International Journal of Software Engineering
and Knowledge Engineering
Vol. 26, No. 2 (2016) 183–215
© World Scientific Publishing Company
DOI: 10.1142/S0218194016500091

World Scientific
www.worldscientific.com

Input Contract Testing of Graphical User Interfaces

Tugkan Tuglular
Department of Computer Engineering
Izmir Institute of Technology, Urla, Izmir 35430, Turkey
tugkantuglular@iyte.edu.tr; http://www.iyte.edu.tr/∼tugkantuglular

Fevzi Belli
Department of Computer Engineering
Izmir Institute of Technology, Urla, Izmir 35430, Turkey
and
Department of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
belli@upb.de
http://ivknet.de/index.php/en/associates/81-websites/ivknet-mitarbeiter/107-fevzi-belli-en

Michael Linschulte
Andagon GmbH, Scheidtweilerstr. 4, 50933 Köln, Germany
m.linschulte@andagon.com

Received 29 April 2014
Revised 23 June 2014
Accepted 19 March 2015

User inputs are critical for the security, safety, and reliability of software systems. This paper proposes a new concept called *user input contracts*, which is an integral part of a design-by-contract supplemented development process, and a model-based testing approach to detect violations of user input contracts. The approach generates test cases from an input contract integrated with graph-based model of user interface specification and applies them to the system under consideration. The paper presents a proof-of-concept tool that has been developed and used to validate the approach by experiments. The experiments are conducted on a web-based system for marketing tourist services to analyze input robustness of system under consideration with respect to user input contracts.

Keywords: Model-based testing; design-by-contract; event sequence graphs; input validation testing; security testing.

1. Introduction

As Whittaker [56] indicated, "data is the lifeblood of software; when it is corrupt, the software is as good as dead." According to Whittaker, this is indeed the bottom line for software developers and testers. One must consider every single input from every external resource to have confidence in the system under consideration's (SUC) ability to properly handle malicious attacks and unanticipated operating environments. Deciding which inputs to trust and which to validate is a constant balancing act. Experiences from safety and security fields [56] have shown that user inputs, mostly obtained from graphical user interfaces (GUI), should be validated thoroughly to prevent attacks ranging from injection to denial of service and resulting in intrusion or even in system crashes. The same is true for safety violations.

The *contract* notion is the key used in our work to describe input properties in precise terms. Preventing invalid input from ever getting to the application in the first place is possible only at the user interface. Therefore, GUIs should be specifically designed to filter unwanted or unexpected input. This can be achieved through input contracts that are defined and used in our work. Model-based specification of input contracts is achieved through an input contract model, which enables the input data and corresponding actions to be defined with their constraints.

Our paper primarily addresses functional testing that checks whether or not a software element fulfills its specification. Thus, for simplicity, the term "testing" is used to refer to function-based, specification-oriented testing, or black-box testing. The testing approach proposed defines the process to perform tests that are derived from contracts supporting the creation of test input values and test oracles. Although there exist several approaches for contract-based automatic testing, the approach presented in our work is novel because it focuses on user inputs. The paper suggests that an automatic input testing process is possible with a GUI test driver that invokes mouse clicks and enters text into rich client GUIs. In this context, contracts form a valuable source of information regarding the intended semantics of the software. As noted by Ciupa and Leitner [14], the validity of a software element can be ascertained by checking the software with respect to its contracts. Therefore, contracts establish the ground for the automation of the testing process. Accordingly, the primary goal of our work is to develop and implement a fully automated test case generation for contract-based GUI input testing.

The proposed approach suggests converting graphical user interface specification into a model, which is employed to generate positive and negative test cases. An event-based formal model, called *event sequence graph (ESG)* [4], is chosen for the specification of GUIs. ESG merges inputs and events and turns them to vertices of an event transition diagram for easy understanding and checking the behavior of the GUI under consideration.

Our paper uses some aspects of the preliminary work of the authors [53], where specifications are utilized to find boundary overflow vulnerabilities. The novelty of the presented approach stems from the following substantial extensions and improvements:

(a) The notion of *input contract model*, which utilizes decision tables (DTs), is formally introduced. GUI input contracts are explored through an example that runs through the paper.
(b) The input contract testing approach is entirely new and introduced along with its algorithm for test case generation. This algorithm generates test cases from the decision table (DT) defined by input contracts, using parse trees and test coverage criteria, and utilizes equivalence class partitioning with boundary value selection. Integration of the input contract testing method with contract-supplemented ESGs is achieved to obtain a comprehensive GUI testing process.
(c) The present paper extensively improves and extends the tool introduced in the preliminary work [53] in order to include several new facilities, such as an input contract browser, test suite browser, test result browser, etc.
(d) The approach has been validated by means of a new case study, which tests a web-based large commercial system for marketing tourist services, such as hotel room reservation and special agreements for hotel rooms.

In summary, the main contribution of our work is the input contract testing approach. The proposed approach presents the following novelties:

- The input contract model for GUIs.
- A test case generation algorithm to generate test cases from the DT defined by contracts, using parse trees and applying equivalence class partitioning and boundary value techniques.
- A solution to test coverage and test oracle problems within the context of input contract testing.

Section 2 giving a summary of related work is followed by theoretical background, namely Sec. 3, which presents a theoretical comparison of existing definitions used in GUI testing. Section 4 describes modeling GUIs with input contracts and contract-supplemented ESGs. Section 5 presents the input contract testing approach. This enables the expansion of the contract-supplemented ESG view for considering GUI-based input testing processes. Section 6 presents the above mentioned case study that experimentally validates the approach introduced. This section also discusses the limitations of the approach, lessons learned by experiments, and threats to their validity. In Sec. 7 the tool developed for input contract testing is described. Section 8 outlines the planned future work and concludes the paper.

2. Related Work

It is widely accepted that interface signatures, even with comments, are insufficient to capture and control the salient properties of an application [2]. Thus, supplementary specifications are needed on the functional and non-functional aspects, such as architecture, performance, and quality of services [15]. Contracts, a technique for specifying behavioral compositions and the obligations on participating objects [23], seem a good candidate for this purpose. Contracts formalize object behavior collaborations and relationships [26]. There is no generally accepted formalism for contract representation.

Meyer [36] introduced DbC as an object-oriented design technique. According to DbC, a method can be evaluated with respect to preconditions, postconditions, and invariants similar to a legal contract. Wampler [54] explains DbC briefly as follows: "DbC is a way of describing the contract of a component interface in a programmatically-testable way. Preconditions specify the inputs the client must provide for enabling the component to successfully perform its work. Postconditions are guarantees made by the component on the results of that work, assuming the preconditions are satisfied."

Some of the approaches utilizing the contract idea for testing are as follows. Zheng and Bundell [58] introduced Test by Contract, which is an UML-based software component testing technique. There is also a contract-based testing technique for testing web services [22]. For uncovering errors some languages are extended to conform to DbC. In [43], the DbC concept is adapted into Python by integrating mechanisms, where method parameters and instance variables are evaluated by dynamic type checking. Guerreiro [20] used DbC in C++ through inheritance of the assertion class.

There are many research papers published on the subject of testing using formal specifications [27, 39, 44]. Specifications can be used for testing in several ways: as filter for invalid inputs, as guidance for test generation, as coverage criterion, and as an automated oracle [13, 47].

A testing criterion imposes rule(s) on a set of test cases [38]. In our work, full predicate coverage criterion [38], which requires testers to generate inputs derived from each clause in each predicate at minimum, is satisfied.

Test oracles decide whether the result of a test case is a success or not. Since manual development of test oracles is expensive [11] and a general methodology for the generation of test oracles does not exist, they are frequently complex and error prone [3, 10]. For instance, Binder [10] discusses the idea of using assertions based on contracts as test oracles. Taking it a step further, Ciupa *et al.* [12] used a contract-based oracle to achieve full automation in the testing process.

Input validation process controls the syntax of information provided through GUI and partly its semantics [21]. Missing input validation may cause a software to malfunction or may introduce vulnerabilities to be exploited by attacks. To validate GUIs various specification-based test techniques exist [40]. Event sequence graphs [4] and event flow graphs [33] can be used for the validation of GUI specification. In our work ESG notation, which intensively uses formal notions and algorithms known from graph theory and automata theory, is chosen because of its relevance to the approach introduced in our work.

Karam *et al.* [25] modeled GUIs using objects, their properties, and actions applied to those properties. Another method used for modeling GUIs is to specify them as a set of operations acting on objects [35]. In another work, Miao and Yang [37] treated GUI as a series of interfaces, where each interface is regarded as a state. They modeled a GUI state as a quadruple (W, O, P, V), where W represents windows, O objects, P properties, and V values, respectively.

Some model-based testing tools for GUI testing were available [34, 50]. Silva *et al.* [50] used task models for the generation of oracles in GUI testing. Memon *et al.* [34] developed a test oracle technique that utilizes formal model of the GUI under consideration to determine if a GUI behaves as expected for a given test case. There were also examples where DbC concepts are utilized for testing GUIs [29, 41]. Paiva *et al.* [41] described a GUI testing process using a formal specification developed in Spec, which employs DbC concept. Lamancha *et al.* [29] developed test oracle procedures from UML state machines.

In addition to model-based GUI testing, there are formal, theoretically sound approaches based on search algorithms, genetic algorithms, neural networks, and machine learning for GUI testing. Marchetto and Tonella [32] claimed that the main drawback of state-based approach is that the exhaustive generation of semantically interacting event sequences limits quite severely the maximum achievable length, while longer sequences would have higher fault exposing capability. They investigated a search-based algorithm for the exploration of large space of long interaction sequences, in order to select those that are most promising, based on a measure of test case diversity. Gross *et al.* [19] indicated that unit-level generated test cases for GUIs may be infeasible meaning that a test case may represent an execution that would never occur in reality. They also found that unit-level generated test cases for GUIs in spite of higher coverage produce false failures — that is, indicating a problem in the generated test suite rather than the program. They recommended considering search-based testing as an alternative to avoid false failures.

Rauf *et al.* [46] presented a GUI testing and coverage analysis technique centered on genetic algorithms, since they claimed that genetic algorithms search for the best possible test parameter combinations that are according to some predefined test criterion. Huang *et al.* [24] used a genetic algorithm to evolve new test cases to increase test suite's coverage while avoiding infeasible sequences. Their experimental results showed that the genetic algorithm outperforms a random algorithm trying to achieve feasible coverage goal in almost all cases.

Ye *et al.* [57] used acceptable images of GUI as a set of contracts and trained multi-weighted neural networks, which then acted as an oracle for testing the GUI. Gove and Faytong [18] claimed that a model of the GUI may not completely represent the GUI, and therefore may allow infeasible test cases to be generated that violate constraints in the GUI. They used two different machine learning techniques, namely *support vector machines* and *grammar induction*, to identify infeasible test cases. They demonstrated that these techniques are robust across different-length test cases and different GUI constraints.

