Event-Based Input Validation Using Design-by-Contract Patterns

T. Tuglular, C. A. Muftuoglu
Department of Computer Engineering,
Izmir Institute of Technology, Turkey
{tugkantuglular/ardamuftuoglu@iyte.edu.tr}

F. Belli, M. Linschulte
Department of Computer Science, Electrical Engineering and Mathematics,
University of Paderborn, Germany
{belli/linschulte@adt.upb.de}

Abstract

This paper proposes an approach for validation of numerical inputs based on graphical user interfaces (GUI) that are modeled and specified by event sequence graphs (ESG). For considering complex structures of input data, ESGs are augmented by decision tables and patterns of design by contract (DbC). The approach is evaluated by experiments on boundary overflows, which occur when input values violate the range of specified values. Furthermore, a tool is presented that implements our approach enabling a semi-automatically detection of boundary overflow errors and suggesting correction steps based on DbC.

Keywords*: Input Validation, Event Sequence Graphs, Decision Tables, Design by Contract, Boundary Overflow, Security Testing*

1. Introduction

Input validation testing chooses test data that attempt to show the presence or absence of specific faults pertaining to input tolerance [16]. This paper focuses on numerical input validation testing of graphical user interfaces (GUI). Our approach for input validation suggests to specify user interface requirements and to convert this specification into a model from which valid and invalid test cases can be generated [3]. For specification of user-system interactions we choose an event-based formal model, where the inputs and events are merged and assigned to the vertices of an event transition diagram, called event sequence graph (ESG); arcs visualize the sequence relation of the events. An ESG is a simple albeit powerful formalism for capturing the behavior of interactive systems. However, modeling complex boundary restrictions on input data as well as dependencies among them inflates the ESG model of a system under consideration (SUC). To overcome this problem, we refine the nodes of the underlying ESG by decision tables, which visualize Boolean algebraic constraints on input data [4]. Decision table augmented ESG is supplemented with design by contract (DbC) patterns so that decision table rules for numerical input validation are refined to pre-condition rules. Based on these concepts, test data are generated. Equivalence class partitioning and boundary value approaches support the test case generation process [1,2].

This paper is an extension of our preliminary work [26], where we introduced algorithms for detection and correction of boundary overflow vulnerabilities through static analysis. The novelty of the present paper stems from following:

(i) Theoretical background is extended by incorporating ESGs.

(ii) Concept of DbC patterns have also been formalized. Especially pre-condition pattern of DbC plays an important role in refining decision tables for input validation. The formalism we introduce in Section 3.3 enables to considerably improve test case generation algorithm.

(iii) The tool we introduced in our preliminary work is improved. Our tool now adds an exception handling mechanism, which we built on DbC concept, instead of a simple *if* statement wherever necessary.

(iv) For validation of the approach we tested three open source port scanners, developed in C++, in a local

978-0-7695-3878-5/09 $26.00 © 2009 IEEE
DOI 10.1109/ISSRE.2009.20

IEEE computer society

area network (LAN). The resulting network packets are captured using a network utility.

(v) We enriched the paper by reporting on our experiences we gained by experiments in (iv).

Next section summarizes related work before Section 3 outlines the theoretical background of the approach. The core of the paper, Section 4, presents our test by DbC supplemented ESG approach. Sections 5 and 6 include technical details of the approach and case studies on different port scanners. Section 7 concludes the paper and outlines future work planned.

2. Related Work

Our approach combines input validation with static analysis for evaluating given constraints. Input validation checks the syntax and partly semantics of information provided by user via GUI [17]. Because input validation errors may lead to malfunctions of the entire system as well as to vulnerabilities for attacks [18], various specification-based and implementation-based test techniques exist to validate user interfaces [16]. Event sequence graphs [3] can be used for analysis and validation of UI requirements prior to implementation and testing of the code [19].

Static analysis techniques are used to handle buffer overflow problems, which are one of the common security issues as they may lead to vulnerabilities like system crash, corruption of data or undesirable system access. They occur when a programmer implements incorrect bound checks on buffer size or even fails to do bounds checking where data is written into a fixed length buffer [12]. By definition buffer overflow is similar to boundary overflow, which is an input error and occurs when values are entered that violate the range of values. Such entries exceed the implicitly or explicitly specified but not implemented boundary values. Therefore, research results on buffer overflows can be applied to boundary overflow problems.

According to [6], static analysis tool BOON applies integer range analysis to determine whether a C program can index an array outside its bounds. UNO, another static analysis tool accepts user-defined properties of application specific requirements to overcome specific problems [7]. In [8], taint propagation is defined as a technique which is used by static analysis tools to find software vulnerabilities caused by failed or missing input validation. In taint propagation, the tool tracks the tainted data, including also the parts of the program where the tainted data has effect on. A taint analysis is performed to find the places where data is read from an untrusted source [9], e.g., by using Patterson's value range propagation algorithm for calculating the range of possible values for each variable [10].

