PAPER

# Model-Based Contract Testing of Graphical User Interfaces

Tugkan TUGLULAR[†a)], *Member*, Arda MUFTUOGLU[††], Fevzi BELLI[†,†††], *and* Michael LINSCHULTE[††††], *Nonmembers*

**SUMMARY** Graphical User Interfaces (GUIs) are critical for the security, safety and reliability of software systems. Injection attacks, for instance via SQL, succeed due to insufficient input validation and can be avoided if contract-based approaches, such as Design by Contract, are followed in the software development lifecycle of GUIs. This paper proposes a model-based testing approach for detecting GUI data contract violations, which may result in serious failures such as system crash. A contract-based model of GUI data specifications is used to develop test scenarios and to serve as test oracle. The technique introduced uses multi terminal binary decision diagrams, which are designed as an integral part of decision table-augmented event sequence graphs, to implement a GUI testing process. A case study, which validates the presented approach on a port scanner written in Java programming language, is presented.
***key words:*** *model-based testing, GUI testing, event sequence graphs, multi terminal binary decision diagrams*

## 1. Introduction

Input validation testing aims to show the existence or non-existence of certain defects by referring to the robustness of the software under consideration [1]. The goal of the present approach is to build test scenarios from a model, which is developed from the specifications of the graphical user interface requirements [2]. The graph model chosen for user-software communication is event-based, where vertices of the graph, that is, event transition diagram, are designated to inputs and system responses. The arcs and paths resemble event sequences, to build up the *event sequence graph* (*ESG*), that enable testers to formally seize the nature of interactive systems. This paper suggests the nodes of ESG be advanced by decision tables (DTs) to represent Boolean algebraic constraints on input data of a system under consideration (SUC) [3]. More important, the approach improves the existing approach by integrating design by contract (DbC) patterns to form contracts and by utilizing multi-terminal binary decision diagrams (MTBDDs) to generate test cases from decision table rules for input validation.

Preventing the input from ever getting to the application in the first place is possible only at the interfaces, such as GUIs. GUIs should be specifically designed to filter unwanted input, which can be achieved through GUI data contracts that are defined and used in this work. A model-based specification of GUI data contracts is achieved through DT-augmented ESGs.

The functional testing approach proposed performs at the GUI level, where tests are derived from contracts by creation of test (case) input values and test oracles. Although there are some contract-based testing techniques, the approach presented in this work for GUI testing is novel for several reasons. First, the proposed methodology is based on contracts, which provide information related to the interface semantics of GUIs. As noted by Ciupa and Leitner [4], the validity of a software component can be ascertained by checking the software with respect to its contracts that have not yet been applied to GUI testing.

The second novelty of the paper stems from enabling the tester to provide formal specifications in the form of DT rules. Contract-based specifications in this format are more compact than the corresponding test code, and easy to understand and maintain. Generally, specification errors can be revealed by testing approaches that utilize *direct* specifications [5]. For instance, specifications were utilized to find boundary overflow vulnerabilities in the previous work of the authors [6]. In addition to using contracts to automatically generate test input values, contracts can be used as test oracles as they define valid and invalid conditions for the software. Thus, utilization of contracts eliminates the necessity of developing a test oracle for each test case [7]. The novelty of the presented approach can be summarized as follows:

- The term "GUI data contract" is introduced and explored using the application analyzed in the case study. GUI data contracts are based on application domain knowledge instead of software implementation of GUI. The use of MTBDD for modeling data contract is the main contribution.
- A new GUI data contract testing approach is introduced along with its novel test case generation algorithm. The algorithm generates test cases from a contract MTBDD using equivalence class partitioning and boundary value approaches. Solutions to problems of test coverage criteria and test oracle within the context of GUI data contract testing are also presented.

Manuscript received October 31, 2014.
Manuscript revised February 16, 2015.
Manuscript publicized March 19, 2015.
[†]The authors are with Izmir Institute of Technology, Izmir, 35430 Turkey.
[††]The author is with M.O.S.S. Computer Grafik Systeme GmbH, 82024 Munich, Germany.
[†††]The author is with University of Paderborn, Germany.
[††††]The author is with Andagon GmbH, 50933 Köln, Germany.
a) E-mail: tugkantuglular@iyte.edu.tr
DOI: 10.1587/transinf.2014EDP7364

- The approach has been validated by means of a case study on a port scanner, written in Java, and the results obtained are summarized.

Section 2 summarizes related work. Section 3 outlines the theoretical background on ESGs and MTBDDs. The core of the article, Sect. 4, represents the introduced GUI data contracts concept with ESGs and MTBDDs. Section 5 provides the GUI data contract testing approach, which builds it on DT-augmented ESG approach for a more comprehensive GUI testing process. A case study is performed in Sect. 6 and its results are analyzed to validate the proposed approach and discuss its characteristics feature. Section 7 concludes the paper and outlines the planned work.

## 2. Related Work

The approach is related to GUI testing, input validation, contracts, and contract-based testing.

Memon et al. [8] define *GUI* as a hierarchical and graphical artefact of a software for accepting user-generated events as inputs and deterministically producing corresponding outputs. *Input validation* examines the syntax and partly semantics of input entered through graphical user interface [9]. Errors due to input validation may cause malfunctions of software and create vulnerabilities for attacks [10]. For this reason, there are various specification-based test approaches for validation of user interfaces [1]. ESGs [2], event flow graphs [11], and their extension, that are, event dependency graphs [12], can be used for validation of user interface specifications even before starting their implementation and testing [13].