The term *input contract for window systems* is defined as contracts that determine the input interface of the window under consideration [45]. Later, the term *input contract* is defined for software components as contracts upon which the component relies [49]. Similar definitions can be found for component-based software [17]. In our work, GUI input contracts are defined as contracts established on the GUI input component between GUI and user.

The approach presented in our work differs from the ones mentioned above in that it concentrates on GUI input components and presents a novel input contract testing approach with its new test case generation algorithm.

Differing from these works, Petrenko *et al.* stated in 2012 that a long-term challenge in test generation is related to the data aspect of test models [42]. One recent work proposed reducing test length for FSMs with extra states [51]. Our work integrates input contracts to GUI model, and thus is a step in this direction.

All of the above mentioned approaches have in common that they rather introduce notions and techniques for test case handling and generation, but hardly for optimization of test suites and thus test effort, as, e.g. contrarily suggested by Belli [4], Aho *et al.* [1] and Wan *et al.* [55] which will be considered in this paper.

In addition to the brief review and comparison of related, theory-based work in this section the next section will discuss further approaches that also are of sound theoretic background. Reasons why a specific method, namely Event Sequence Graphs is chosen, are given in the following section.

3. Theoretical Background

Our work follows model based testing approach for GUI testing. Two main types of models considered in this testing context are *holistic models* and *scenario models*, where the former are models which attempt to describe the expected behavior of a system under consideration (SUC) as completely as possible in a single formal specification, while the latter focus on a number of test scenarios describing various aspects of the expected behavior [42].

One type of holistic models is *sequential* models, which include *finite state machines (FSMs)*, *input/output transition systems (IOTSs)*, and various *state machines* [42]. Our work follows FSM-based GUI testing that is explained in the following before we compare it with other, theoretically sound approaches.

A FSM M can be represented by a directed graph $G = (V, E)$ where the set $V = \{v_1, \ldots, v_n\}$ of vertices represents the set of specified states S of the FSM and a directed edge represents a transition from one state to another in the FSM [1]. In this context, a precise definition of input and output operations is necessary which follows.

Definition 1. The sets of input and output operations defined for every state of M are called the *permissible input set* I and the *permissible output set* O, respectively. Each edge in G is labeled by an input operation $a_k \in I$ and a corresponding output operation $o_l \in O$. Thus, an edge in E from v_i to v_j has label a_k/o_l if and only if FSM

M, in state s_i, upon receiving input a_k produces output o_l, and moves into state s_j. Since there may be more than one transition from state v_i to v_j with different input and output operations, there are multiple edges in G. Therefore, an edge in G is fully specified by a triple (v_i, v_j, L) where $L \equiv a_k/o_l$, $L^{(i)} \equiv a_k$, and $L^{(o)} \equiv o_l$ [1].

Definition 1 assures that a path of G representing a FSM M is a sequence of inputs and outputs, which is suitable for capturing behavior of GUIs. Two main FSM based models that capture behavior of GUIs are Event Flow Graphs (EFGs) and Event Sequence Graphs. The definition of EFG is as follows [35]:

Definition 2. An *event-flow graph* for a component C is a quadruple $<V, E, B, I>$, where:

(1) V is a set of *vertices* representing all the events in the component. Each $v \in V$ represents an event in C;
(2) $E \subseteq V \times V$ is a set of *directed edges* between vertices. Event e_i follows e_j if and only if e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in E$ if and only if the event represented by v_y follows the event represented by v_x;
(3) $B \subseteq V$ is a set of *vertices* representing those events of C that are available to the user when the component is firstly invoked; and
(4) $I \subseteq V$ is the set of *restricted-focus events* of the component.

In the definition, a GUI component C is an ordered pair $<RF, UF>$, where RF represents a model window in terms of its events and UF is a set whose elements represent modeless windows also in terms of their events. Each element of UF is invoked either by an event in UF or RF.

Based on EFG notion, Memon *et al.* also defined integration tree to show how GUI components are integrated to form the GUI as follows [35].

Definition 3. An *integration tree* is a triple $<N, R, B>$, where N is the set of components in the GUI, $R \in N$ is a designated component called the Main component. It is said that a *component C_x invokes component Cy* if Cx contains a restricted-focus event e_x that invokes C_y. B is the set of directed edges showing the *invokes relation between components*, i.e., $(C_x, C_y) \in B$ if and only if C_x invokes C_y.

EFG has practical advantages for GUI test automation and supports tool builders with its notation. On the other hand, EFG does not explicitly define any mechanism for modularization, or refinement, and optimization, which are dedicated to GUI development. A similar FSM-oriented GUI testing approach, based on event sequence graph notion (ESG), provides necessary mechanism for refinement (see Definitions 4–7) and optimization. ESG achieves minimization [5] through the algorithm rooted in Chinese postman problem [16], whereas EFG uses genetic algorithm to eliminate infeasible test sequences [24].

Notions necessary for exploiting the advantages of ESG-based approach are borrowed from [4, 6, 28] and represented in the following, starting with a precise definition of ESG.

Definition 4. An *event sequence graph* $ESG = (V, E, \Xi, \Gamma)$ 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), Ξ, $\Gamma \subseteq V$ 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 $\langle \xi, v_0, \dots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \dots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \dots, k-1$ and $v \neq \xi$, γ.

Ξ (ESG), Γ (*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 ']' $\notin V$. 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 iff $(v, v') \in E$. 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.

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

Furthermore, α *(initial)* and ω *(end)* are functions to determine the *initial vertex* and *end vertex* of an ES, e.g. for ES $= (v_0, \dots, v_k)$ initial vertex and end vertex are $\alpha(\text{ES}) = v_0$, $\omega(\text{ES}) = v_k$, 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 $\gamma \in \Gamma$.

Definition 5. Let V, E be defined as in Definition 4. Then any sequence of vertices $\langle v_0, \dots, v_k \rangle$ is called an *event sequence* (*ES*) iff $(v_i, v_{i+1}) \in E$, for $i = 0, \dots, k-1$.

The function l(*length*) of an ES determines the number of its vertices. In particular, if $l(\text{ES}) = 1$ 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 included in ESs nor

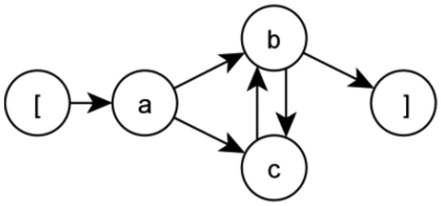


Fig. 1. An ESG with a as entry and b as exit and pseudo vertices [,].

considered to determine the initial vertex, end vertex, and length of the ESs. An ES $= \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (EP).

Definition 6. An *ES* is a *complete ES* (or, it is called a *complete event sequence*, *CES*), if $\alpha(\text{ES}) = \xi \in \Xi$ is an entry and $\omega(\text{ES}) = \gamma \in \Gamma$ is an exit.

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. CESs represent walks from the entry of the ESG to its exit, realized by the form

$$(\textit{initial})\ \textit{user inputs} \rightarrow (\textit{interim})\ \textit{system responses} \rightarrow \cdots$$
$$\rightarrow (\textit{final})\ \textit{system response}.$$

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. To keep the size of ESGs tractable, the ESGs topmost layer can be refined in several modularization steps resulting in a hierarchical set of ESGs. In Fig. 2, an example of a vertex v being refined by another ESG is given. The figure also contains the completed, or resolved, version without refinement.

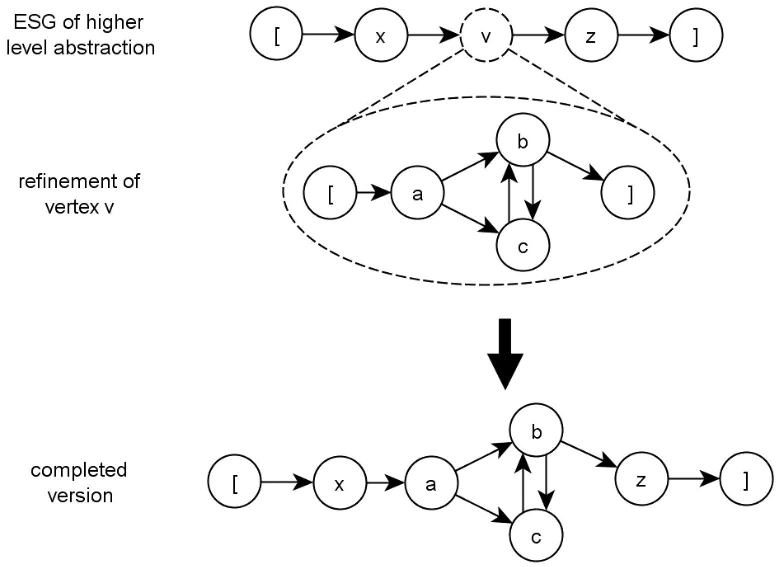


Fig. 2. Refinement of a vertex v and its embedding in the completed (resolved) ESG.

Definition 7. 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 $V_3 = V_1 \cup V_2 \setminus \{v\}$, and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1replaced}$ ('$\setminus$': 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 $v_i \in N^-(v)$ and $v_k \in N^+(v)$ (replaced arcs of ESG_1).

Modeling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model since vertices represent events and edges allowed sequences of events and not transitions as in automata theory. Assuming that a condition for choosing input data can be evaluated to true or false, the combination of conditions results in $2^{|\mathrm{C}|}$ combinations, where $|\mathrm{C}|$ represents the number of conditions. Each combination of conditions would have to be modeled as vertex and is to be connected with the appropriate successor. Thus a DT with n binary conditions subsumes 2^n nodes to realize a thorough evaluation considering all combinations. To avoid this inflation, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled. The successors of such refined vertices represent the actions of the DT and vice versa.

Definition 8. A *Decision Table* $DT = (C, A, R)$ represents actions that depend on certain constraints, where

- $C \neq \emptyset$ is the set of *constraints* (*conditions*) as Boolean predicates,
- $A \neq \emptyset$ is the set of *actions*, and
- $R \neq \emptyset$ is the set of *rules*, each of which triggers executable actions depending on a certain combination of constraints.

Decision tables are popular in information processing and are used for testing, e.g. in cause and effect graphs. A decision table (DT) logically links conditions ("if") with actions ("then") that are to be triggered, depending on combinations of conditions ("rules") [9]. The class diagram of DT is given in Appendix A.

Definition 9. Let R be defined as in Definition 8. Then, a *rule* $r \in R$ can be defined by $r = (C_{True}, C_{False}, A_x)$, where

- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to true,
- $C_{False} \subseteq C$ is the set of constraints that have to be resolved to false, and
- $A_x \subseteq A_{ui} \times A_{xcpt}$ 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 with
  - — A_{ui} containing possible user interactions,
  - — A_{xcpt} containing exception messages.