However, all of these techniques lack clearly arranged representations enabling a systematic evaluation. Therefore, we suggest modeling with ESG that are augmented with decision tables and DbC patterns to provide a simple, nevertheless powerful representation of contracts for checking a SUC on numerical input vulnerabilities.

There exist some approaches that adopt the DbC-idea for testing. Zheng et al. [22] introduced an UML-based software component testing technique called Test by Contract. There are also some contract-based testing techniques focused on web service testing [21]. Languages like Phyton, C++, Java are extended to comply with DbC for catching bugs [23]. In [24], the DbC concept is integrated into the programming language Python and adopted by adding mechanisms for dynamic type checking of method parameters and instance variables. Guerreiro [23] used design by contract in C++ by using and inheriting the Assertions class.

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 tool distinguishing between legal and illegal situations. These requirements are fulfilled by ESGs.

3.1. Event Sequence Graphs

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 $\langle \xi, v_0, \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for i = 0,…,k-1 and $v \neq \xi, \gamma$.

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.

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

Our approach assumes that upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the desirable final or interim situation. Due to this requirement, a complementary view is necessary to consider potential user errors in the modeling of the system. Graphically speaking, missing edges of the ESG represent undesirable user-system interactions, i.e., *faulty event pairs (FEP)*. FEPs can systematically be constructed by either (1) adding arcs in the opposite direction wherever only one-way arcs exist, or (2) adding two-way arcs between vertices wherever no arcs connect them, or finally, (3) adding self-loops to vertices wherever none exist.

Definition 3. Let $ES = \langle v_0,\ldots,v_k\rangle$ be an event sequence of length k+1 of an ESG and FEP = $\langle v_k,v_m\rangle$ a faulty event pair. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0,\ldots,v_k,v_m\rangle$. FES is *complete* (or, it is called a *faulty complete event sequence, FCES*) iff $v_0 \in \Xi$. The ES as part of a FCES is called a *starter.*

CES and FCES form test cases to our SUC. The SUC is supposed to accept test inputs described by CESs in the specified order whereas test inputs described by FCESs should result in a warning.

3.2. Decision Table Augmented ESGs

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, (e.g. see Table 2). Such refined nodes are double-circled.

Definition 4: A Decision Table $DT = \{C,A,R\}$ represents actions that depend on certain constraints where:

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

Decision tables [11] 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") [4].

Definition 5: Let R be defined as in Definition 4. Then a *rule* $R_i \in R$ is defined as $R_i = (C_{True},C_{False},A_x)$ where:

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

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ 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 $C_{True} \cup C_{False} \subset C$. We use DT to refine data input of GUI's.

3.3. DbC Patterns for Decision Tables

DbC is an object-oriented design technique that was first introduced by Meyer in 1992 [20]. DbC focuses on the extension of source code, e.g., a method, by pre-conditions, post-conditions, and invariants that can be evaluated during runtime (similar to a legal contract). Pre-conditions have to be fulfilled before a method is executed; post-conditions have to be ensured after a method is executed. Invariants are conditions that must hold anytime a method is invoked [21]. Software components are extended by those pre-conditions, post-conditions, and invariants so that the compliance with them can be verified during runtime. Although decision tables can contain a wide variety of constraints, we

classified them into three groups; namely pre-conditions, post-conditions, and invariants, by utilizing DbC patterns for automation purposes. The automation to be achieved comes in two folds: (1) test automation to detect error(s) and (2) automatic code correction to remove defects from software.

Now, for consider DbC concepts and moreover exception messages to be thrown, we refine the definition of rules in Definition 6.

Definition 6: Let C_{True} and C_{False} be defined as in definition 5 and A be defined as $A = A_{xcpt} \cup A_{ui}$ with A_{xcpt} containing exception messages and A_{ui} containing possible user interactions. Then a *rule* can be defined by

$R_i = (t, C_{True}, C_{False}, A_x)$ where

- $t \in \{t_<, t_>, t_{<>}\}$ is a time marker with
  - $t_<$ indicating pre-condition
  - $t_>$ indicating post-condition
  - $t_{<>}$ indicating invariant
- $A_x \subseteq A_{corr}$ with $A_{corr} \subseteq A_{ui} \times \{A_{xcpt} \cup \varepsilon\}$ and ε defining an empty exception

Example. For DT presented in Table 1, following sets and rules are given:

A_{ui} = {accept, abort, btn_3}

A_{xcpt} = {Exception1, Exception2, Exception 3}

R_1 = ({$t_<$}, {Condition1, Condition2}, {}, {(accept, ε), (abort, ε), (btn_3, ε)})

R_2 = ({$t_<$}, {Condition1}, {Condition2}, {(accept, Exception1), (abort, ε), (btn_3, Exception3)})

