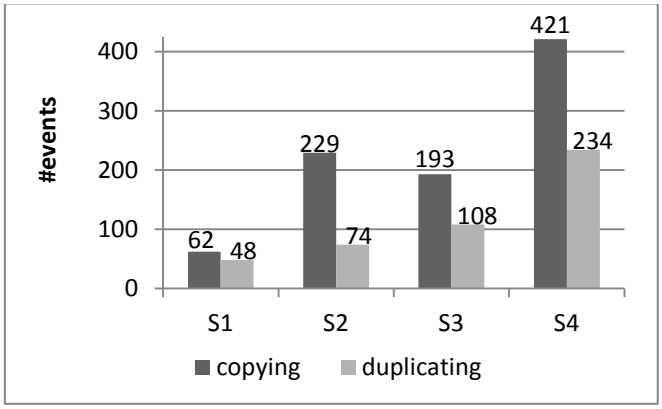

Figure 8. Comparison copying sequence with duplicating edges

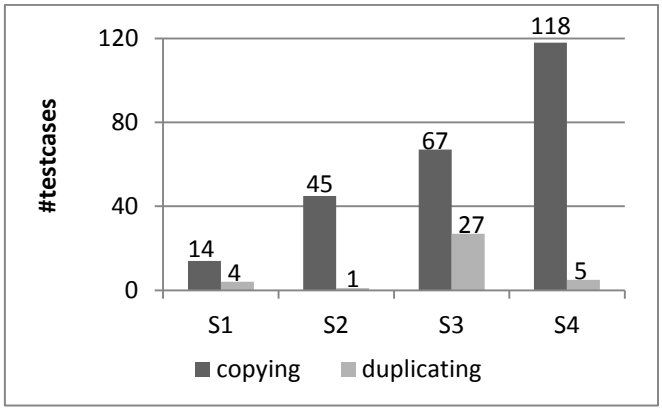

Figure 9. Comparison copying sequence with duplicating edges

A. *Results of the Case Study*

The results of the case study are summarized in Table VI and Table VII. Table VI shows the number of faults detected per scenario or ESG, respectively. Table VII gives a short description of each fault. All faults are related to some input of data. The system accepted data in some cases where it was not expected, e.g., a malformed email address. Furthermore, it has been able to use some past dates where it doesn't make sense and therefore should not be accepted as input. Since all detected faults depend on input data the case study verifies the structure of SUC.

Fig. 8 and Fig. 9 compare the effort for positive testing according to the new approach with the one that suggested copying appropriate sequences for combining input data. Fig. 8 compares the number of events, Fig. 9 compares the number of test cases. The average saving effort according to the total number of events executed is approximately 50%. The strategy for generating the negative tests did not lead to any savings since every FEP has to be tested by an own test sequence and improvements are not feasible.

B. *Threats to Validity*

In general, testing a system by a model-based technique assumes that the underlying model is correct and complete with respect to the considered features. The same holds for ESGs. We can just draw a conclusion about faults that can be detected by analyzing our model, that is, on faults in executing events and their order but not, e.g., on faults in database interactions. To keep the example easy to understand, the underlying ESG model we use is very simple in its fault modeling; it solely visualizes sequences of events that are considered as well as data dependencies. This can be seen as restrictive; however, this view is straightforward which explains the strength of the approach (for further explanation and fault detection effectiveness of ESG approach see [16] and [4]). In spite of its simplicity, also ESG used in the case study revealed numerous faults. We believe that the simplicity of the underlying ESG concept does not influence the validity of the new test case generation approach but rather underlines its characteristic features.

Another concern is about the generalization of the results achieved which heavily depend on the given SUC, its development process and on many more factors. In our case, a web application (ISELTA) is tested on the basis of several independent scenarios. But we are not able to conclude that the same results hold for other systems or models. In contrast, we believe that the test suite reduction capabilities will not change substantially.

C. *Test Process and Tool Support*

The case study has been supported by a tool called *Test Suite Designer* (TSD). TSD allows modeling of ESGs and DTs by means of a graphical user interface. A screenshot of the user interface can be seen in Fig. 10. Nodes can be refined by a DT. The second screen on the right side of Fig. 10 allows to model and edit DTs refining vertices of the corresponding ESG. TSD is extended by the testing approach described in Section 4 to generate test cases. TSD also supports the generation of specific values for the underlying domains following a specific pattern.

In order to enable the automatic execution of test cases, TSD allows specifying executable code fragments for each event. These code fragments represent instructions that must be executed by the given test execution environment to perform the specific event. The code fragments of DT-nodes can be extended by placeholders that are replaced by the generated input values later on. This is due to the fact that the code fragment itself is defined during the modeling process, but the selection of specific input values is done during the test generation process. After test generation, the single code fragments are combined along the generated test sequences to achieve executable test scripts.