That is, one rule represents a specific combination of conditions, where each condition is evaluated either to true or to false. Depending on one rule, one or several follow-on actions are allowed. In the other way around, the execution of a specific action is only allowed if input data is chosen along a rule which possesses the considered action as allowed successor. As already stated above, the combination of conditions results in $2^{|\mathrm{C}|}$ combinations, that is, $2^{|\mathrm{C}|}$ rules can be formulated without producing redundancy. Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ under regular circumstances. In certain cases it is inevitable to mark conditions with a *don't care* (symbolized with a '-' in DT), i.e. such a condition is not considered in a rule and $C_{True} \cup C_{False} \subset C$. A DT is used to refine data input of GUI's.

Example 2. An example of DT is given in Table 1. This DT can be used to refine a node of an ESG. This node will be double-circuled and next event, which is an action in the DT, is decided with respect to DT that is attached to this double-circuled node. Such an ESG is called DT-supplemented ESG and is shown in Fig. 3.

Table 1. An example of DT.

		Rules		
		R_0	R_1	R_2
Constr.	v_0	F	T	T
	v_1	-	F	T
Actions	y	X	X	
	z			X

For DTs, such as the one presented in Table 1, X entry indicates an action, or for GUIs a user interaction. No exception is defined for actions y and z. As an example, rule 1 (R_1) reads as follows: **If** v_0 is resolved to *true* and v_1 is resolved to *false*, **then** action y will be executed. If this DT is used to refine a node of ESG, such as given in Fig. 3, then regarding to R_1 next event after v will be y and the ES will be $(\ldots, v, y, \ldots)$.

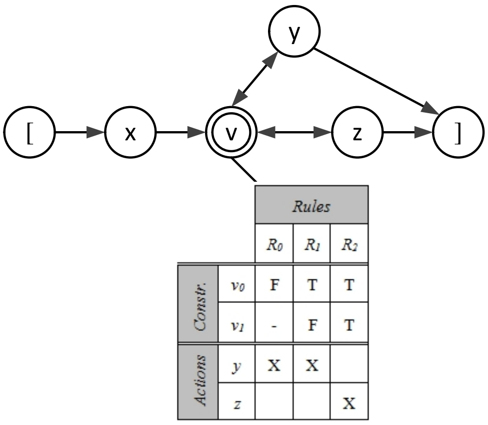


Fig. 3. An example of DT-supplemented ESG.

4. Modeling GUI with Input Contracts

This section, together with Sec. 5, introduces the notions and concepts of the proposed approach, illustrated by a simple example. In interactive systems, user inputs are usually forwarded to the systems application logic by means of a GUI that can be described as a composition of

- objects, such as menus, panels, labels, input/output areas, and buttons, and
- operations performed on the objects, such as "enter text" and "press button".

Users direct the course of action taken by the software through GUI interactions. Since the events cause changes in the state of the software through GUI objects, the GUI objects reflect that change in their state. GUIs can be characterized by the objects contained, the properties of the corresponding objects, and event-driven input to the corresponding objects.

Since large GUIs may contain numerous objects and operations, it is necessary to decompose a GUI into components to enable a manageable structure. Mao *et al.* [31] defined GUI component as an independent, replaceable module which encapsulates data and operations used to implement some specific functions. A GUI component is a unit composed of GUI objects that communicate through contractually specified interfaces.

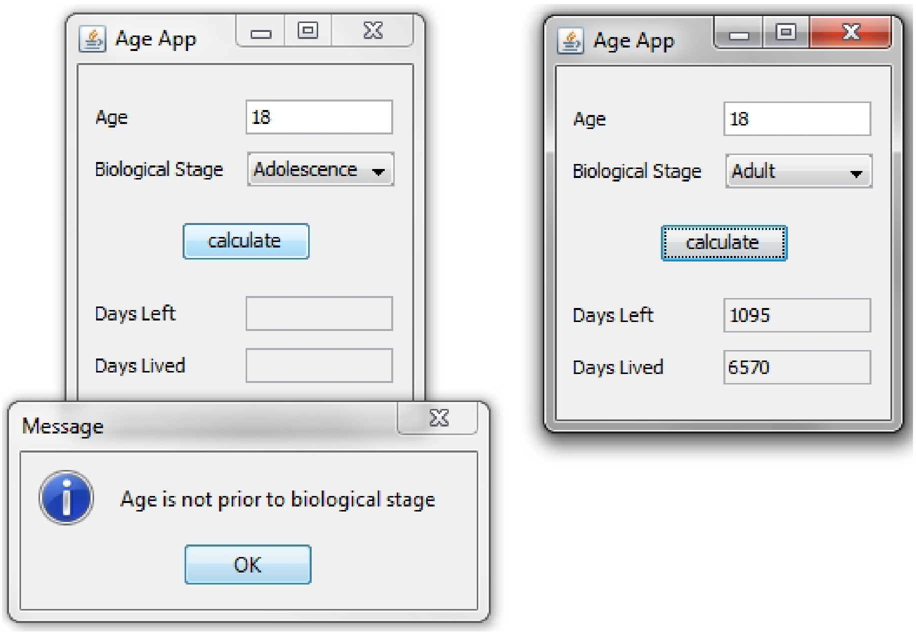
The notions and concepts of the proposed approach will be illustrated by a running example drawn from a simplified input component of `Age Application` which calculates the number of days lived up to that given age and number of days left to a biological stage from a given age. Note that this running example (Fig. 4), implemented using Java Swing, is kept very simple to ease understanding of the proposed approach. Section 6 presents a case study, which is drawn from a large, commercial, real-life project. This two-level presentation of examples is supposed to increase the readability and at the same time to demonstrate and validate the versatility and scalability of the approach introduced.

Definition 10. Given a GUI, the quadruple σ is an *input contract model* (Io, Dv, Ac, Co), where

- Io is the finite set of *GUI input objects*;
- Dv is the finite set of *data variables*;
- Ac is the finite set of *actions on GUI*;
- Co is the *input contract definition* represented by a *DT*.

With respect to the running example, its input contract has the following data variables used in DT constraints: `age`, which holds the entered age, and `biologicalStage`,which holds the selected biological stage.

It is assumed that the value for age is entered to the `age` variable through the GUI input object `inputArea[Age]`. Similarly, it is assumed that the value for biological stage is entered to the `biologicalStage` variable through the GUI input object `comboBox[Biological Stage]`. Biological stages are *adolescence* (12–20) and *adult*



Age 18
Biological Stage Adolescence
Days Left
Days Lived

Age is not prior to biological stage

Age 18
Biological Stage Adult
Days Left 1095
Days Lived 6570

Fig. 4. Screenshots of `Age Application`.

(21-death). These values are stored in variables as follows: `adolescenceLB` with a value 12 and `adultLB` with 21. The actions of *Age Application* are `calculate` and `error/warning`.

GUI input contract of Age Application is given in Table 2 with the following assertions:

- The user of *Age Application* promises that the value for age to be entered is of type integer, greater than 0 and less than 150.
- The user of *Age Application* promises that if the selected biological stage is *adolescence* then the value for age to be entered is less than 12 and if the selected biological stage is *adult* then the value for age to be entered is less than 21.
- The *Age Application* promises that if `calculate` button is pressed number of days lived for that age is to be displayed on response.
- If the user entered an invalid value for age, the most appropriate explanatory `error/warning` message will be presented.

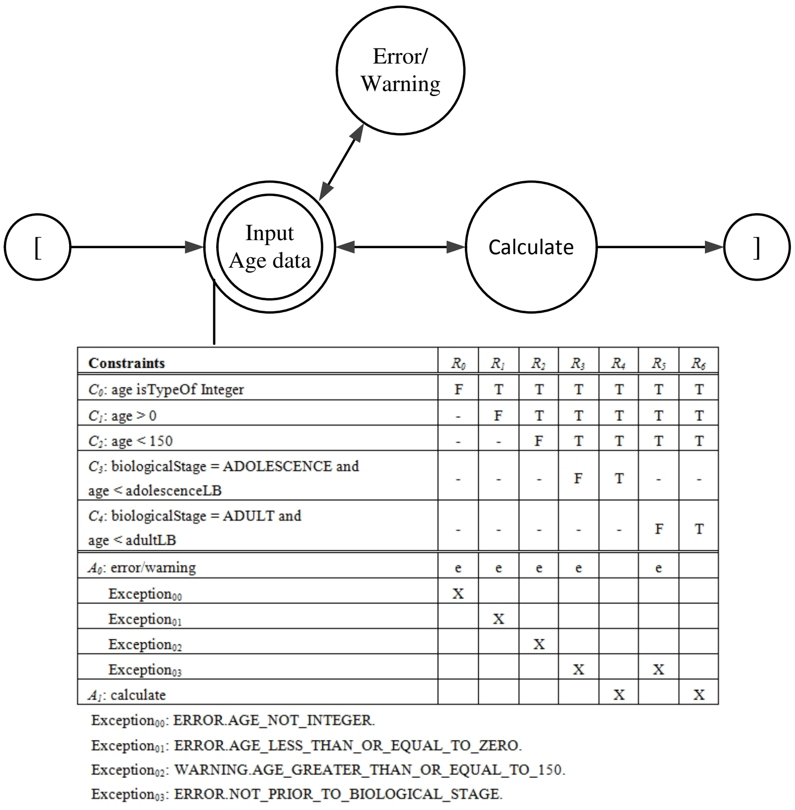
To sum up, complete input contract model of *Age Application* is given as follows: $\sigma = (Io, Dv, Ac, Co)$ with

- $Io = \{\texttt{inputArea}[\texttt{Age}], \texttt{comboBox}[\texttt{Biological Stage}]\}$;
- $Dv = \{\texttt{age}, \texttt{biologicalStage}, \texttt{adolescenceLB}, \texttt{adultLB}\}$;
- $Ac = \{\texttt{error}/\texttt{warning}, \texttt{calculate}\}$;
- Co is given in Table 2.

Table 2. DT for *Age Application*.

Constraints	R_0	R_1	R_2	R_3	R_4	R_5	R_6
C_0: age is Type Of Integer	F	T	T	T	T	T	T
C_1: age > 0	—	F	T	T	T	T	T
C_2: age < 150	—	—	F	T	T	T	T
C_3: biologicalStage = ADOLESCENCE and age $<$ adolescenceLB	—	—	—	F	T	—	—
C_4: biologicalStage = ADULT and age $<$ adultLB	—	—	—	—	—	F	T
A_0: error/warning	e	e	e	e		e	
Exception$_{00}$	X						
Exception$_{01}$		X					
Exception$_{02}$			X				
Exception$_{03}$				X		X	
A_1: calculate					X		X

Exception$_{00}$: ERROR.AGE_NOT_INTEGER.
Exception$_{01}$: ERROR.AGE_LESS_THAN_OR_EQUAL_TO_ZERO.
Exception$_{02}$: WARNING.AGE_GREATER_THAN_OR_EQUAL_TO_150.
Exception$_{03}$: ERROR.NOT_PRIOR_TO_BIOLOGICAL_STAGE.