R_3 = ({$t_<$}, {Condition2}, {Condition1}, {(accept, Exception2), (abort, ε), (btn_3, ε)})

R_4 = ({$t_<$}, {}, {Condition1, Condition2}, {(accept, Exception1), (accept, Exception2), (abort, ε), (btn_3, Exception3)})

Table 1. Example of a refined DT with exceptions

$t_<$		R1	R2	R3	R4
Condition1		T	T	F	F
Condition2		T	F	T	F
accept		X			
	Exception1		X		X
	Exception2			X	X
abort		X	X	X	X
btn_3		X		X	
	Exception3		X		X

As an example, rule 3 reads as follows: If Condition2 is resolved to *true* and Condition1 is resolved to *false*, a press of accept button results in Exception1, a press of abort or btn_3 will throw no exception and therefore the input is accepted.

We use pre-conditions, post-conditions, and invariants to supplement DTs with specific classes of rules.

3.4. Test Case Generation Algorithm

Nodes of an ESG represent either events or other ESGs or decision tables. An ESG visualizes sequences of events and therefore allows detection of discrepancies in the sequential execution of user-system-interaction [4]. DTs augment the ESG given to support analysis of causal dependencies and restrictions of events. Especially, pre-condition pattern of DbC could be used to avoid vulnerabilities introduced by invalid inputs. Pre-conditions will ensure that the inputs taken from GUI are valid. For input validation, only pre-condition rules are entered into the DT. Hence, decision table augmented ESGs are reduced and simplified by considering only the pre-conditions of the GUI inputs.

Equivalence class testing partitions the input space into equivalence classes according to the input conditions. Test cases are designed by selecting at least one condition from each equivalence class. Our approach supplements this technique with boundary value analysis [1,2] which complements the equivalence partitioning by selecting test cases at the edges of a class [2]. Thus, we strengthen equivalence class testing by the cause-effect testing approach which uses decision tables to generate test cases where the input conditions represent the causes and actions represent the effects. This leads to an algorithm to generate test case values from DbC-supplemented decision tables by considering three validation types: *isolated validation*, *interdependency validation*, and *service-specific validation* [25], as depicted in Figure 1.

Isolated validation checks boundary conditions (restrictions) and interdependency validation checks relations between the variables (dependencies). Service-specific validation considers conditions related to business or service. As an example, consider port values: For isolated validation, the considered variables should be between the port ranges (0-65535). The restriction that the minimum port value should be lower than the maximum port value is associated with interdependency validation. The dynamic and/or private ports are from 49152 through 65535 [5]. No ports can be registered in the dynamic range and it is commonly used by operating system kernels. The port allocations are only valid for the duration of the session of the connection. For service-specific validation, the port values between the dynamic ranges are not valid when the session is closed, although the values are inside the port ranges and the dependency requirement

holds for the port values. Hence, input validation aspect supports all three types.

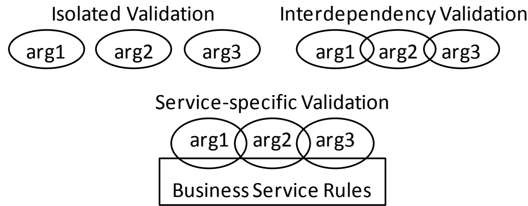

Figure 1. Input validation types [25]

Algorithm 1 shows the test case generation algorithm. For all the rules in the decision table, a specific test value for each variable is generated by regarding all its conditions and test values for all variables constitute the test case for that rule. First four columns in the decision table are for holding the type of the validation, variable1, operator and boundary, which can be a value or a variable depending on the type of the validation, respectively.

The decision table that is used to generate the test cases using the algorithm in Section 6 (Case Study) is shown in Table 2 and the test cases generated are depicted in Table 3. The algorithm creates a list for each constraint containing the conditions of the variables and generates the test values according to the following policy: First, test values are generated according to the boundary conditions for each variable in isolation. The generated values are alternating around the boundary values. Second, the relationships and dependencies for the variables are considered and the test values are altered according to the relation if needed. Finally, the test values are modified by considering the service-specific conditions.

4. Numerical Input Validation

The numerical input validation approach proposed here is composed of two phases: (1) testing the SUC with the test cases generated by DbC Supplemented ESGs and (2) detecting/correcting deficient input validation code if errors are found during the test phase.

4.1. Testing by DbC Supplemented ESGs

As a first step, the approach generates test cases by using decision table augmented-ESGs as defined in Section 3. Equivalence class testing supplemented with boundary value analysis is used to generate test cases, which are selected from the values that are at the edges of each equivalence class. Equivalence class testing is strengthened by the cause-effect testing approach which uses decision tables to design test cases.

As a second step, the SUC is tested manually in real environment by entering these values to its user interface. The faults are obtained and extracted manually to a file. Applying our proposed detection and correction method (see next section), the new corrected version of the SUC is tested in the real environment again. The faults before and after applying our method are compared. The approach is summarized in Figure 2.