Interface signatures are not sufficient to validate the functional and non-functional, quality aspects of a software [14] that necessitates involving more sophisticated specification techniques [15]. Contracts, a technique describing behaviors and obligations of the participating objects [16], are candidates for this purpose. Le Traon et al. [17] defines *contract* as a set of assertions, which are checked before and after the execution of a method. There is one contract for each method in the program that is composed of a pre- and a postcondition and of the invariant of the class. Meyer [18] used contracts in Eiffel language at a low level of abstraction.

Wampler [19] explains DbC as a method of presenting the contract of an interface in an automatically testable way.

Vinoski [20] proposes that data contracts should be part of any service contract of interfaces used in service-oriented architectures. Truong et al. [21] suggest an abstract model for data contracts, and based on this abstract model, they propose techniques for evaluating data contracts that can be integrated into data service selection and composition frameworks.

The literature on testing based on formal specifications [22]–[24] is rich. 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 test oracle [25]. Binder [26], for example, discusses the idea of using assertions based on contracts as test oracles. Zheng et al. [27] introduced contract idea for testing called Test by Contract. Similarly, a contract-based testing approach for testing of web service was presented in [28].

To sum up, a general methodology for the generation of test oracles does not exist, therefore they are frequently complex and error prone [26], [29].

The approach presented in this work differs from the ones mentioned above in that it presents a novel GUI data contract testing approach including a sound solution to oracle problem. It also differs from previous work of authors [6], which proposes a method combining model-based GUI testing with static analysis for fault localization. It aims to minimize number of test cases for acceptance testing whereas the method proposed in [6] aims to maximize number of test cases for fault localization. It follows path coverage based on MTBDD whereas the method proposed [6] follows rule coverage based on decision table.

## 3. Theoretical Background

While testing a system, a model of the system helps to predict and control its behavior. Modeling a system acquires the understanding of its abstraction, and in the case of testing GUIs, there is the need of a formal specification technique for distinguishing between *legal* and *illegal* situations. These requirements are fulfilled by ESGs.

Apart from the notion of finite state automata (FSA), in ESG, the simplification by merging the inputs and states helps the test engineer to easily understand and check the external behavior of the system, hence the “inputs” and “states” are turned into “events”.

**Definition 1**: 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), $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$, and $\gamma \in \Gamma$, called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \dots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$, $k \in N_0$, whereby $N_0$ represents the set of the natural numbers including 0 (zero), 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 $v \neq \xi, \gamma$.

$\Xi$ *(ESG)*, $\Gamma$ *(ESG)* represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex ‘[’ $\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex ‘]’ $\notin V$. The semantics of an ESG is as follows: Any $v \in V$ represents an event. For two events $v, v' \in V$, the event $v'$ must be enabled after the execution of $v$ if and only if $(v, v') \in E$. The operations on identifiable components of the GUI are controlled and/or perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of $V$ and lead interactively to a succession of user inputs and expected desirable system outputs.

**Definition 2**: Let $V$, $E$ be defined as in Definition 1. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$, $k \in N_0$, whereby $N_0$ represents the set of the natural numbers including 0 (zero), is called an *event sequence (ES)* if and only if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$. Moreover, an ES is *complete* (or, it is called a *complete event sequence*, *CES*), if and only if $v_0 \in \Xi$ and $v_k \in \Gamma$. ■

Note that the pseudo vertices '[', ']' are not included in ESs. An ES = $\langle v_i, v_k \rangle$ of length 2 is called an *event pair (EP)*. 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.

Modeling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model. To avoid this, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled (see Fig. 2).

Decision tables [30] are popular in information processing and are also used for testing, e.g., in cause and effect graphs. A decision table logically links conditions ("if") with actions ("then") that are to be triggered, depending on combinations of conditions ("rules") [3].

**Definition 3**: A Decision Table $DT = \{C, A, R\}$ represents actions that depend on certain conditions where:
$C \neq \emptyset$ is the set of conditions,
$A \neq \emptyset$ is the set of actions,
$R \neq \emptyset$ is the set of rules that describe executable actions depending on a certain combination of constraints.

A *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, A_x)$ where:

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

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ under regular circumstances. In certain cases it is inevitable to remark 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 GUIs.

MTBDDs, also called algebraic decision diagrams, are generalizations of binary decision diagrams (BDDs) from Boolean values to values of any finite domain [31].

**Definition 4**: Let $D$ be a finite set and *Var* be a finite set of Boolean variables equipped with a total ordering $< \subset Var \times Var$. A multi terminal binary decision diagram (MTBDD) over $(Var, <)$ is a rooted acyclic directed graph with vertex set $V$ and the following labelling: Each terminal vertex $v$ is labelled by an element of $D$, denoted by *value(v)*. Each non-terminal vertex $v$ is labelled by a variable $var(v) \in Var$ and has two children $then(v), else(v) \in V$. In addition the labelling of the non-terminal vertices by variables respect the given ordering <, i.e. $var(then(v)) > var(v) < var(else(v))$ for all non-terminal vertices $v$. ■

The edge from $v$ to *then(v)* represents the case where *var(v)* is true; conversely the edge from $v$ to *else(v)* the case where *var(v)* is false.

**Definition 5**: A MTBDD M is called *reduced* iff

- for each *non-terminal vertex v* the two children are distinct, i.e. $then(v) \neq else(v)$. Each *terminal vertex v* has a distinct *value(v)*.
- for all vertices $v$, $v'$ with the same labeling, if the subgraphs with root $v$ and $v'$ respectively are isomorphic (i.e. coincide up to the names of the services) then $v = v'$. Formally, if $var(v) = var(v')$ and $else(v) = else(v')$ and $then(v) = then(v')$, then $v = v'$. ■

Reduced MTBDDs effectively represent DTs as a graph, which is used to generate test cases in the presented approach.