Constraints	R_0	R_1	R_2	R_3	R_4	R_5	R_6
C_0: age isTypeOf Integer	F	T	T	T	T	T	T
C_1: age > 0	-	F	T	T	T	T	T
C_2: age < 150	-	-	F	T	T	T	T
C_3: biologicalStage = ADOLESCENCE and age $<$ adolescenceLB	-	-	-	F	T	-	-
C_4: biologicalStage = ADULT and age $<$ adultLB	-	-	-	-	-	F	T
A_0: error/warning	e	e	e	e		e	
Exception$_{00}$	X						
Exception$_{01}$		X					
Exception$_{02}$			X				
Exception$_{03}$				X		X	
A_1: calculate					X		X

Exception$_{00}$: ERROR.AGE_NOT_INTEGER.
Exception$_{01}$: ERROR.AGE_LESS_THAN_OR_EQUAL_TO_ZERO.
Exception$_{02}$: WARNING.AGE_GREATER_THAN_OR_EQUAL_TO_150.
Exception$_{03}$: ERROR.NOT_PRIOR_TO_BIOLOGICAL_STAGE.

Fig. 5. Contract-supplemented ESG for input component of *Age Application*.

As stated before, event sequence of GUI is modeled with contract-supplemented ESG. It is an ESG with a special DT, where conditions of DT come from constraints of input contracts. Input contract model provides a guideline for the construction of a contract-supplemented ESG for the GUI it represents. Input objects, such as `inputArea` and `comboBox`, and button objects indicate possible events. Event sequences are established among these events through drawing edges between vertices. Figure 5 presents contract-supplemented ESG of *Age Application*. The ESG given in Fig. 5 models GUI events of *Age Application* as follows: After age data is inputted, with respect to DT next action is determined. If Error/Warning is displayed, the user dismisses it and *Age Application* waits for another input. If there is no Error/Warning, the user presses Calculate button to see the result of calculation. Then the user can either exit the application or go back to input age data again.

5. Input Contract Testing

Specifications can be used for testing in several ways: as filter for invalid inputs, as guidance for test generation, as coverage criterion, or as an automated oracle. As explained in Sec. 4, an input contract is used when designing the input contract testing process for GUI. This section explains how input contract violations can be detected within the input context of GUI specification.

For GUI input contract testing, test scope is always a GUI. A set of GUI components that make up a window can be tested if event-based testing is integrated into input contract testing. Therefore, GUI input contracts are modeled with contract-supplemented ESGs in our work so that a seamless testing process can be achieved for a window or a composition of GUI input elements. Solutions for automating test case generation and test result interpretation stages are described in the following paragraphs.

A *test case* specifies input values for a method of an input component, which may work on one or more `inputArea`. A *test suite* is composed of test cases to check validation of all assertions offered by an input contract. The input values making up a test case can be derived from the constraints of a provided contract. Expected outputs are actions with or without exceptions given in DT. Please note that an input contract is not supposed to cover all inputs, its purpose is to filter.

GUI input contract testing process is given in Algorithm 1. In our work, full event coverage and full rule coverage criterion is fulfilled in terms of coverage. For full event coverage criterion, each event is executed at least once. In other words, each node of ESG is visited at least once. For full rule coverage criterion, each rule should be tested independently. These test cases should be sampled from input space composed of valid and invalid values of constraints.

The DT is used to produce test cases automatically. Since it is often not feasible to include all possible input values for a test case, the central question of testing is about the selection of test input values most likely to reveal faults. This problem comes down to grouping data into equivalence classes, which should comply with the

Algorithm 1. The input contract testing process.

```
generate the corresponding ESG
cover all events by means of CESs
foreach CES with decision tables do
    generate data-expanded CES using corresponding DT (input contract-
         based test case generation)
apply the test suite to GUI
observe GUI output to determine whether a correct response or a faulty
    event occurs
```

property that if one value in the set causes a failure, then all other values in the set will cause the same failure. Conversely, if a value in the set does not cause a failure, then none of the others should cause a failure. This property allows using only one value from each equivalence class as a representative for its set.

Equivalence class testing divides the test value domain into equivalence classes using contract conditions. Each test case selects one input value from each equivalence class. This approach is improved by boundary value selection of input values for numeric and date data, which appear at the boundaries of equivalence classes [30]. For string data, such as names, and for other types of data, such as files, a set of input values representing each equivalence class should be manually prepared in advance with respect to the input contract and then test input values are selected randomly for each equivalence class. Thus, in our work, cause-effect testing, which generates test values from decision tables, is used to strengthen equivalence class testing. In the presented approach, causes are input conditions and effects are represented by actions. This proposed approach is presented in Algorithm 2, namely input contract-based test case generation algorithm, which derives test inputs from contract-supplemented ESG.

The input contract-based test case generation algorithm produces test values for each rule in the DT. The DT is represented with a data structure that contains the set of variables, the set of clauses along with its variable(s) and their equivalence classes, the set of actions and exceptions, and the set of rules wherein each rule is composed of a conjunction of clauses and conjunction of actions and exceptions.

For each rule, the function `findTestInputValue` is called. It attempts to find values for variables that satisfy the Boolean expression that is a special case of a *constraint satisfaction problem* [48] of the corresponding rule in the DT. The function `solveCSP` determines valid and invalid equivalence classes for each clause and searches the values that make the Boolean expression true. The runtime complexity of the whole algorithm mainly depends on this function, which has to be solved for each rule of the DT.

The algorithm of `getAssignment` within the function `solveCSP` starts by assigning a value to a single variable and extends the solution step-by-step with the other variables by assigning values. If a value assignment to the current variable is

Algorithm 2. The input contract-based test case generation algorithm.

```
foreach event with DT do
     foreach rule in DT do
          findTestInputValue(DT, rule_i)
```

```
function findTestInputValue
begin
   tc_inputs : Test_Case_Inputs
   set_clauses : LIST[<Clause, Variable, EquivalenceClasses]
   b : BooleanExpr
   set_clauses ← getClauses(DT)
   b ← determineBooleanExpr(DT, rule)
   tc_inputs ← solveCSP(b, set_clauses)
   return tc_inputs
end
```

```
function solveCSP
begin
   assignment : LIST[<Variable , SelectedValue>]
   g ← getConstraintGraph(b, set_clauses)
   assignment ← getAssignment(g, assignment)
   return assignment
end
```

not possible due to previously selected values, the algorithm steps back and chooses next value from the set of boundary values for the current variable. This procedure is also called “simple backtracking”. The proposed algorithm combines backtracking with the techniques “Arc Consistency Check” and “Minimum Remaining Values”, see [48] for further information, to solve the given constraint satisfaction problem modeled by DT.

The runtime complexity for backtracking is given as $O(n * d)$ where n is the number of nodes for the corresponding constraint graph and d is the depth of the graph. The runtime complexity for the consistency check is given as $O(n^2d^3)$ [48]. However, in practice the number of variables on a GUI is strictly limited due to usability restrictions.

Simultaneously, this also limits the number of corresponding constraints so that the runtime complexity of this algorithm is negligible. Furthermore, the search space for numerical values may be narrowed by considering only boundary values of equivalence classes. Finally, the function `solveCSP` returns test case inputs for a rule in the decision table. Resulting test cases contain test input values as well as expected results.

The development of test oracles, which automatically performs a pass/fail evaluation of the test case, is an important issue in software testing. Developing such test oracles manually when writing test drivers is expensive and error-prone. Since our work proposes that assertions based on contracts can effectively be utilized as test oracles, the presented methodology is composed using different techniques to derive the oracles from the contracts in synchronization with the generation of test input values.

Fully automated testing requires automating the handling of oracles. In this case, evaluation of test results is straight forward due to the presence of contracts as specifications. Test cases are generated with expected test results automatically from the DT, which is constructed from input contracts. Since the test oracle in our work uses executable input contracts by means of checking test case results, test outputs can be easily compared with expected test results. Thus, the test oracle in our work enables an automatic pass/fail evaluation of the test case. If the obtained results match the expected results, then the test case passes, otherwise it fails.

For the running example, the test suite produced by the tool introduced in Sec. 7 employing Algorithm 1 and Algorithm 2 is as follows:

```
No. of CES: 1 with No. of Events: 14
[Input_Age_data(age:A,biologicalStage:Adult=>C0:F,C1:-,C2:-,C3:-,C4:-),
Error/Warning E00,
Input_Age_data(age:-1,biologicalStage:Adolescence=>C0:T,C1:F,C2:-,C3:-,C4:-),
Error/Warning E01,
Input_Age_data(age:200,biologicalStage:Adult=>C0:T,C1:T,C2:F,C3:-,C4:-),
Error/Warning E02,
Input_Age_data(age:18,biologicalStage:Adolescence=>C0:T,C1:T,C2:T,C3:F,C4:-),
Error/Warning E03,
Input_Age_data(age:7,biologicalStage:Adolescence=>C0:T,C1:T,C2:T,C3:T,C4:-),
Calculate,
Input_Age_data(age:25,biologicalStage:Adult=>C0:T,C1:T,C2:T,C3:-,C4:F),
Error/Warning E03,
Input_Age_data(age:18,biologicalStage:Adult=>C0:T,C1:T,C2:T,C3:-,C4:T),
Calculate ]
```

6. Case Study

The case study describes tests carried out for validating a web-based hotel reservation system named Isik's System for Enterprise-Level Web-Centric Tourist Applications (ISELTA) using a set of input contracts, thus demonstrating the practical use of the approach presented, and discussing also its characteristic features and limitations. These tests generated using the approach introduced in our paper revealed reliability- and safety-relevant faults, such as possible setting of a past arrival date in the case of hotel reservation system.

6.1. *System under consideration*

ISELTA, the system under consideration for the case study, is an online reservation system for hotel providers and travel agents, as well as end users. It is a product of the cooperation between a mid-size German company (Isik Touristik Ltd.) and University of Paderborn for marketing tourist services with 53,000 source lines of code. For instance, booking can be performed by the search menu given in Appendix B. It shows a website for booking special offers. These offers consist of a fixed number of days of accommodations within a certain time interval and may include other special services that are included.

For the case study described here, the "Enter Specials" module of ISELTA was selected. At first glance, this module seems to be relatively small; however, its behavior is quite complex because the module contains several different contexts. Accordingly, the formal model constructed is large and far from trivial. To keep the presentation of this case study in our paper compact, only the main interface is modeled resulting in 12 nodes, 25 edges, and 2 DTs each having 17 rules and analyzed to generate and perform tests.