4.2. Deficient Input Validation Code Detection and Correction

We propose a detection algorithm to check the error handling mechanism of the SUC related to validation of numerical inputs. The algorithm scans the source code statically, detects the points that may cause problems (possible violation of boundaries) and checks the error handling mechanism of the SUC against validation errors. The deficient parts of the error handling mechanism related to numerical input validation are identified first. Once detected, a mechanism is required to correct the deficiencies. The correction mechanism relies upon the DbC technique discussed in Section 3. Our correction algorithm provides an error handling mechanism through extension of source code by pre-condition contract methods where control for the numerical input validation vulnerability does not exist.

Algorithm 1. Test case generation algorithm

```
Input: Decision Table
Output: Test Cases
for each rule
        generate a test case by considering each variable in isolation
        modify the test case by considering the interdependencies between the variables
        modify the test case by considering the service-specific conditions
end for
```

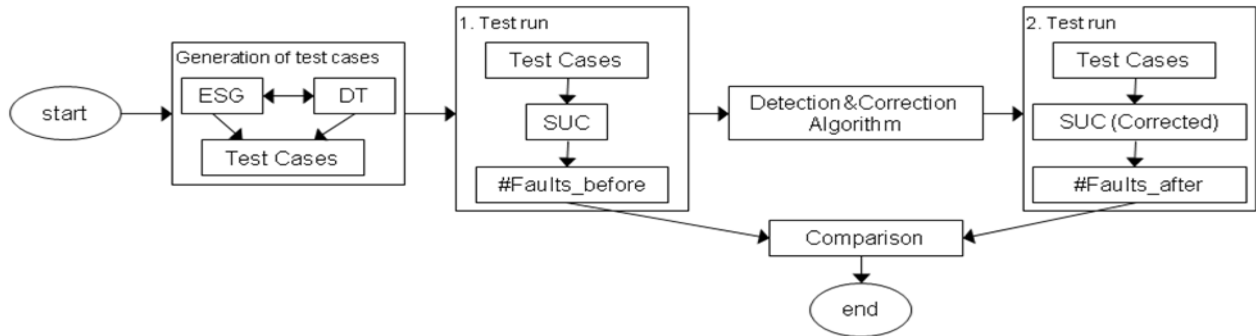

Figure 2. Summary of the approach

Deficient numerical input validation code detection algorithm consists of four steps. In step 1, pre-conditions, which are defined in the decision table that is generated by the decision table augmented ESG method, are uploaded from related directory. In step 2, the variable definitions along with their specified types are found from the source code and displayed in a table. In step 3, the variables shown in the table are matched with the uploaded pre-conditions. Matched variable's boundary condition is set to true. Step 4 traces the variables from the source code and finds the first line that the variable is used. After that, the pre-conditions are compared with the conditions of the variable, written in the source code.

After detection algorithm is completed correction algorithm, which is considered as step 5, may be executed. The correction mechanism extends the source code by inserting "Require" function (the function for implementing pre-conditions in DbC) as in [23]. As mentioned in the previous subsection, three types of pre-conditions are checked. The mechanism is applied after the trace line of the variable where the condition check for the variable does not exist.

5. Implementation and Tool support

For the implementation of our approach as introduced in Section 4, we developed a numerical input validation analysis tool in Java in Microsoft Windows environment, working on software developed in C++. As a static analysis tool, it analyzes the source code of SUC, finds the deficient parts that may cause numerical input validation vulnerabilities and extends the source code by inserting pre-condition functions to ensure that the specified conditions hold before the inputs are processed. The class "Assertions" [23] that provides the functions required for emulating pre-conditions and post-conditions is used for exception handling, where in our case, only the pre-conditions are considered. Its "Require" function is used by our tool to be inserted where the deficiency of a control mechanism exists for numerical input validation.

The numerical input validation analysis tool takes two inputs: (1) the directory of the software to be analyzed and (2) DbC supplemented DT for the GUI. Our implementation requires a manual matching of the listed variables with the pre-conditions from DT augmented ESG model of the GUI. The tool outputs the variables that have the boundary condition, displays the conditions of the variables as well as whether or not condition checks exist in the source code related to numerical input validation. The correction mechanism is applied by informing the user about the insertion of the exception handling code where the pre-condition checks do not exist. Figure 3 shows GUI of the tool that enables to input source directory of the software to be checked, shows the detection steps, suggests corrections and displays the outputs.