## 4. Data Contracts for Graphical User Interfaces

A *contract* defines contractual obligations between two of communicating parties. *Contractual obligations* consist of data and operation obligations, where each participant must provide necessary data and related sequence of actions. In this case, participants are users and GUI, and they conform to the contract to be able to interact with each other.

As an introductory example, Table 1 shows the informal contract specification for JAPS Java Port Scanner [32] GUI shown in Fig. 1, which is the application analyzed in the case study.

The data contract of JAPS' GUI should contain the following constraints:

- The user of JAPS application promises that the value to be entered for any port is in [0..65535] and *begin_port* is lower than or equal to *end_port*.
- Private ports [33] from 49152 through 65535 are commonly used by the operating system kernels. Finding any port in that range open may mean nothing for a single scan and therefore not necessary to scan.
- The GUI promises on response that open ports are displayed.
- If the user enters an invalid value for ports, the best explanatory error or warning message is displayed.

GUI data contracts are defined as contracts established on the GUI component of SUC between GUI and user. The

**Table 1** Informal contract specification for GUI of JAPS port scanner.

| Participants | Obligations | Benefits |
|---|---|---|
| User | Make sure that entered value for any port is valid, *begin_port* is lower than or equal to *end_port*, and ports are meaningful | Optionally learn that conditions are satisfied or get error message indicating what is wrong |
| GUI | Optionally inform user that conditions are satisfied and call the function that scans given range of ports | Give error message if value for any port is invalid, if *begin_port* is higher than *end_port*, or ports are not meaningful |

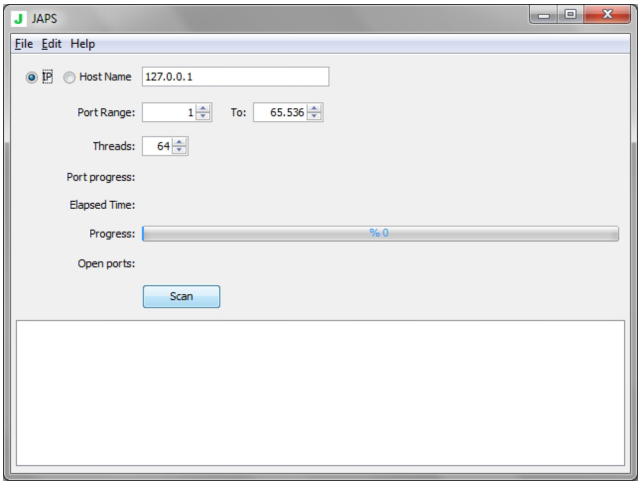


**Fig. 1** GUI of JAPS port scanner [32].

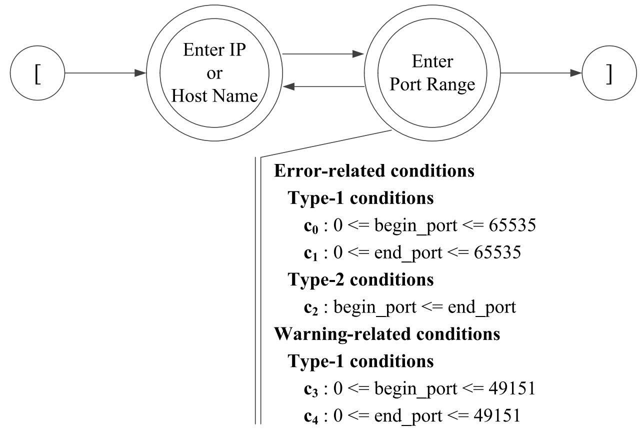


**Fig. 2** ESG for GUI of JAPS with data contract conditions.

condition types used in defining GUI data contracts given below are modified from [34]:

**Definition 6**: A GUI data contract has Type-1 and Type-2 conditions:

- Type-1 conditions define expected values with their allowed domain with respect to syntactic definitions and business rules.
- Type-2 conditions define expected relationships among values to be held not necessarily among input values but also among input values and state values of SUC. ■

If any of the above stated condition checks fail, an error or a warning message specified in the data contract should be presented to the user. The approach presented here suggests that the GUI should warn the user and show the right operation for an acceptable situation when there is a faulty input. Therefore, a complementary view is followed to also cover possible input errors in the model of the GUI under consideration. Error and warning messages is part of the GUI data contract. Therefore, conditions are also classified as error-related conditions and warning-related conditions. In the proposed approach, GUI is tested first for error-related conditions and then for warning-related conditions separately. The reason behind is that there might be a subsumes relation between an error-related condition and warning-related condition as seen between $c_0$ and $c_3$ as well as $c_1$ and $c_4$ given in Fig. 2.

In this work, data input and output operations of a GUI are refined by a DT that attributes the corresponding arcs of ESG that models the SUC. The constraints contained in the decision table are classified into *error* and *warning* conditions and then *Type-1*, and *Type-2* conditions as explained above.

GUI data contracts are represented using DT-augmented ESGs. As an example, the DT-augmented ESG for GUI of JAPS application is shown in Fig. 2.

Conditions from $c_0$ through $c_4$ for JAPS port scanner GUI of port inputs are shown in Fig. 2. Actions for JAPS port scanner GUI with respect to data contract for port range are specified as follows:

- $a_0$: deny input with error message “any port value should be in [0..65535]”.
- $a_1$: deny input with error message “begin_port value should be lower than or equal to end_port value”.
- $a_2$: deny input with error message of $A_0$ concatenated with error message of $A_1$.
- $a_3$: accept input with warning message “any port from 49152 through 65535 is used by operating systems. Finding any port in that range open may mean nothing for a single scan”.
- $a_4$: accept input.