The "Enter Specials" module enables a hotel or travel agent to offer and manage special prices for the selected hotel. The webpage shown in Appendix C is used to create such a special offer. The input fields for arrival, departure, number of items, price, and name are mandatory fields. Dates and the file name of the photo are selected by the user via dedicated dialogs. The site for administrating the specials also enables the user to edit or delete existing ones. Existing items are shown within an HTML table above the input fields. Each row describes an item and contains two buttons to delete or edit it. When editing an item, the existing data is loaded into the corresponding input fields. One of the DTs of contract for "Enter Specials" module is presented in Table 3.

6.2. *Test process*

The "Enter Specials" module is represented by an ESG given in Appendix D. It shows the ESG that models the menu given in Appendix C for setting up, editing, and deleting specials. In this ESG, "Special data 1" node with two circles is refined by the DT for "Enter Specials", which is given in Table 3. This DT augments the nodes with two circles of the ESG in Appendix D to consider the variety of the input data. In the same ESG, "Special data 2" node with two circles is refined by another DT, which has exactly the same content of DT for "Enter Specials", which is given in Table 3.

Without DTs, all combinations of conditions would have to be modeled as vertices which are to be connected with the appropriate successor. Thus, a DT with n binary conditions subsume 2^n nodes to realize a thorough evaluation considering all combinations. This fact demonstrates the power of DTs to enable a reduction in the size of the model.

The model developed for testing the whole ISELTA web application consisted of 8 ESGs augmented by 9 DTs containing the contracts. These 8 ESGs build up a hierarchical set of models and can be combined into one large resolved model (see Definition 7 and Fig. 2). Details of these 8 ESGs are given in Table 4. The ESG_8 corresponds to the "Enter Specials" described earlier.

Table 3. Decision table for testing specials.

	Conditions	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}
1	ArrivalDate > TODAY	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T	T	T
2	DepartureDate > TODAY	—	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T	T
3	DepartureDate > ArrivalDate	—	T	F	T	F	T	T	T	T	T	T	T	T	T	T	T	T
4	RoomType ∈ ROOMTYPES	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T
5	NoOfRooms MATCHES([1-9][0-9]*)	T	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T	T
6	TotalPrice MATCHES([1-9][0-9]*(,[0—9]{1,2})?)	T	T	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T
7	PhotoFileName = EMPTY	T	T	T	T	T	T	T	T	F	F	F	F	T	T	T	T	T
8	PhotoFileType MATCHES([jpg | gif])	—	—	—	—	—	—	—	—	T	F	T	T	—	—	—	—	T
9	PhotoFileSize > 0 MB	—	—	—	—	—	—	—	—	T	T	F	T	—	—	—	—	T
10	PhotoFileSize < 1 MB	—	—	—	—	—	—	—	—	T	T	T	F	—	—	—	—	T
11	DescriptionNat = EMPTY	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	T
12	DescriptionNat MATCHES([\w\.:;,!@$€äöüß\s]+)	—	—	—	—	—	—	—	—	—	—	—	—	T	F	—	—	—
13	DescriptionInt = EMPTY	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T
14	DescriptionInt MATCHES([\w\.:;,!@$€äöüß\s]+)	—	—	—	—	—	—	—	—	—	—	—	—	—	—	T	F	—
15	Name MATCHES([a-zA-Z0-9]+)	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F
A_1	Give Error Message			X	X	X	X	X	X		X	X	X		X		X	X
A_2	Add/Save	X	X							X				X		X		
A_3	Cancel	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 4. Size of ESG models used in the case study.

	ESG_1	ESG_2	ESG_3	ESG_4	ESG_5	ESG_6	ESG_7	ESG_8	Resolved ESG
# Nodes	6	4	7	14	17	7	22	12	**68**
# Edges	10	4	25	26	46	10	93	25	**251**
# Decision Tables	0	0	0	0	3	0	4	2	**9**

The resolved model, or ESG, from which test cases can also be generated, consists of 65 nodes and 244 edges. Note that the number of nodes and edges of this large ESG does not correspond to the sum of the single ESGs. This is due to the fact that the pseudo start and end nodes of refining ESGs are deleted during transformation. For more information on this transformation refer to Definition 7 and [6]. However, Belli *et al.* [7] indicated that for large models test case generation becomes very costly and the generated test case set is very large leading to infeasible long test execution time. They also showed that modularization of the modeling process can make test process efficient and feasible.

The resolved ESG is augmented by 9 DTs. These DTs have 4, 3, 12, 9, 11, 5, 6, 17, 17 rules respectively, in total 84 rules. Thus, these DTs saves the resolved ESG visualizing 84 additional vertices. Therefore, the resolved ESG represents in reality an ESG with $68 + 84 = 152$ nodes.

On the basis of the ESG models and DTs, test cases comprising 41 different kinds of datasets have been generated and executed, testing particularly the given input contracts. For the resolved ESG, 5 CES (see Definition 6) are automatically generated with events 805, 33, 3, 3 and 4 respectively, in total 848 events. Automatic test generation took less than 900 ms for all the trials performed on a computer with 2.70 GHz Intel i7 Processor and 8 GB RAM. Test generation is supported by a tool described in Sec. 7.

6.3. *Analysis of test results and lessons learned*

Table 5 contains the list of faults detected for the Specials module during test execution. In Table 5, the column title "Erroneous?" means whether the given test input should result in an error or not, whereas the column title "Error Message?" means that is there an error message with the given test input. In total, 25 faults have been revealed. The large number of faults related to the Specials mask is due to the fact that this is a new feature in ISELTA and had not yet been tested thoroughly.

Even some already known faults have been detected for other components, which have been marked as fixed in the past, e.g. check-in days could remain empty where at least one check-in day was expected. Moreover, faults 18 and 23–25 do not lead to errors, i.e. the system does not accept the given input data yet does not output an error message. Finally, the decision tables have been heavily consolidated, i.e. the rules has been reduced wherever redundancy and/or contradiction were detected.

Table 5. Outputs of test cases detecting a fault.

#	Test Input	Description	Erroneous?	Error Message?
1	Arrival > {TODAY} and Departure = EMPTY	Initial selection of date for departure is not arrival + 1	Yes	Yes
2	Photo_File_Size > 1 MB	Files larger than 1 MB in size is chosen.	Yes	No
3	Photo_File_Size > 1 MB	Files larger than 1 MB in size is appended.	Yes	No
4	Photo_File_Name = text.txt	All possible file types can be chosen.	Yes	No
5	departure < {TODAY}	If date of departure is set to a value in the past, only the date of the arrival is adapted.	Yes	No
6	No_of_items above 500 digits	Values of arbitrary length are possible for number of items.	Yes	No
7	TotalPrice above 500 digits	Values of arbitrary length are possible for prices.	Yes	No
8	Special above 500 digits	Values of arbitrary length are possible for names of a special.	Yes	No
9	Arrival = EMPTY and Departure > {TODAY}	It is possible to set only the date of the departure, but not of the arrival. This leads to a PHP warning after sending the data.	Yes	No
10	Name = "asdfg\"	If the name of a special ends with "\", a warning (parser error) is shown.	Yes	No
11	No_of_items = "0123"	If "0" precedes the number of items, it is not removed.	Yes	No
12	Price = "0123"	If "0" precedes a price, it is not removed.	Yes	No
13	TotalPrice = "100" and TotalPrice = "50.00"	Prices are not uniformly set on the overview to two decimal places.	Yes	—
14	Price = "455\56"	Price can contain "\", moreover, this character is duplicated after sending the data.	Yes	No
15	No_of_items = 0	Number of items can be set to "0".	Yes	No
16	Photo_File_Name = "test.jpg" and Price = "abc"	The path of the image file is lost, if one of the other fields is faulty.	Yes	No
17	Photo_File_Name = "0.jpg" and Photo_File_Size = 0	Files with size of 0 byte lets the application freeze.	Yes	No
18	Arrival = EMPTY and Departure = EMPTY	If input fields for arrival and departure are empty, these fields are not highlighted, when sending data.	No	No
19	Description_Nat = "abc\def"	character "\" duplicated in Description_Nat	Yes	—
20	Description_Int = "abc\def"	character "\" duplicated in Description_Int	Yes	—
21	Arrival < {Today}	Arrival can be set to the past using save action	Yes	No
22	Departure < {Today}	Departure can be set to the past using save action	Yes	No
23	TotalPrice = "abc"	No warnings for prices in case of faulty inputs.	No	No
24	No_of_items = "abc"	No warnings for no of items in case of faulty inputs	No	No
25	Arrival = EMPTY and Departure = EMPTY	There is no warning in case of empty dates for arrival and departure.	No	No

The results of the analysis of the test process encourage the generalization that in the practice, preconditions and postconditions are not paid attention adequately and thus necessary measures are not considered during software development. For existing software, tools such as the one used in the practical case study (see the next section) are strongly recommended to uncover deficient or inadequate control mechanisms in the software so that failures or undesirable situations that may occur can be prevented.

If software is to be developed from scratch, then formal representation of input contracts, such as presented in our paper, are considerably useful for the correct implementation of specifications, as well as for the automation of software development and of software testing. Not only user interfaces but also component interfaces may be separated from business logic through input contracts, which may help both correct development and validation of the business logic part of the SUC.

Moreover, it is observed that reusing input contracts is possible, which is also valid for ESGs and DTs. For instance, input contracts containing name based preconditions uses the "MATCHES([a-zA-Z]+)" regular expression in 3 DTs 4 times and the "MATCHES([a-zA-Z0-9]+)" regular expression in 2 DTs, repeatedly. Reusability, as in software development, increases the efficiency of specification writing and test case generation, which has a direct impact on the feasibility of the approach.

6.4. *Threats to validity*

Analysis of the results from the case study demonstrates that the proposed approach is efficient for testing especially condition checking parts of GUIs. All the results obtained in our work are valid with respect to the theoretical background explained in Secs. 3–6, especially Algorithms 1 and 2. However, following aspects are to be considered:

First, it is assumed that SUC can be modeled by a collection of ESGs. This might not be feasible in some situations. Furthermore, the model used in our work is assumed to be correct and complete for the desired behavior.

Second, in the case studies, internal system events or system outputs are not explicitly included in the models, partially because their focus is on the user events and they follow a black-box testing approach. However, they are used to determine contexts of user events and derive expected outputs. Nevertheless, explicit inclusion of these elements (internal system events or system outputs) may be required in some applications.

Third, coverage-based testing is used in the proposed framework. Using coverage-based approaches for large and complex systems might result in too many and too long test sequences. Although long test sequences tend to find more faults, unnecessarily too long can be unnecessarily too expensive [8]. The decision whether or not to stop testing is made based on the coverage criteria used.