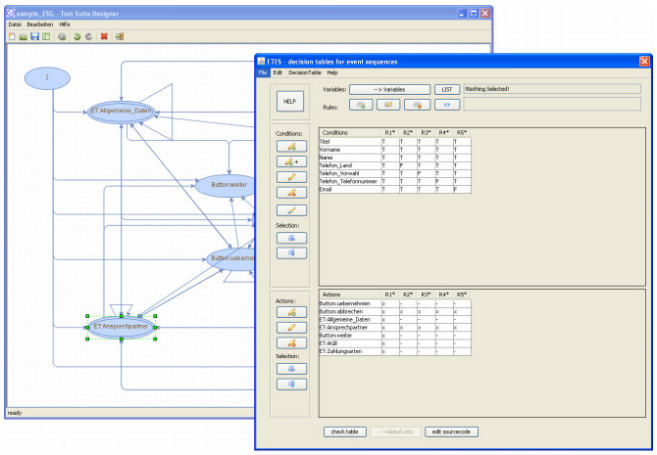

Figure 10. TSD screenshot

In our case, we used the freely available web test tool *Selenium IDE*[1] for test execution. *Selenium IDE* requires a specific kind of html code as test script. An example for entering some login data and pressing a login-button required to log onto ISELTA can be seen below:

```
<!--CES EVENT: 1--><!--DT - input login data-->
<tr>
        <td>type</td>
        <td>name=username</td>
        <td>MyName</td>
</tr>
<tr>
        <td>type</td>
        <td>name=password</td>
        <td>TopSecret</td>
</tr>
<!--CES EVENT: 2--><!--Button:login-->
<tr>
        <td>clickAndWait</td>
        <td>name=btn_login</td>
        <td></td>
</tr>
```

Loading such test scripts as files into *Selenium IDE* enables the user to execute the test cases automatically. Furthermore the user can track all events live in his browser during execution. The results of the test execution are given by success- or failure-flags for each test case and every executed event in each test case.

Since passed test cases and commands are highlighted with a green background and failed ones with a red background, Selenium IDE offers a nice overview of the execution of the test cases as shown in Fig. 11. Moreover, the user is able to track which events and data inputs result in failures.

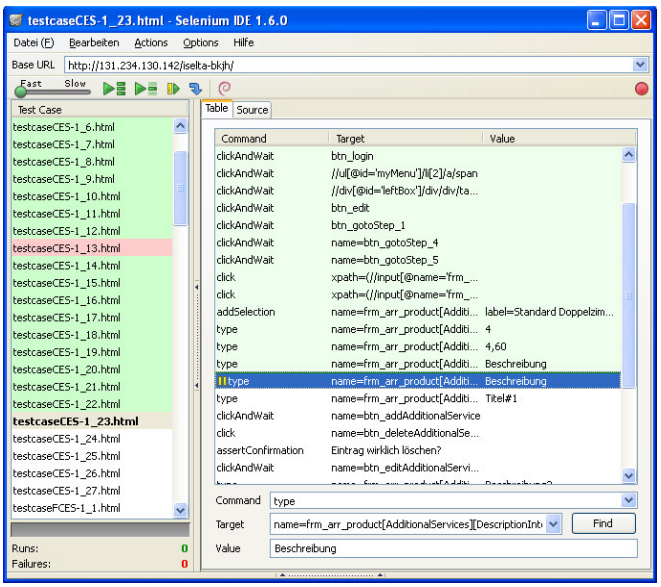

Figure 11. Selenium IDE

To sum up, the underlying test process is shown in Fig. 12. Based on a specification, ESGs and DTs are created using TSD. After that, TSD generates test sequences. Furthermore, TSD creates test-files that can be executed by Selenium IDE. Selenium executes the test files against the SUC. The result is a list of test sequences which has been executed successfully or not.