## 5. Graphical User Interface Testing Using Data Contracts

In this work, contracts are used as a specification mechanism. Accordingly, the testing approach is contract-based. A test case specifies input values for an operation of GUI. A test suite is composed of test cases, which check all assertions offered by a data contract of GUI. The input values making up a test case can be derived from the conditions of provided data contract. Expected outputs are also created from data contract that are associated with the operations of GUI. The data conract of a GUI is represented by a DT, of which actions are “accept input” and “deny input” with error messages. Warnings may accompany “accept input” action.

The data contract-based GUI testing process is presented in Algorithm 1. First, DT-augmented ESG is developed using GUI data contract. Then, CESs (see Definition 2) are created by covering all events. In the next step, for each CES, contract-based test cases are generated. Finally, test suite is applied to GUI and outputs are checked for a faulty event.

**Algorithm 1** Data contract-based GUI testing process.

```
generate the DT-augmented ESG
cover all events by means of CESs
foreach CES
    generate contract-based test cases
apply the test suite to GUI
```

observe GUI output to check whether a faulty event occured

MTBDDs are used in generation of contact-based test cases. One advantage of using a MTBDD is that it can be reduced by eliminating nodes with same left and right children. This reduction is valuable since the number of test cases is minimized with respect to test coverage.

**Definition 7**: A *Contract-MTBDD* (C-MTBDD) is a reduced MTBDD $(C \cup A, E)$, whereby $C$ is the set of the non-terminal vertices, each holding one condition $c_i$ for $i \in \{0, 1, \ldots, n\}$, $A$ is the set of terminal vertices, each holding one action $a_j$ for $j \in \{0, 1, \ldots, k\}$, and $E$ is the set of edges connecting vertices, with $n$ and $k \in N_0$, whereby $N_0$ represents the set of the natural numbers including 0 (zero). Note that $A = \{A_{deny} \cup A_{accept}\}$, where $A_{deny}$ represents deny actions and $A_{accept}$ accept actions. The conditions in the C-MTBDD are ordered in such a way that a Type-2 condition follows Type-1 conditions, which contain the variables appearing in the Type-2 condition. ■

A C-MTBDD does not contain test input values but conditions with variables. These variables should be instantiated to obtain concrete test input values. Another advantage of using a C-MTBDD is that it solves test coverage problem at the same time, since traversing each path from root to terminal nodes covers whole graph, or diagram. Moreover, C-MTBDD enables testers to stop testing at the *input acceptance line*, if there are limited resources for testing and priority is given to *deny input* cases.

The contract-based test case generation process is presented in Algorithm 2. First, the DT is transformed to a MTBDD using *Generalized If Then Else* (*GITE*) algorithm [35]. Then, this MTBDD is separated into two sub-MTBDDs by the acceptance line, which clusters deny input actions and accept input actions. Such a MTBDD is presented in Fig. 3 for JAPS' GUI of port inputs. In the next step, each sub-MTBDD is reduced in itself following reduction rules defined by Bryant [36]. The resulting MTBDD is a *C-MTBDD*. An example of C-MTBDD is shown in Fig. 4.

To convert DT to MTBDD, an adjacency matrix, where the value 0 represents dashed edge and the value 1 represents solid edge, is used. No value means that there is no edge between two vertices. Since each rule in the decision table is a path in the MTBDD, when all paths are represented in the adjacency matrix, then the MTBDD is constructed from the adjacency matrix. In the next step, it is converted to C-MTBDD as explained above.

**Algorithm 2** Contract-based test case generation process.

```
foreach event with DT do
    convert DT into C-MTBDD
    foreach path in C-MTBDD
        generate test data by solving CSP
        using equivalence class boundary values
endfor
```

Attempting to find values for variables to satisfy the C-MTBDD path predicate is a special case of a *constraint satisfaction problem* (*CSP*) [37]. The function for solving CSP uses boundary values from valid and invalid equivalence classes for each clause and searches the values that make the expression true. The runtime complexity of the whole algorithm mainly depends on this function, which has to be solved for each path of the C-MTBDD. In this work backtracking is combined with the techniques "arc consistency check" and "minimum remaining values" (see [37] for further information) to solve the given CSPs in the C-MTBDD.

The runtime complexity of the backtracking algorithm 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 $O(n^2 d^3)$ [37]. However in practice, the number of variables on a GUI is strictly limited due to usability restrictions. For instance, in the presented case study the number of variables of the JAPS' GUI is 5. Simultaneously, this limits also the corresponding constraints so that the runtime complexity of this algorithm can be neglected in the end. Furthermore, the search space for numerical values may be narrowed by considering only boundary values of equivalence classes. Finally, the function for solving CSP returns test case values for each path in the C-MTBDD.