7. Tool Support

For practical use of the presented approach in real-life projects, an input contract testing tool is developed in Java. This tool loads the contract as a DT from a file and displays the contract via its *Decision Table* window. The *Model Browser* window presents the model for test case generation (see Appendix D). It provides basic functionalities for creation, editing, saving, and loading an ESG model. Moreover, it is able to check the ESG model for its correctness with respect to Definition 4. The model also contains double circled vertices with corresponding decision tables according to the input contracts. An example *Decision Table* window is given in Appendix E.

Another window of input contract testing tool is *Test Suite Browser*, which generates and presents the generated test cases. Moreover, Test Suite Browser is able to generate test scripts that can be run automatically. This is due to the fact that, within the model browser, a vertex can be annotated with pieces of source code executing the underlying event within a given test execution environment. This enables automatic generation of test scripts along the computed test sequences.

Developed main packages of the input contract testing tool are inputContract, fsm, esg, decisionTable, and solver. External graphics libraries, such as jgraphx.jar and jgrapht.jar, are utilized in the tool along with swing and database connection libraries.

The process to use this tool is as follows: First, the ESG model and the DTs are constructed in order to fulfill the imported input contracts. On top of the model, the test cases and test scripts can be generated via Test Suite Browser in the next step. The test scripts are not executed by the input contract testing tool automatically. For automatic test execution, the *Selenium test runner* (http://docs.seleniumhq.org/) has been used, which provides automatic execution of the generated test scripts. All tests have been executed using Mozilla Firefox Browser Version 28.0, which is controlled by Selenium and which is installed on a computer with 2.70 GHz Intel i7 Processor and 8 GB RAM.

8. Conclusion and Future Work

Based on formal notions of event sequence graph, decision tables, and contracts, our paper introduces a practical and novel framework to model and test input components. The approach generates test cases and test oracles using preconditions and postconditions. For cost reduction during project work, a tool has been implemented, which includes a fault detection component.

A case study validates and analyzes the proposed approach. Revealing 25 faults of the SUC investigated in the case study indicates that the approach is able to detect faults, especially deficiencies in the precondition checking parts of GUIs with respect to input contracts. To sum up, the main contributions of our work are:

- The introduced input contract model can easily be implemented using DTs.

- The input contract testing approach introduced includes an algorithm to generate test cases from a DT constructed using input contracts, parse trees and applying equivalence class partitioning and boundary value techniques. For an ESG model with 68 nodes, 251 edges, and 9 DTs with 84 rules in total, our algorithm automatically generates 5 complete event sequences with 848 events in total.
- Aspects of test coverage and test oracle problems within the context of input contract testing are presented and applied.

Different paths are set forth for future work. The first one is to perform an empirical research on the presented approach to answer the following questions: a) How much overhead does the presented approach impose on a tester of large software systems? b) How far can a tester develop contracts for such an application and how long would it take? c) How would a tester developing contracts for applications impact speed of execution during testing?

Analysis of the test case generation process reveals the fact that ESGs are to be transformed into one large model for test case generation. On the other hand, DTs could be consolidated, which results in reduced number of rules. Both facts give some clues about the scalability of the presented approach. Transforming ESGs into one large model might complicate test case generation and the intuitive partitioning of SUC intended by the tester would be lost. Further test generation techniques are considerable, which make use of the intuitive partitioning of the tester to reduce and/or simplify test sequences and their generation, especially with regard to input contracts. The more input contracts exist, the more costly is their evaluation. This is due to the fact that adding just one single input contract doubles (in the worst case) the number of combinations of input contracts to be tested. Thus, further techniques to reduce the evaluation complexity of large sets of input contracts could be helpful, such as partitioning of input contracts that could be achieved by a hierarchical set of DTs.

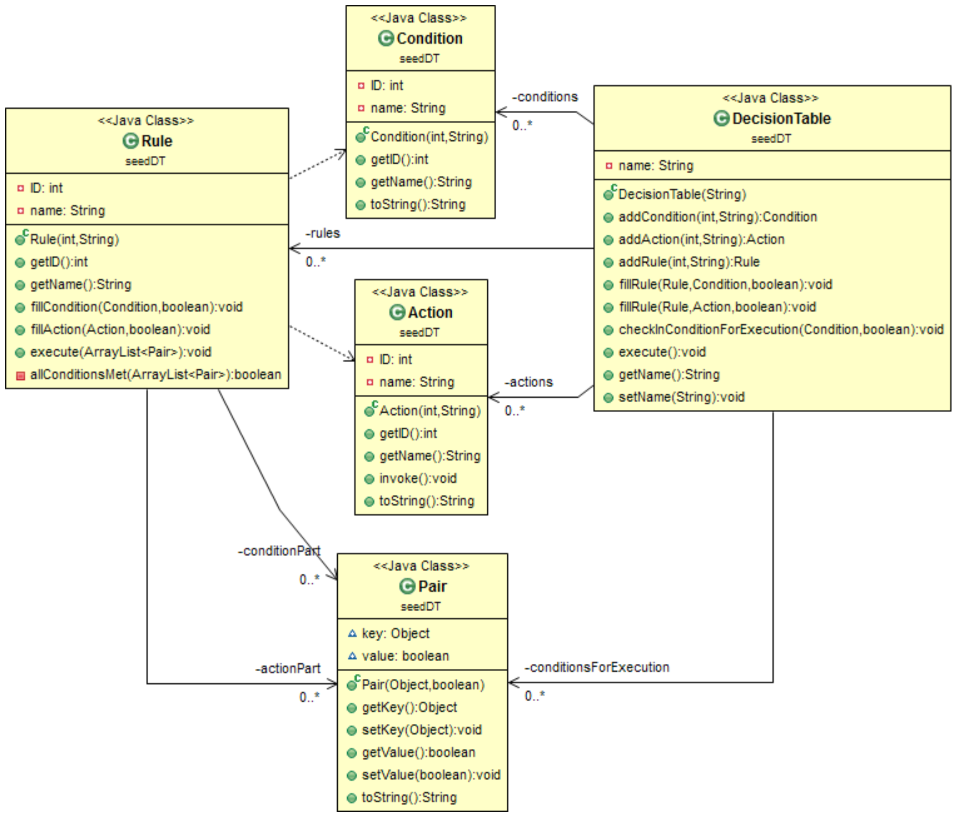
The second path for future work is to introduce formal semantics for input contracts. Therefore, extension of the approach is planned for refinement and inclusion operations on contracts. These operations aim to provide distinct means to express complex input behavior in terms of simpler behavior. Furthermore, a refinement enables specialization of contractual obligations and invariants of other contracts, whereas inclusion allows contracts to be composed from simpler sub-contracts [23]. With these contract operations, an approach for combining input components and testing them as a single unit is to be defined and implemented. Those extensions are necessary to judge whether or not it pays off to go through the effort to test for specific types of errors. More formalism for input contracts is necessary to systematically discuss and justify the differences between input contracts and classical Meyer contracts with respect to inheritance mechanism, @pre operator, simple data types, and method calls, which is also planned as future work.

A further future goal is to work on mutation-based analysis of contracts [52] to estimate their efficiency. In addition, contracts can be used to correct existing source code with missing/erroneous input validation by static insertion of corresponding input validation source code.

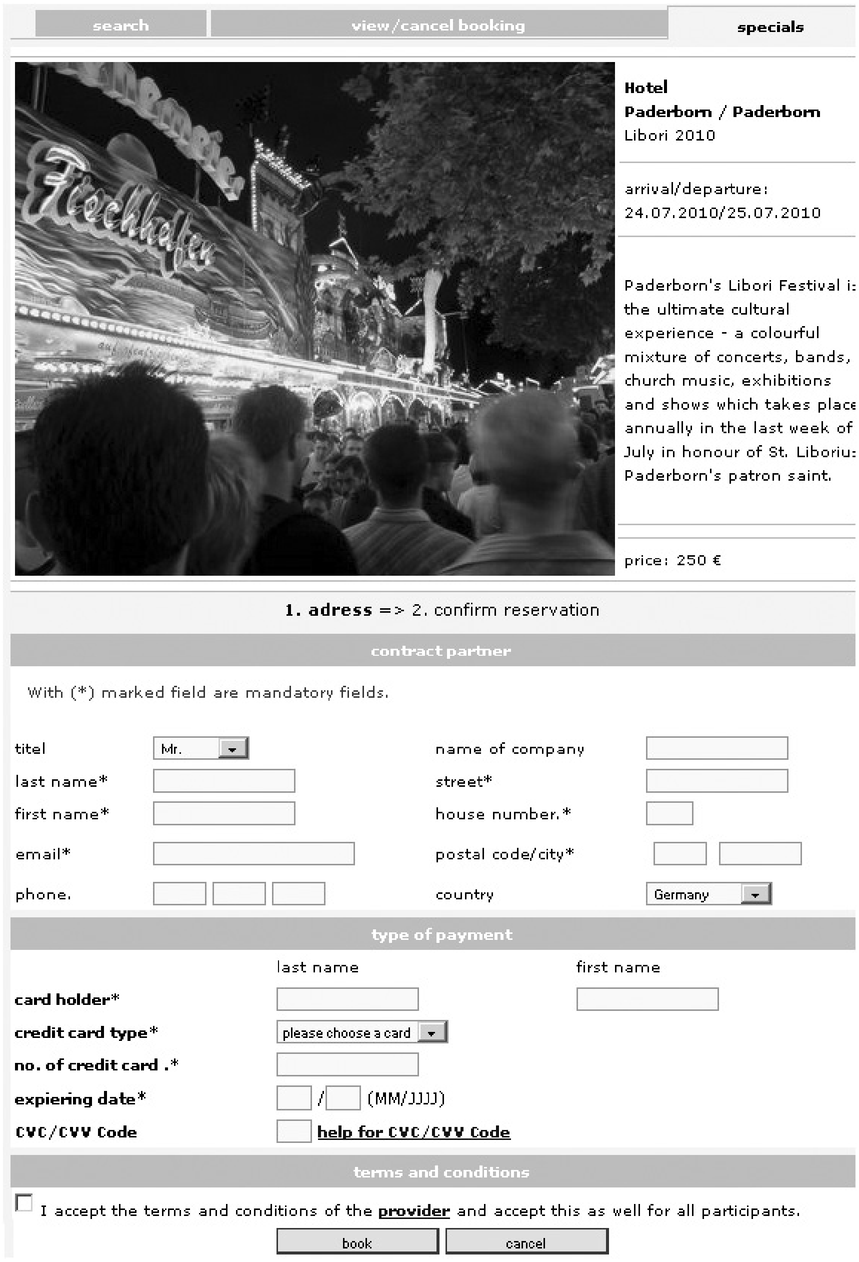
Acknowledgments

We thank Dr.-Ing. Axel Hollmann, Andagon GmbH, Germany, and M.Sc. Can Arda Muftuoglu, M.O.S.S. GmbH, Germany, for valuable hints and discussions.