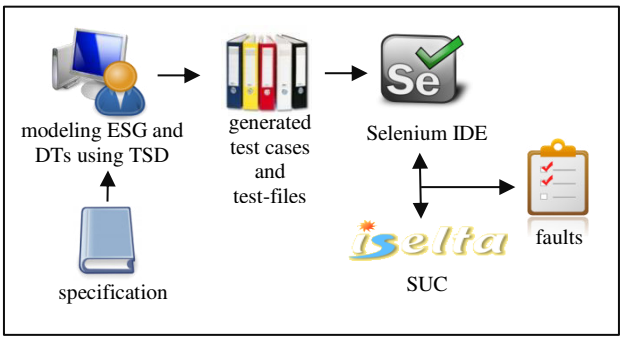

Figure 12. Test process

I. Conclusions

This paper introduced a strategy to derive input data-supplemented test sequences with reduced effort for test execution. First, the input data is generated to get the number of datasets to be tested. Then, this kind of information is considered during the generation of test sequences. Test sequence generation is supported by an event-based model called event sequence graphs. Input data and their constraints are modeled by and derived from decision tables. The strategy has been implemented in a tool called Test Suite Designer to support the automation of the approach. A case study based on a commer-

[1] Mozilla Plugin from http://www.seleniumhq.org/

cial system validated the approach. It turned out that the effort could be reduced by approximately 50 % in average.

Unfortunately, the strategy works only with a test sequence generation strategy covering event pairs. Therefore, future work will concentrate on the coverage of test sequences of higher length. Furthermore, there is some space to improve the automation of the test process. Test Suite Designer allows to generate test scripts which need to be loaded manually into a separate test execution environment. It is desirable that generated test cases are executed automatically out of Test Suite Designer to further improve the automation of the test process.

References

[1] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63-82, 1993.

[2] J. Edvardsson, "A survey on automatic test data generation," in *Second Conference on Computer Science and Engineering*, Linköping, Sweden, 1999, pp. 21-28.

[3] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering Methodologies*, vol. 15, no. 1, pp. 63-86, 1996.

[4] J. Zhang, "Specification analysis and test data generation by solving boolean combinations of numeric constraints," in *Asia-Pacific Conference on Quality Software*, 2000, p. 267.

[5] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *International symposium on Software testing and analysis*, 1998, pp. 53-62.

[6] M. J. Gallagher and V. L. Narasimhan, "ADTEST: A test data generation suite for Ada software systems," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 473-484, 1997.

[7] M. Alshraideh and L. Bottaci, "Automatic software test data generation for string data using heuristic search with domain specific operators," in *Software Testing Conference UKTest*, Sheffield, UK, 2005, pp. 137-149.

[8] R. A. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900-910, 1991.

[9] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167-199, 2005.

[10] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, pp. 676-686, June 1988.

[11] S. Russel and P. Norvig, "Constraint satisfaction problems," in *Artificial Intelligence*, Prentice Hall, Ed., 2003, ch. 5, pp. 137-159.

[12] A. Bulatov and D. Marx, "Constraint satisfaction problems and global cardinality constraints," *Commun. ACM*, vol. 53, no. 9, pp. 99-106, 2010.

[13] D. Parnas, "On the use of transition diagrams in the design of a user interface for an interactive computer system," in *24th national conference*, 1969, pp. 379-385.

[14] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *11th International Symposium on Software Reliability Engineering (ISSRE)*, 2000, pp. 110-119.

[15] F. Belli, "Finite-state testing and analysis of graphical user interfaces," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2001, pp. 34-43.

[16] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modelling with FSMs," *Software and System Modeling*, vol. 4, no. 3, pp. 326-345, 2005.

[17] A. M. Toersel, "Automated test case generation for web applications from a domain specific model," in *35th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, 2011, pp. 137-142.

[18] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study," *Software Testing, Verification & Reliability*, vol. 16, no. 1, pp. 3-32, 2006.

[19] International Standards Organization, Information processing - specification of single-hit decision tables (ISO 5806), 1984.

[20] G. J. Myers, *The art of software testing*. New York: John Wiley & Sons, 1979.

[21] R. V. Binder, *Testing object-oriented systems.*: Addison-Wesley, 2000.

[22] F. Belli and C. Budnik, "Test minimization for human computer interaction," *International Journal of Applied Intelligence*, vol. 26, no. 2, pp. 161-174, 2007.

[23] D. B. West, *Introduction to graph theory.*: Prentice Hall, 1996.

[24] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment problems*. Philadelphia: Society for Industrial and Applied Mathematics, 2009.

[25] H. Thimbleby, "The directed chinese postman problem," *Software: Practice and Experience*, vol. 33, no. 11, pp. 1081-1096, 2003.

[26] R. Barták, "Constraint propagation and backtracking-based search," *First international summer school on CP*, 2005.

[27] A. Biere, M. JH. Heule, H. van Maaren, and T. Walsh, *Handbook of satisfiability.*: IOS Press, 2009.

[28] D. Le Berre, "CSP2SAT4J: A simple CSP to SAT translator," *Publications of Centre de Recherche en Informatique de Lens (CRIL)*, 2005.

[29] F. Belli and M. Linschulte, "On negative tests of web applications," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 44-56, 2007.