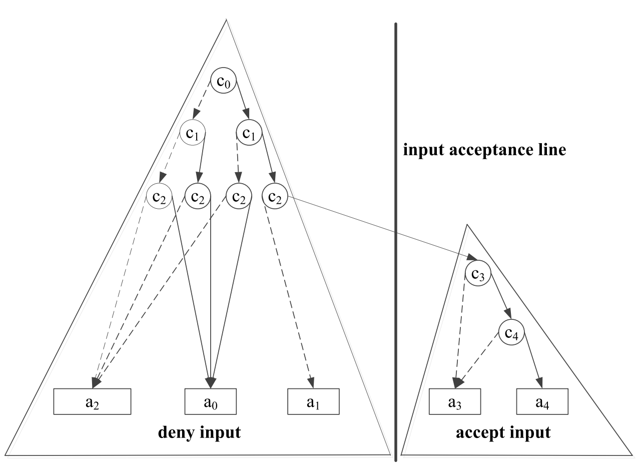


**Fig. 3** MTBDD for JAPS' GUI of port inputs.

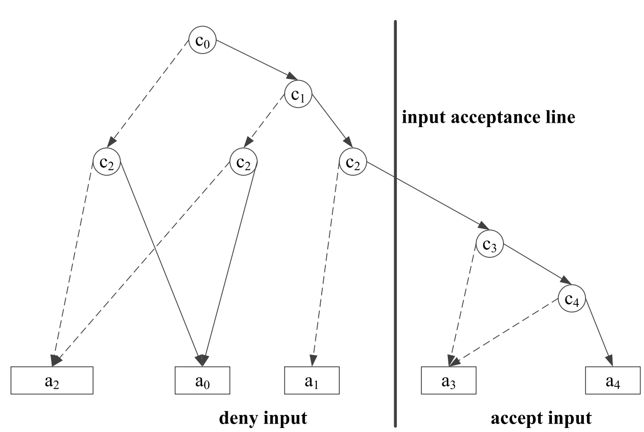


**Fig. 4** Contract MTBDD for JAPS' GUI of port inputs.

Since it is often not feasible to include all possible input values for a test case, the central question of testing becomes how to select a set of test input values most likely to uncover faults. This problem comes down to grouping data into equivalence classes, which should have the property that, if one value in the set causes a failure, then all other values in the set will cause the same failure, and conversely, if a value in the set does not cause a failure, then none of the others should cause a failure [25]. This property allows using only one value from each equivalence class as a representative for the class.

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. The presented technique strengthens this approach by boundary value selection of input values, which appear at the boundaries of equivalence classes [38]. In summary, to automatically generate concrete test cases from C-MTBDD, equivalence class boundary value selection method is employed.

This work proposes use of equivalence class partitioning with boundary value analysis to derive test oracles from the contracts in synchronization with the generation of test case values. Moreover, the test oracle here uses data contracts as means of checking test case result, hence test outputs can be easily compared with expected test results. This test oracle makes a pass/fail evaluation of the test case: if obtained results match the expected results, then the test case passes, otherwise it fails.

A tool that implements Algorithm 2 is developed in Python programming language using the following libraries:

- pyeda 0.26.0, which provides Symbolic Boolean algebra classes with a selection of function representations (https://pypi.python.org/pypi/pyeda).
- python_constraint 1.2, is a module implementing support for handling CSPs over finite domains (https://pypi.python.org/pypi/python-constraint).
- graphviz 0.4.2, which is a simple Python interface for Graphviz (https://pypi.python.org/pypi/graphviz).

The tool takes a decision table as input and then draws MTBDD using graphviz library. After that, it constructs C-MTBDD utilizing pyeda library. Finally, CSPs represented by each path are solved through python_constraint library.

## 6. Case Study, Experimental Analysis, Strengths and Limitations of the Approach

The presented approach is evaluated by means of the port scanner developed in Java, namely JAPS Java Port Scanner. Note that for the previous work [6] three port scanners written in C++ were evaluated. A port scanning software analyzes a single port or ports in a given range represented by *begin_port* and *end_port* to check for being open. The case study is exemplified using the JAPS port scanner, of which GUI is shown in Fig. 1. The case study is focused on the port input of the GUI. The DT-augmented ESG of the GUI of JAPS software is given in Fig. 2. JAPS Java Port Scanner is a simple but good example as a case study because its GUI data contract contains all possible constraints [34] that a contract based on variables can have.

The C-MTBDD of JAPS port scanner GUI, which is constructed from data contract given in Sect. 4, using the method explained in Sect. 5, is given in Fig. 4. For each path in the C-MTBDD, a test case value pair—since there are two variables for the GUI under test—is instantiated using valid and invalid values, which are determined by equivalence class partitioning with boundary value selection.

Each node in the C-MTBDD represents a condition containing a single variable with two equivalence classes, namely valid equivalence class $EC_v$ and invalid equivalence class $EC_{inv}$ [38]. For instance, integer variable *begin_port* should be in [0..65535] for condition $c_0$, which means $EC_v$ for begin_port is [0..65535] and invalid equivalence class is partioned for *begin_port*; one $EC_{inv-}$ is small in values $(-\infty..-1]$ and the other one $EC_{inv+}$ is large in values $[65536..+\infty)$. In case of testing for valid values defined by $EC_v$, one boundary value is 0 and another one is 65535 for integer variable *begin_port* in $c_0$. In case of testing for invalid values defined by $EC_{inv-}$ and $EC_{inv+}$, −1 and 65536 are considered respectively. In summary, $c_0.EC_v = \{0, 65535\}$ and $c_0.EC_{inv} = \{-1, 65536\}$ for $c_0$. For the test case generation algorithm presented here, $EC_v$ and $EC_{inv}$ for all conditions should be priorly defined.

The proposed contract-based test case generation algorithm takes a path of C-MTBDD starting from root ending at a terminal node and constructs a predicate for it. Then the algorithm tries to solve the predicate using values in $EC_v$ and $EC_{inv}$ sets. If there is a solution, then the values making the solution possible are assigned as test case values. The test case generation algorithm repeats this operation for each path existing in C-MTBDD.