6. Case Study

We evaluated our approach and the tool introduced in Section 5 by means of three port scanners. A port scan function analyzes a single port or a range of ports, i.e., ports between a given minimum and maximum, to check whether they are open or not. The user interface behavior of the port scan function is modeled by using DT augmented ESG. Test cases are generated for minimum and maximum port from the decision table using test case generation algorithm presented in Section 3. Test of the port scan function is evaluated in a real network environment and faults have been recorded. As a next step, our tool analyzes the source directory to detect and correct the vulnerabilities related to boundary overflow. Finally, the faults detected before and after applying the boundary overflow detection algorithm are compared.

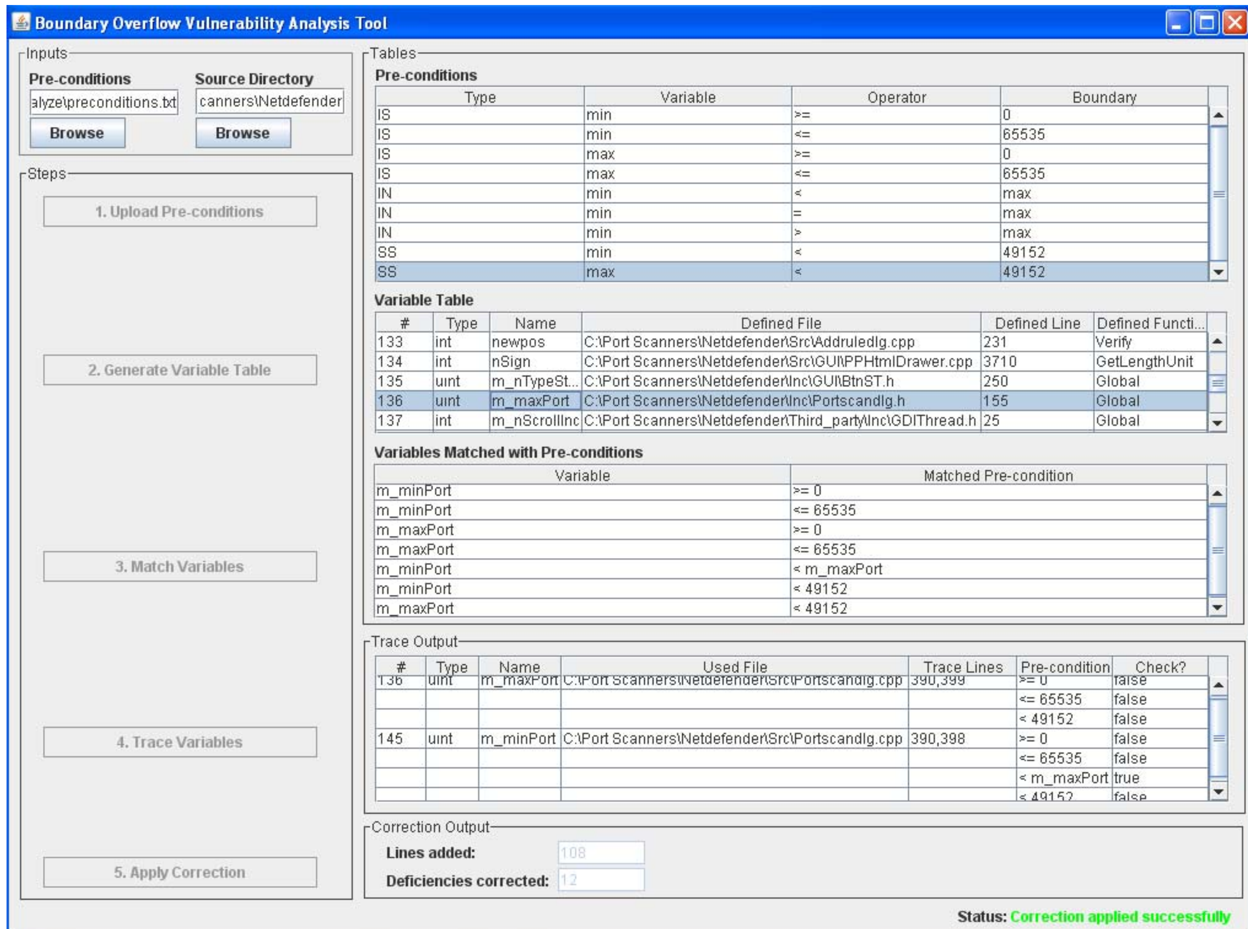

Figure 3. Numerical input validation analysis tool - graphical user interface screen

6.1. Boundary Overflow Detection

We exemplify the case study on the basis of the port scanner part of open source firewall software, i.e., Netdefender Firewall (version 1.5) [13]. Its GUI is shown in Figure 4. The ESG model of the port scanner is given in Figure 5. The decision table given by Table 2 refines the related nodes of the ESG, which are double-circled [4], e.g., the node labeled "enter min&max ports" of Figure 5 is refined by Table 2.