Appendix A. Class Diagram of *DT*



Appendix B. Booking Mask for "Special Offers" in ISELTA



Hotel
Paderborn / Paderborn
Libori 2010

arrival/departure:
24.07.2010/25.07.2010

Paderborn's Libori Festival is the ultimate cultural experience - a colourful mixture of concerts, bands, church music, exhibitions and shows which takes place annually in the last week of July in honour of St. Liborius Paderborn's patron saint.

price: 250 €

1. adress => 2. confirm reservation

contract partner

With (*) marked field are mandatory fields.

titel	Mr.	name of company	
last name*		street*	
first name*		house number.*	
email*		postal code/city*	
phone.		country	Germany

type of payment

	last name	first name
card holder*		
credit card type*	please choose a card	
no. of credit card .*		
expiering date*	/ (MM/JJJJ)	
CVC/CVV Code	**help for CVC/CVV Code**	

terms and conditions

☐ I accept the terms and conditions of the **provider** and accept this as well for all participants.

Appendix C. Input form to "Enter Specials" in ISELTA

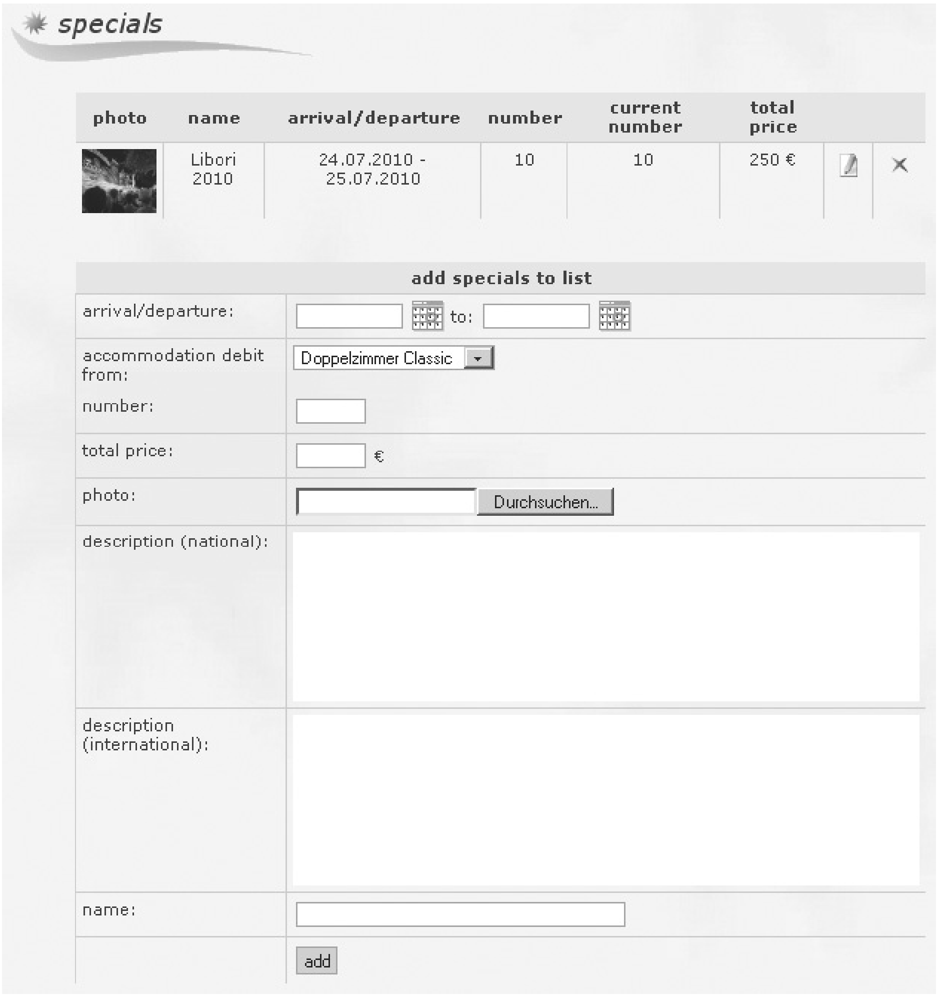
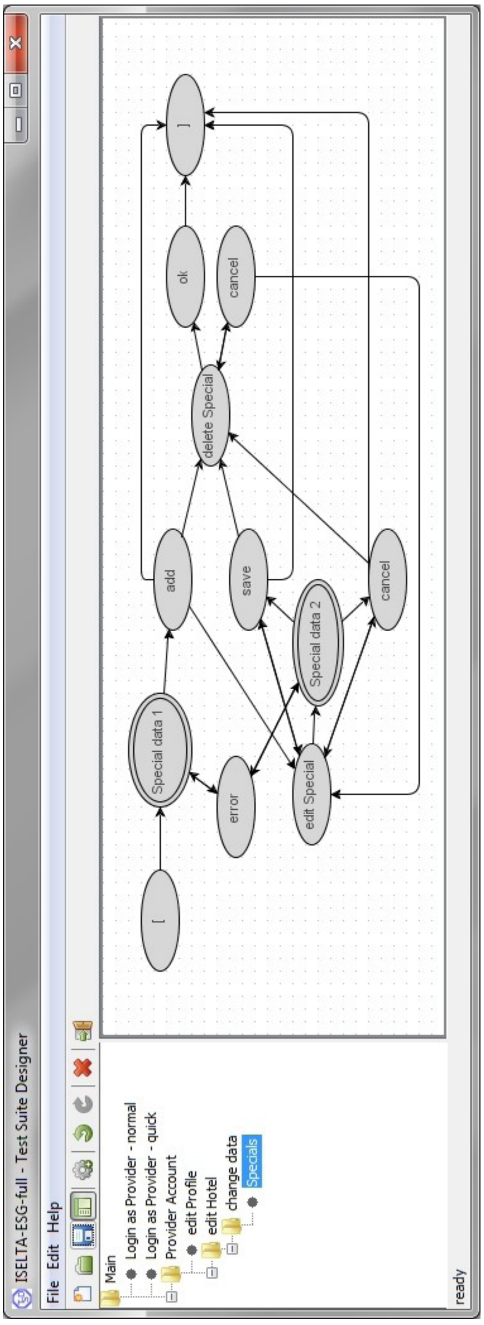

specials

photo	name	arrival/departure	number	current number	total price		
	Libori 2010	24.07.2010 - 25.07.2010	10	10	250 €		×

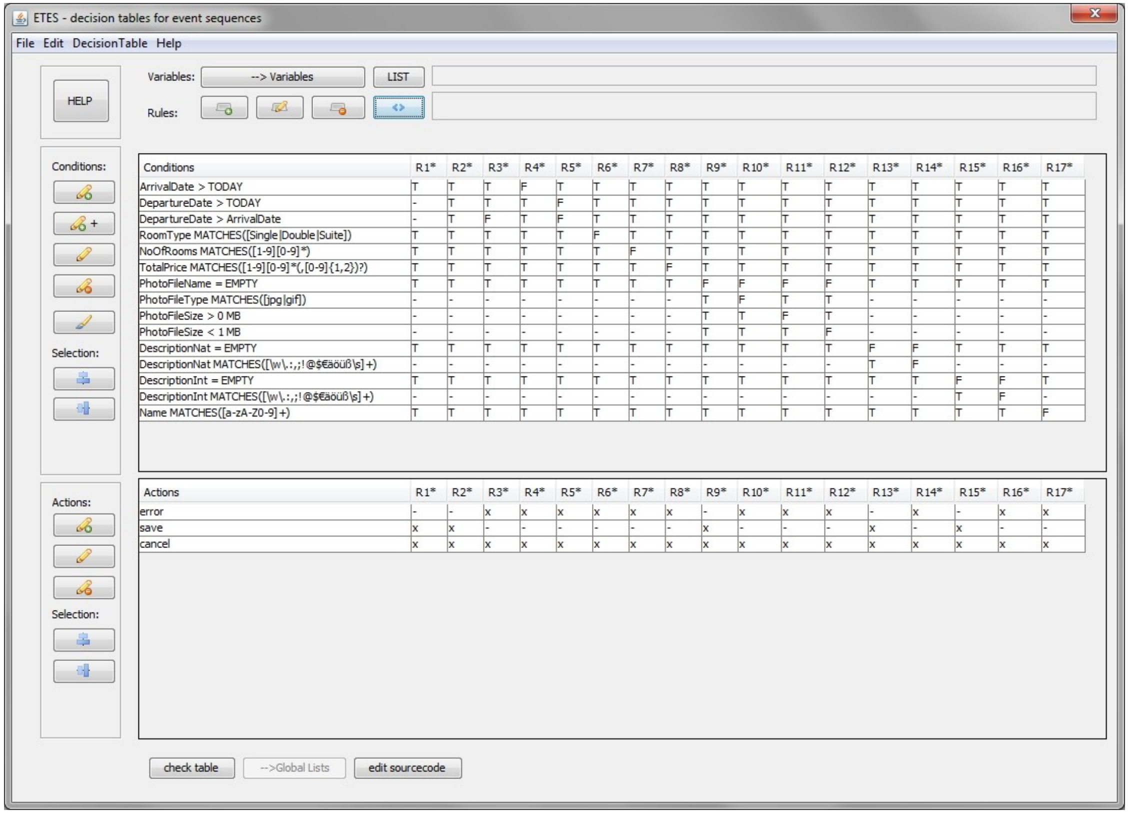
add specials to list

arrival/departure:	to:
accommodation debit from:	Doppelzimmer Classic
number:	
total price:	€
photo:	Durchsuchen...
description (national):	
description (international):	
name:	
	add

Appendix D. Input Contract Testing tool: Model Browser GUI for "Enter Specials" in ISELTA



Appendix E. Input Contract Testing Tool: Decision Table GUI for "Enter Specials" in ISELTA