For the leftmost path of C-MTBDD given in Fig. 4, the test case generation algorithm works as follows. The path predicate representing the leftmost path in the C-MTBDD is $\neg c_0 \wedge \neg c_2$. The actual predicate that is solved by the CSP solver is $\neg(0 <= \mathit{begin\_port} <= 65535) \wedge \neg(\mathit{begin\_port} <= \mathit{end\_port})$. Candidate test case values for *begin_port* in $c_0$ are priorly defined as $c_0.EC_v = \{0, 65535\}$ and $c_0.EC_{inv} = \{-1, 65536\}$. For $(\mathit{begin\_port}, \mathit{end\_port})$ pairs in $c_2$, equivalence classes are determined using interdependency relation as $c_2.EC_v = \{(0, 65535)\}$ and $c_2.EC_{inv} = \{(0, -1), (65536, 65535)\}$. Since value 65536 for $c_0$ and value pair $(65536, 65535)$ for $c_2$ presents a solution to path predicate $\neg c_0 \wedge \neg c_2$, the concluded test case for $(\mathit{begin\_port}, \mathit{end\_port})$ for the leftmost path of C-MTBDD is $(65536, 65535)$.

Executing test case generation algorithm for the path with action $a_1$ at its leaf results in a $(65535, 0)$ test case for $(\mathit{begin\_port}, \mathit{end\_port})$. Candidate test input values for *end_port* in $c_1$ are determined as $c_1.EC_v = \{0, 65535\}$ and $c_1.EC_{inv} = \{-1, 65536\}$. By checking the path expression $c_0 \wedge c_1 \wedge \neg c_2$, it can be found that $(65535, 0)$ test case for $(\mathit{begin\_port}, \mathit{end\_port})$ makes the path expression true. For

**Table 2** Application of test cases to GUI of JAPS port scanner.

| # | Test Case (begin_port, end_port) | GUI Output | Violation of any Condition? | Gives Error or Warning Message? |
|---|---|---|---|---|
| 1 | (65536,65535) | gives Error Message 1 | Yes | Error Message |
| 2 | (-1,0) | scans 65535.. 65536 | Yes | No |
| 3 | (0,-1) | scans 65535.. 65536 | Yes | No |
| 4 | (65535,65536) | scans 65535.. 65536 | Yes | No |
| 5 | (65535,0) | scans 65535.. 65536 | Yes | No |
| 6 | (-1,49151) | gives Error Message 1 | Yes | Error Message |
| 7 | (0,49152) | gives Error Message 1 | Yes | Error Message |
| 8 | (0,49151) | gives Error Message 1 | No | Error Message |

the rightmost path $c_0 \wedge c_1 \wedge c_2 \wedge c_3 \wedge c_4$ in the C-MTBDD, $(0, 49151)$ test case for $(begin\_port, end\_port)$ is found by checking the following candidate test case values:

- for *begin_port* in $c_0$: $c_0.EC_v = \{0, 65535\}$ and $c_0.EC_{inv} = \{-1, 65536\}$.
- for *end_port* in $c_1$: $c_1.EC_v = \{0, 65535\}$ and $c_1.EC_{inv} = \{-1, 65536\}$.
- for $(begin\_port, end\_port)$ pair in $c_2$: $c_2.EC_v = \{(0, 65535)\}$ and $c_2.EC_{inv} = \{(0, -1), (65536, 65535)\}$.
- for *begin_port* in $c_3$: $c_3.EC_v = \{0, 49151\}$ and $c_3.EC_{inv} = \{-1, 49152\}$.
- for *end_port* in $c_4$: $c_4.EC_v = \{0, 49151\}$ and $c_4.EC_{inv} = \{-1, 49152\}$.

The proposed contract-based test case generation algorithm produces the *begin_port* and *end_port* test case value pairs, which are given at the second column of Table 2.

GUI of the JAPS port scanner is tested with generated test case values. The GUI outputs are captured and recorded at Table 2. Table 2 displays the test case value pairs, GUI outputs, results of test cases (violation of any constraint or not) with respect to data contract, any error or warning messages given by the GUI, and faults if found. The application gives only one error message, namely Error Message 1: "The format of the IP you supplied or the begin and end port are wrong!" as indicated in Table 2, although there should be three error messages and one warning message.

There are 7 faulty input pairs but the program behaves as only 4 of them were not faulty and does not stop processing the related task, which results in erroneous operation. Moreover, for the last test case in Table 2 the program behaves as if it is faulty, although it is not.

Path coverage testing using reduced MTBDD instead of row coverage testing using decision table reduces generated number of test cases considerably with the same power of uncovering faults. The reduction in the number of test cases for this case study is from 12 (number of paths in MTBDD of Fig. 3) to 8 (number of paths in reduced MTBDD of Fig. 4).

The generated test suite, which is given in Table 2, is applied to other four port scanners, namely Pscan, Multi-scan, Free Port Scanner, and Angry IP Scanner. The test results are presented in Table 3. The generated test suite revealed faults at each port scanner. One interesting case is that Pscan is frozen when 0 is applied as a port value. Another one is that Angry IP Scanner, although it does not accept 65536 as end_port value with error message "correct port range is between (1–65535)", accepts 0 and 65536 as a begin_port port value.