Table 2 structures the decision process by modeling possible actions for related conditions. The decision table is built to generate test data for the minimum and maximum port values of the port scanner according to the rules. Algorithm 1 is applied to generate test data according to the rules of the decision table. For each rule, a test pair is generated based on equivalence class testing and boundary value approach. The constraints in the first part (rows 1-4) of the decision table indicate the boundary conditions. Meanwhile, the constraints in the rows 5-7 indicate the relations of the variables with each other. Furthermore, the constraints in the rows 8-9 are included due to service-specific constraints.

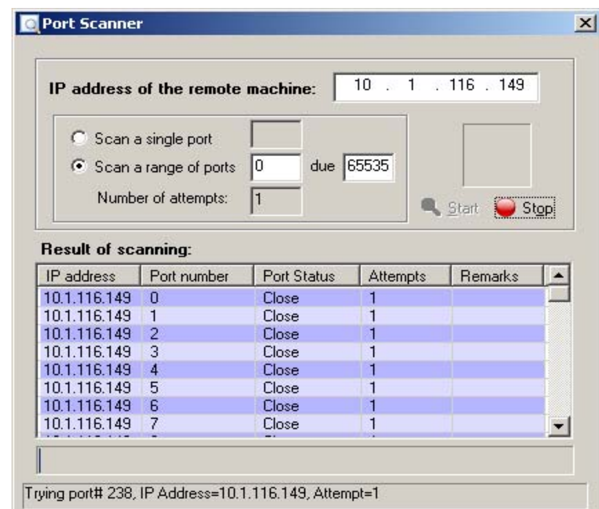

Figure 4. Netdefender Port Scanner

The algorithm creates a list for each constraint containing the conditions of the variables and generates the min and max test case pairs. Table 3 shows the generated test values as the output of the test data generation algorithm.

6.2. Testing Process in Real Environment

The port scanner is evaluated in a local area network (LAN) and the generated test values are applied as inputs to the GUI of the port scanner. The user interface outputs are obtained and the network packet outputs are captured. The outputs are extracted to a spreadsheet document. Table 4 shows a sample view of the spreadsheet document. The document displays the test values as input pair, GUI and network packet outputs, state of the case (erroneous or not), and the error message.

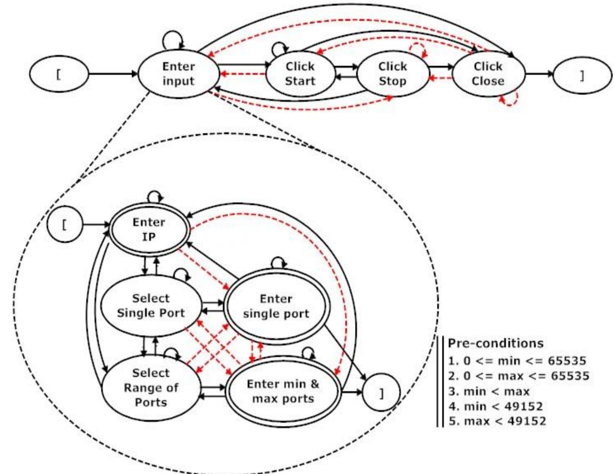

Figure 5. ESG model of the port scanner showing legal and illegal interaction pairs

Table 2. Decision Table for "Enter min&max ports" of Figure 5

	Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
1	min >= 0	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	min <= 65535	T	T	T	T	T	T	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T
3	max >= 0	F	F	F	T	T	T	F	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T
4	max <= 65535	T	T	T	F	T	T	T	F	F	F	T	T	T	T	F	F	T	T	T	T	T	T	T	T
5	min < max	F	F	T	T	T	T	T	F	F	T	F	F	F	F	T	T	F	F	F	F	F	T	T	T
6	min = max	F	T	F	F	F	F	F	F	T	F	F	F	F	F	F	F	F	F	F	T	T	F	F	F
7	min > max	T	F	F	F	F	F	F	T	F	F	T	T	T	T	F	F	T	T	T	F	F	F	F	F
8	min < 49152	T	T	T	T	T	T	F	F	F	F	F	F	F	T	F	T	F	F	T	F	T	F	T	T
9	max < 49152	T	T	T	F	F	T	T	F	F	F	F	T	T	T	F	F	F	T	T	F	T	F	F	T
	Actions																								
A1	Exception 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X								
A2	Exception 2	X	X						X	X		X	X	X	X			X	X	X	X	X			
A3	Exception 3				X	X		X	X	X	X	X	X	X		X	X	X	X		X		X	X	
A4	Accept input																								X

To sum up, the cases with out of boundary input pairs give rise to problems in the network environment. In certain cases (2, 3, 7, 9, 10, 13, 14, 15, 16), there are faulty input pairs that are out of boundary values but the program behaves as they are not faulty. This is critical, because the program does not abandon processing the related task, hence the resulting situation forces the program to work erroneously. In some cases (2, 3, 7, 13, 14), the client does not stop sending the TCP packets to the target computer, keeps on sending the packets in an infinite loop and generates a flood in LAN.