Conditions	R1*	R2*	R3*	R4*	R5*	R6*	R7*	R8*	R9*	R10*	R11*	R12*	R13*	R14*	R15*	R16*	R17*
ArrivalDate > TODAY	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T	T	T
DepartureDate > TODAY	-	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T	T
DepartureDate > ArrivalDate	-	T	F	T	F	T	T	T	T	T	T	T	T	T	T	T	T
RoomType MATCHES([Single|Double|Suite])	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T	T	T
NoOfRooms MATCHES([1-9][0-9]*)	T	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T	T
TotalPrice MATCHES([1-9][0-9]*(,[0-9]{1,2})?)	T	T	T	T	T	T	T	F	T	T	T	T	T	T	T	T	T
PhotoFileName = EMPTY	T	T	T	T	T	T	T	T	F	F	F	F	T	T	T	T	T
PhotoFileType MATCHES([jpg|gif])	-	-	-	-	-	-	-	-	T	F	T	T	-	-	-	-	-
PhotoFileSize > 0 MB	-	-	-	-	-	-	-	-	T	T	F	T	-	-	-	-	-
PhotoFileSize < 1 MB	-	-	-	-	-	-	-	-	T	T	T	F	-	-	-	-	-
DescriptionNat = EMPTY	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T	T	T
DescriptionNat MATCHES([\w\.:,;!@$€äöüß\s]+)	-	-	-	-	-	-	-	-	-	-	-	-	T	F	-	-	-
DescriptionInt = EMPTY	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	T
DescriptionInt MATCHES([\w\.:,;!@$€äöüß\s]+)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	T	F	-
Name MATCHES([a-zA-Z0-9]+)	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F

Actions	R1*	R2*	R3*	R4*	R5*	R6*	R7*	R8*	R9*	R10*	R11*	R12*	R13*	R14*	R15*	R16*	R17*
error	-	-	x	x	x	x	x	x	-	x	x	x	-	x	-	x	x
save	x	x	-	-	-	-	-	-	x	-	-	-	x	-	x	-	-
cancel	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

References

1. A. V. Aho, A. T. Dahbura, D. Lee and M. U. Uyar, An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours, *IEEE Trans. Commun.* **39**(11) (1991) 1604–1615.
2. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord and K. Wallnau, Technical concepts of component-based software engineering, Defense Technical Information Center, 2000.
3. B. Beizer, *Software Testing Techniques*, *2nd ed.* (Van Nostrand Reinhold, New York, 1990).
4. F. Belli, Finite state testing and analysis of graphical user interfaces, in *Proceedings of 12th International Symposium on Software Reliability Engineering*, 2001, pp. 34–43.
5. F. Belli and C. J. Budnik, Test minimization for human-computer interaction, *Applied Intelligence* **26**(2) (2007) 161–174.
6. F. Belli, C. J. Budnik and L. White, Event based modelling, analysis and testing of user interactions: Approach and case study, *Software Testing, Verification and Reliability* **16**(1) (2006) 3–32.
7. F. Belli, N. Guler and M. Linschulte, Does "Depth" Really Matter? On the Role of Model Refinement for Testing and Reliability, in *35th Annual Computer Software and Applications Conference*, 2011, pp. 630–639.
8. F. Belli, N. Güler and M. Linschulte, Are longer test sequences always better? — A reliability theoretical analysis, in *Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, 2010, pp. 78–85.
9. F. Belli and M. Linschulte, On negative tests of web applications, *Annals of Mathematics, Computing & Teleinformatics* **1**(5) (2008) 44–56.
10. R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley Professional, 2000).
11. L. C. Briand, Y. Labiche and H. Sun, Investigating the use of analysis contracts to improve the testability of object-oriented code, *Software: Practice and Experience* **33**(7) (2003) 637–672.
12. I. Ciupa, A. Pretschner, A. Leitner, M. Oriol and B. Meyer, On the predictability of random tests for object-oriented software, in *1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 72–81.
13. I. Ciupa, Strategies for random contract-based testing, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 18143, 2008.
14. I. Ciupa and A. Leitner, Automatic testing based on design by contract, in *Proceedings of Net.ObjectDays*, 2005, pp. 545–557.
15. P. Collet, R. Rousseau, T. Coupaye and N. Rivierre, A contracting system for hierarchical components, *Component-Based Software Engineering*, 2005, pp. 187–202.
16. J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *J. ACM* **19**(2) (1972) 248–264.
17. S. H. Edwards, G. Shakir, M. Sitaraman, B. W. Weide and J. Hollingsworth, A framework for detecting interface violations in component-based software, in *Proceedings of Fifth International Conference on Software Reuse*, 1998, pp. 46–55.
18. R. Gove and J. Faytong, Machine learning and event-based software testing: Classifiers for identifying infeasible GUI event sequences, *Adv. Comput.* **86** (2012) 109–135.
19. F. Gross, G. Fraser and A. Zeller, Search-based system testing: High coverage, no false alarms, in *International Symposium on Software Testing and Analysis*, 2012, pp. 67–77.
20. P. Guerreiro, Simple support for design by contract in C++, in *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, 2001, pp. 24–34.

21. J. H. Hayes and J. Offutt, Input validation analysis and testing, *Empirical Software Engineering* **11**(4) (2006) 493–522.
22. R. Heckel and M. Lohmann, Towards contract-based testing of web services, *Electronic Notes in Theoretical Computer Science* **116** (2005) 145–156.
23. R. Helm, I. M. Holland and D. Gangopadhyay, Contracts: Specifying behavioral compositions in object-oriented systems, 1990.
24. S. Huang, M. B. Cohen and A. M. Memon, Repairing GUI test suites using a genetic algorithm, in *Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 245–254.
25. M. R. Karam, S. M. Dascalu and R. H. Hazimé, Challenges and opportunities for improving code-based testing of graphical user interfaces, *J. Computational Methods Sci. and Eng.* **6** (2006) 379–388.
26. Z. Kiziltan, T. Jonsson and B. Hnich, On the definition of concepts in component based software development, Department of Information Science, Uppsala University, 2000.
27. R. Kramer, iContract-the Java TM design by contract TM tool, in *Proceedings Technology of Object-Oriented Languages*, 1998, pp. 295–307.
28. B. Kruger and M. Linschulte, Cost reduction through combining test sequences with input data, in *IEEE Sixth International Conference on Software Security and Reliability Companion*, 2012, pp. 207–216.
29. B. P. Lamancha, M. Polo, D. Caivano, M. Piattini and G. Visaggio, Automated generation of test oracles using a model-driven approach, *Information and Software Technology* **55**(2) (2013) 301–319.
30. H. Liu and Kuan H. B. Tan, Covering code behavior on input validation in functional testing, *Information and Software Technology* **51**(2) (2009) 546–553.
31. C. Mao, Y. Lu and J. Zhang, Regression testing for component-based software via built-in test design, in *Proceedings of the ACM Symposium on Applied Computing*, 2007, pp. 1416–1421.
32. A. Marchetto and P. Tonella, Search-based testing of Ajax web applications, in *1st International Symposium on Search Based Software Engineering*, 2009, pp. 3–12.
33. A. M. Memon, An event-flow model of GUI-based applications for testing, *Software Testing, Verification and Reliability* **17**(3) (2007) 137–157.
34. A. M. Memon, M. E. Pollack and M. Lou Soffa, Automated test oracles for GUIs, *ACM SIGSOFT Software Engineering Notes*, 2000, pp. 30–39.
35. A. M. Memon, M. Lou Soffa and M. E. Pollack, Coverage criteria for GUI testing, *ACM SIGSOFT Software Engineering Notes* **26**(5) (2001) 256–267.
36. B. Meyer, Applying design by contract, *Computer* **25**(10) (1992) 40–51.
37. Y. Miao and X. Yang, An FSM based GUI test automation model, in *11th International Conference on Control Automation Robotics & Vision*, 2010, pp. 120–126.
38. A. J. Offutt, Y. Xiong and S. Liu, Criteria for generating specification-based tests, in *Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 1999, pp. 119–129.
39. J. Offutt, S. Liu, A. Abdurazik and P. Ammann, Generating test data from state-based specifications, *Software Testing, Verification and Reliability* **13**(1) (2003) 25–53.
40. J. Offutt, Y. Wu, X. Du and H. Huang, Web application bypass testing, in *Proceedings of the 28th Annual International Computer Software and Applications Conference*, 2004, pp. 106–109.
41. A. C. R. Paiva, J. C. P. Faria, N. Tillmann and R. A. M. Vidal, A model-to-implementation mapping tool for automated model-based GUI testing, in *Formal Methods and Software Engineering*, 2005, pp. 450–464.

42. A. Petrenko, A. Simao and J. Maldonado, Model-based testing of software and systems: Recent advances and challenges, *Int. J. Softw. Tools Technol. Transf.* **14**(4) (2012) 383–386.
43. R. Plosch, Design by contract for Python, in *Proceedings of Software Engineering Conference, Asia Pacific and International Computer Science Conference*, 1997, pp. 213–219.
44. R. Plosch and J. Pichler, Contracts: From analysis to C++ implementation, in *Proc. Technology of Object-Oriented Languages and Systems*, 1999, pp. 248–257.
45. R. Rao, Implementational reflection in Silica, in *European Conference on Object-Oriented Programming*, 1991, pp. 251–267.
46. A. Rauf, S. Anwar, M. A. Jaffer and A. A. Shahid, Automated GUI test coverage analysis using GA, in *Seventh International Conference on Information Technology: New Generations*, 2010, pp. 1057–1062.
47. D. J. Richardson, TAOS: Testing with analysis and oracle support, in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1994, pp. 138–153.
48. S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik and D. D. Edwards, *Artificial Intelligence: A Modern Approach*, Vol. 2 (Prentice Hall, Englewood Cliffs, 1995).
49. A.-M. Sassen, G. Amorós, P. Donth, K. Geihs, J.-M. Jézéquel, K. Odent, N. Plouzeau and T. Weis, QCCS: A methodology for the development of contract-aware components based on aspect oriented design, in *AOSD Early Aspects Workshop*, 2002.
50. J. L. Silva, J. C. Campos and A. C. R. Paiva, Model-based user interface testing with spec explorer and concurtasktrees, *Electronic Notes in Theoretical Computer Science* **208** (2008) 77–93.
51. A. Simão, A. Petrenko and N. Yevtushenko, On reducing test length for FSMs with extra states, *Softw. Testing, Verif. Reliab.* **22**(6) (2012) 435–454.
52. Y. Le Traon, B. Baudry and J.-M. Jézéquel, Design by contract to improve software vigilance, *IEEE Trans. Software Engineering* **32**(8) (2006) 571–586.
53. T. Tuglular, C. A. Muftuoglu, F. Belli and M. Linschulte, Event-based input validation using design-by-contract patterns, in *20th International Symposium on Software Reliability Engineering*, 2009, pp. 195–204.
54. D. Wampler, Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces, in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2006, pp. 27–30.
55. B. Wan, G. V. Bochmann and G. Jourdan, Evaluating reliability-testing usage models, in *36th Annual Computer Software and Applications Conference*, 2012, pp 129–137.
56. J. A. Whittaker, Software's invisible users, *Software* **18**(3) (2001) 84–88.
57. M. Ye, B. Feng and L. Zhu, Automated oracle based on multi-weighted neural networks for GUI testing, *Information Technology Journal* **6**(3) (2007).
58. W. Zheng and G. Bundell, Test by contract for UML-based software component testing, in *International Symposium on Computer Science and Its Applications*, 2008, pp. 377–382.