**Table 3** Application of test cases to five different port scanners.

| # | Test Case (begin_port, end_port) | Violation of any Condition? | JAPS | Pscan | Multi scan | Free Port Scanner | Angry IP Scanner |
|---|---|---|---|---|---|---|---|
| 1 | (65536,65535) | Yes | Yes 1 | No | No | No | No |
| 2 | (-1,0) | Yes | No | Yes 2 | Yes 2 | Yes 2 | Yes 2,3 |
| 3 | (0,-1) | Yes | No | Yes 2 | Yes 2 | Yes 2 | Yes 2 |
| 4 | (65535,65536) | Yes | No | No | No | No | Yes 3 |
| 5 | (65535,0) | Yes | No | No | No | No | Yes 3 |
| 6 | (-1,49151) | Yes | Yes 1 | Yes 2 | Yes 2 | Yes 2 | Yes 2 |
| 7 | (0,49152) | Yes | Yes 1 | Frozen | No | No | No |
| 8 | (0,49151) | No | Yes 1 | Frozen | No | No | No |

Error or Warning Messages are as follows:
1: The format of the IP you supplied or the begin and end port are wrong!
2: No message but does not allow to enter value -1.
3: correct port range is between (1-65535)

By considering GUI specification as contracts, reducing them through C-MTBDD and assigning values by solving CSP with equivalence class boundary values, the proposed contract-based test case generation algorithm divides the input space into specification-based partitions and selects special values from those partitions that are likely to fail by following *competent programmer assumption* [39]. Therefore, it is more likely that expected number of test cases required to trigger at least one failure, called F-measure [40], is lower for the proposed approach compared to random, adaptive random and anti-random test case generation as well as search-based test case generation techniques, which sweep the input space using different algorithms. However, the cost of generation a single test case is higher for the proposed approach compared to the above mentioned techniques, because after construction of C-MTBDD each path in C-MTBDD is represented as a CSP and solved to obtain each test case.

Case study results encourages the generalization of the fact that especially preconditions related to GUI input data are not considered during software development and thus countermeasure actions are not implemented. Therefore, tools such as the one introduced in [6] are strongly recommended for fault localization. By correcting deficient validation mechanisms in the software, similar undesirable situations can be minimized. A limitation of the proposed technique is that equivalence classes and boundary values for each constraint in the data conract should be prepared manually before building C-MTBDD. Another limitation is that a single change in data contract can affect whole C-MTBDD depending on the ordering.

## 7. Conclusion and Future Work

Based on event sequence graph notion, this paper introduces a formal model to represent GUI data contracts. This model enables a novel testing approach. Moreover, contracts are combined with DT-augmented ESGs for refining the approach. The proposed solution automatically generates test cases and test oracles using conditions defined for input validation. A port scanning software written in Java has been tested for the validation of the proposed approach. Test results show that the proposed approach is effective in detecting faults with respect to data contracts.

To sum up, main contributions of this work are:

- The GUI data contract concept is introduced and implemented using DT-augmented ESGs.
- The introduced data contract testing approach includes an algorithm to automatically generate test cases from an Contract-MTBDD constructed using the data contract and applying equivalence class partitioning and boundary value techniques.
- Finally, aspects of test coverage and test oracle problems within the context of GUI data contract testing are examined as well.

Extension of the presented approach is planned by considering refinement and inclusion operations on data contracts to simplify expression of complex GUI behavior, where refinement enables specialization of contracts and inclusion allows contracts to be built from sub-contracts [16]. With these contract operations, we plan to define and apply an approach for combining GUIs and testing them as a single unit.