6.3. Evaluation and Lessons Learned

Sections 6.1 and 6.2 presented the test results of the first SUC. Table 5 displays the outputs after the evaluation of our tool on the port scanner of the Netdefender firewall. It can be seen that the original software does not have control mechanisms for the out of boundary input values causing boundary overflow.

Table 3. Test Cases generated from rules of the Decision Table

Rule	Test Values	Rule	Test Values
R1	(-1,-2)	**R13**	(49152,-1)
R2	(-1,-1)	**R14**	(0,-1)
R3	(-2,-1)	**R15**	(49152,65536)
R4	(-1,65536)	**R16**	(0,65536)
R5	(-1,49152)	**R17**	(49153,49152)
R6	(-1,0)	**R18**	(49152,0)
R7	(65536,-1)	**R19**	(1,0)
R8	(65537,65536)	**R20**	(49152,49152)
R9	(65536,65536)	**R21**	(0,0)
R10	(65536,65537)	**R22**	(49152,49153)
R11	(65536,49152)	**R23**	(0,49152)
R12	(65536,0)	**R24**	(0,1)

Table 4. Outputs of the test cases

#	Input Pair	GUI Output	Network Packet	Erroneous?	Error Message?
1	(-1,-2)	No output	No packet	Yes	Message 1
2	(-1,-1)	(-1,0,1,2,3,...∞)	65535,1,2,...65535…	Yes	No
3	(-2,-1)	(-2,-1,0,1,2,3,...∞)	65534,65535,1,2,...65535…	Yes	No
4	(-1,65536)	No output	No packet	Yes	Message 1
5	(-1,49152)	No output	No packet	Yes	Message 1
6	(-1,0)	No output	No packet	Yes	Message 1
7	(65536,-1)	(65536...∞)	1,2,...65535,1,2,....65535…	Yes	No
8	(65537,65536)	No output	No packet	Yes	Message 1
9	(65536,65536)	(65536)	No packet	Yes	No
10	(65536,65537)	(65536,65537)	1	Yes	No
11	(65536,49152)	No output	No packet	Yes	Message 1
12	(65536,0)	No output	No packet	Yes	Message 1
13	(49152,-1)	(49152...∞)	49152,49153,...65535,1,2,...65535…	Yes	No
14	(0,-1)	(0...∞)	1,2,...65535,1,2,...65535…	Yes	No
15	(49152,65536)	(49152...65536)	49152,49153,…65535	Yes	No
16	(0,65536)	(0...65536)	1,2,…65535	Yes	No
17	(49153,49152)	No output	No packet	Yes	Message 1
18	(49152,0)	No output	No packet	Yes	Message 1
19	(1,0)	No output	No packet	Yes	Message 1
20	(49152,49152)	(49152)	49152	No	
21	(0,0)	(0)	No packet	No	
22	(49152,49153)	(49152,49153)	49152,49153	No	
23	(0,49152)	(0,49152)	1,2,…49152	No	
24	(0,1)	(0,1)	1	No	

Message 1: "The maximum range cannot be less than the minimum one".

The second and third evaluations were performed on port scanners named Multiscan (version 0.8.5) [14] and Pscan [15]. They are open source port scanners coded in C++, which allow you to scan a range of IP addresses and ports.

As in the first evaluation, we observed that also the second and third SUC have no exception handling mechanisms. The control mechanisms against out of boundary values are deficient for the three port scanners. Hence in all of three cases, our tool inserted control statements to fulfill the deficiencies of the software. After the insertion of control statements related to boundary constraints in the port scanner of Netdefender firewall, the software is evaluated in LAN again and the generated test cases are applied as inputs to the GUI of the port scanner. The outputs considerably differ from the ones in Table 4. In erroneous cases (1-19), the software outputs the right error message and aborts sending the packets.

An overview of the three test runs comparing number of faults detected before and after the detection algorithm can be seen in Table 5. It is evident that our tool has successfully carried out detection and correction operations. Analysis of the evaluation results encourages the generalization that boundary overflow vulnerabilities are not considered and thus countermeasure actions are neglected during software development. Therefore, tools as we introduced in this paper might be useful to prevent likely failures or undesirable situations that may occur as a consequence of deficiency control mechanism in the software.

Table 5. Comparison of the three test runs

		# of faults detected		
Software	**# of test cases**	**Before**	**After**	**Benefit of the approach (% of faults corrected)**
Netdefender	24	19	0	100%
Multiscan	24	10	0	100%
Pscan	24	4	0	100%

7. Conclusion

In this paper, we have proposed a solution for the numerical input validation problem and reported our experience gained through experiments as described in case study. DT augmented ESGs supplemented with

DbC patterns are used for modeling GUI of SUC and generating test cases for input validation. An algorithm is introduced to validate the exception handling mechanism of SUC related to invalid numerical inputs and provide the necessary exception handling mechanism where none exists. A tool we developed supports the deployment of the algorithm introduced. Three port scanners have been tested for evaluation of this tool. Results of those tests show that the approach is very effective for finding deficiencies in the exception handling mechanism of SUC concerning boundary overflow problems. Moreover, our approach includes appropriate checks to compensate those deficiencies of SUC.