**References**

[1] J.H. Hayes and A.J. Offutt, "Increased software reliability through input validation analysis and testing," IEEE 10th International Symposium on Software Reliability Engineering, pp.199–209, Washington, DC, USA, 1999.
[2] F. Belli, "Finite state testing and analysis of graphical user interfaces," IEEE 12th International Symposium on Software Reliability Engineering, pp.34–43, Washington, DC, USA, 2001.
[3] F. Belli and M. Linschulte, "On 'negative' tests of web applications," Annals of Mathematics, Computing and Teleinformatics, vol.1, no.5, pp.44–56, 2007.
[4] I. Ciupa and A. Leitner, "Automatic testing based on design by contract," 6th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, pp.545–557, 2005.
[5] Y. Cheon and G.T. Leavens, "A simple and practical approach to unit testing: The JML and JUNIT way," 16th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol.2374, pp.231–255, Springer-Verlag, 2002.
[6] T. Tuglular, C.A. Muftuoglu, F. Belli, and M. Linschulte, "Event-based input validation using design-by-contract patterns," IEEE 20th Annual International Symposium on Software Reliability Engineering (ISSRE 2009), pp.195–204, Mysuru, India, 2009.
[7] L.C. Briand, Y. Labiche, and H. Sun, "Investigating the use of analysis contracts to improve the testability of object-oriented code," Software Pract. Exper., vol.33, no.7, pp.637–672, 2003.
[8] A.M. Memon, M.L. Soffa, and M.E. Pollack, "Coverage criteria for GUI testing," ACM SIGSOFT Software Engineering Notes, vol.26, no.5, pp.256–267, 2001.
[9] J.H. Hayes and J. Offutt, "Input validation analysis and testing," Empir. Softw. Eng., vol.11, no.4, pp.493–522, 2006.
[10] MSDN, Design guidelines for secure web application, Available at: http://msdn.microsoft.com/library/default.asp?url=/library/enus/secmod/html/secmod77.asp, 2009.
[11] A.M. Memon, "An event-flow model of GUI-based applications for testing," Softw. Test. Verif. Rel., vol.17, no.3, pp.137–157, 2007.
[12] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A.M. Memon, "Lightweight static analysis for GUI testing," IEEE 23rd International Symposium on Software Reliability Engineering, pp.301–310, 2012.
[13] F. Belli, A. Hollmann, and N. Nissanke, "Modeling, analysis and testing of safety issues — An event-based approach and case study," 26th Int. Conf. Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol.4680, pp.276–282, Springer, 2007.
[14] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, "Volume ii: Technical concepts of component-based software engineering," Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.
[15] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre, "A contracting system for hierarchical components," Component-Based Software Engineering, Lecture Notes in Computer Science, vol.3489, pp.187–202, Springer, 2005.
[16] R. Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying behavioral compositions in object-oriented systems," ACM SIGPLAN Notices, vol.25, no.10, pp.169–180, 1990.
[17] Y. Le Traon, B. Baudry, and J.M. Jezequel, "Design by contract to improve software vigilance," IEEE Trans. Softw. Eng., vol.32, no.8, pp.571–586, 2006.
[18] B. Meyer, "Applying 'design by contract'," Computer, vol.25, no.10, pp.40–51, 1992.
[19] D. Wampler, "Contract4j for design by contract in Java: Design pattern-like protocols and aspect interfaces," AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), pp.27–30, Bonn, Germany, 2006.
[20] S. Vinoski, "REST eye for the SOA guy," IEEE Internet Comput., vol.11, no.1, pp.82–84, 2007.
[21] H.L. Truong, G.R. Gangadharan, M. Comerio, S. Dustdar, and F. De Paoli, "On analyzing and developing data contracts in cloud-based data marketplaces," 2011 IEEE Asia-Pacific Services Computing Conference (APSCC), pp.174–181, 2011
[22] J.L. Crowley, J.F. Leathrum, and K.A. Liburdy, "Issues in the full scale use of formal methods for automated testing," ACM SIGSOFT Software Engineering Notes, vol.21, no.3, pp.71–78, 1996.
[23] R. Plosch and J. Pichler, "Contracts: From analysis to C++ implementation," IEEE 30th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 30), pp.248–257, 1999.
[24] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," Softw. Test. Verif. Rel., vol.13, no.1, pp.25–53, 2003.
[25] I. Ciupa, Strategies for random contract-based testing, Ph.D., ETH Zurich (Swiss Federal Institute of Technology Zurich), 2008.
[26] R.V. Binder, Testing object-oriented systems: Models, patterns, and tools, Addison-Wesley Longman, Boston, MA, USA, 1999.
[27] W. Zheng and G. Bundell, "Test by contract for UML-based software component testing," IEEE International Symposium on Computer Science and Its Applications, pp.377–382, Washington, DC, USA, 2008.
[28] R. Heckel and M. Lohmann, "Towards contract-based testing of web services," Electronic Notes in Theoretical Computer Science, vol.116, pp.145–156, 2005.
[29] B. Beizer, Software testing techniques, 2nd ed., Van Nostrand Reinhold, New York, NY, USA, 1990.
[30] ISO 5806, Specification of single-hit decision tables, Information processing, 1984.
[31] H. Hermanns, J. Meyer-Kayser, and M. Siegle, "Multi terminal binary decision diagrams to represent and analyse continuous time

Markov chains," 3rd Int. Workshop on the Numerical Solution of Markov Chains, pp.188–207, 1999.
[32] JAPS, Japs: Java port scanner, Available at: http://antaki.ca/code/java/japs/, 28.01.2014.
[33] IANA, Port numbers, Available at: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt, 28.01.2014.
[34] A. Stevenson, Aspect-oriented smart proxies in Java RMI, M.Sc., University of Waterloo, Ontario, Canada, 2008.
[35] Y. Mokhtari, S. Abed, O.A. Mohamed, S. Tahar, and X. Song, "A new approach for the construction of multiway decision graphs," Theoretical Aspects of Computing, ICTAC 2008, Lecture Notes in Computer Science, vol.5160, pp.228–242, Springer, 2008.
[36] R.E. Bryant, "Symbolic boolean manipulation with ordered binary decision diagrams," ACM Computing Surveys, vol.24, no.3, pp.293–318, 1992.
[37] S.J. Russell and P. Norvig, Artificial intelligence: A modern approach, Prentice-Hall, Upper Saddle River, NJ, USA, 1995.
[38] T. Tuglular, "Test case generation for firewall implementation testing using software testing techniques," 1st International Conf. on Security of Inform. & Networks, pp.196–203, Trafford, Northern Cyprus, 2007.
[39] A.P. Mathur, Foundations of Software Testing, 2nd ed., Pearson, India, 2008.
[40] A. Arcuri, M.Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," IEEE Trans. Softw. Eng., vol.38, no.2, pp.258–277, 2012.

**Tugkan Tuglular** received the B.S., M.S., and Ph.D. degrees in Computer Engineering from Ege University, Turkey in 1993, 1995 and 1999, respectively. He worked as a research associate at Purdue University from 1996 to 1998. He has been with Izmir Institute of Technology since 2000.

**Arda Muftuoglu** received his M.Sc. in Computer Engineering from Izmir Institute of Technology in 2010 and B.Sc. in Computer Engineering from Yeditepe University in 2007. He is currently working as a software engineer at M.O.S.S. Computer Grafik Systeme GmbH in Munich, Germany. Prior to joining M.O.S.S., he worked as a researcher at Technische Universität Darmstadt in Darmstadt, from 2011 to 2012.

**Fevzi Belli** completed his Ph.D. in formal methods for self-correction features in sequential systems in 1978 and his "Habilitation" (German Post-Doctoral degree) in software engineering in 1986 at Berlin Technical University. In 1983, he was awarded a professorship at the University of Applied Sciences in Bremerhaven; in 1989 he moved to the University of Paderborn. Since 2014, he has been a full professor at Izmir Institute of Technology.

**Michael Linschulte** studied Business Computing Systems and received his Ph.D. degree at University of Paderborn before he joined Andagon in Cologne. His research interests include analysis and testing of web applications as well as web services by means of model-based testing. Further, he is interested in test automation, software reliability and fault tolerance.