References

[1]. Tuglular, T. 2007. Test Case Generation for Firewall Implementation Testing using Software Testing Techniques. In Int. Conf. on Sec. of Inf. and Networks, N. Cyprus.

[2]. Liu, H., Kuan Tan, H. B. 2009. Covering code behavior on input validation in functional testing. Information and Software Technology, 51, 2, pp.546-553.

[3]. Belli, F. 2001. Finite state testing and analysis of graphical user interfaces. In Proc. of the 12th Int. Sym. on Software Reliability Engineering, pp. 34-43, IEEE, Washington, DC.

[4]. Belli, F., Linschulte, M. 2007. On Negative Tests of Web Applications, Annals of Mathematics, Computing & Teleinformatics, Volume 1, No. 5, pp. 44-56.

[5]. IANA. 2009. Port Numbers. Retrieved February 16, 2009, from http://www.iana.org/assignments/port-numbers.

[6]. Chess, B., McGraw, G. 2004. Static Analysis for Security. IEEE Security and Privacy 2, 6, pp. 76-79.

[7]. Holzmann, G. 2002. UNO: Static source code checking for user-defined properties, Bell Labs Technical Report, Bell Laboratories, Murray Hill, NJ, 27 pages.

[8]. Kolmonen, L. 2007. Securing Network Software using Static Analysis. Seminar on Network Security, Helsinki University of Technology.

[9]. Sotirov, A. I. 2005. Automatic Vulnerability Detection using Static Source Code Analysis, MS Thesis, University of Alabama.

[10]. Patterson, Jason R. C. 1995. Accurate static branch prediction by value range propagation. In Proc. of SIGPLAN Conf. on prog. language design and implementation, 67–78.

[11]. Information processing 1984. Specification of single-hit decision tables, ISO 5806.

[12]. Mell, P., Tracy, M. C. 2002. Procedures for Handling Security Patches, NIST Special Publication 800-40.

[13]. Netdefender. 2009. Netdefender firewall version 1.5. Retrieved March 3, 2009, from http://www.codeplex.com/netdefender

[14]. Multiscan. 2009. Multiscan port scanner version 0.8.5. Retrieved March 3, 2009, from http://www.sourceforge.net/projects/multiscan

[15]. Pscan. 2003. Pscan port scanner. Retrieved March 3, 2009, from http://www.codeproject.com/KB/cpp/pscan.aspx

[16]. Hayes, J. H., Offutt, A. J. 1999. Increased Software Reliability Through Input Validation Analysis and Testing. In Proceedings of the 10th Int. Sym. on Software Reliability Engineering. IEEE, Washington, DC, pp. 199–209.

[17]. Hayes, J. H., Offutt, J. 2006. Input validation analysis and testing. Empirical Software Engineering, 11, 4, pp.493-522.

[18]. MSDN. 2008. Design Guidelines for Secure Web Application. Retrieved March 3, 2009, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod77.asp.

[19]. Belli, F., Hollmann, A., Nissanke, N. 2007. Modeling, Analysis and Testing of Safety Issues - An Event-Based Approach and Case Study, 26th Int. Conf. Computer Safety, Reliability, and Security, LNCS 4680, pp.276-282, Springer.

[20]. Meyer, B. 1992. Applying "Design by Contract". In Computer. Vol. 25, No. 10, Pages 40-51.

[21]. Heckel, R., Lohmann, M. 2005. Towards contract-based testing of web services. Electr. Notes Theor. Comput. Sci., pp. 145-156.

[22]. Zheng W, Bundell G. 2008. Test by Contract for UML-Based Software Component Testing. Proc. of the Int. Sym. on Comp. Sci. and its Appl., IEEE, pp. 377-382.

[23]. Guerreiro, P. 2001. Simple Support for Design by Contract in C++. In Proc. of the 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems, July 29 - August 03, IEEE, Washington, DC.

[24]. Plösch, R. 1997. Design by Contract for Python, IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference (APSEC97/ICSC97), HongKong, December 2-5.

[25]. Stevenson, A. 2008. Aspect-Oriented Smart Proxies in Java RMI, Thesis for Master of Mathematics in Computer Science, University of Waterloo, Ontario, Canada.

[26]. Tuglular, T., Muftuoglu, C.A., Kaya, O., Belli, F. Linschulte, M. 2009. GUI Based Testing of Boundary Overflow Vulnerability, STA Workshop of 33rd COMPSAC, Seattle, USA.