MUTATION ANALYSIS OF SPECIFICATION-BASED CONTRACTS IN SOFTWARE TESTING

A Thesis Submitted to
the Graduate School of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Abbas KHALILOV

July 2021
İZMİR

ACKNOWLEDGEMENTS

First, I am very grateful to my teacher and main-advisor Assoc. Prof. Dr. Tugkan Tuglular for his support, advice, and dedicated involvement he put in this work. Without his assistance in each step of thesis, this work would have never been accomplished. I also want to sincerely thank my co-advisor Prof. Dr. Fevzi Belli for his advice and valuable comments on my work. Working on thesis with my advisors established my view on Academic world and the way people of science think and work. Therefore, I am grateful to them for broadening my horizons.

Finally, I am grateful to my parents, and grandmother for their moral support. I progressed on my master studies and academic research due to their “do not stop, keep going”, whenever I had “there is no way” in my mind.

ABSTRACT

MUTATION ANALYSIS OF SPECIFICATION-BASED CONTRACTS IN SOFTWARE TESTING

Software used in fields such as medicine, finance, aviation and aerospace, nuclear power etc. is required to be reliable. Any software failures in these fields may have catastrophic consequences such as human and financial losses, which may cause a great damage to the economy and to social well-being. Hence, before launching, software should be rigorously tested. Testing can uncover the conditions, which software cannot handle. Those conditions might be overlooked during development. So, software testing points to the faults in the software under development to be patched. The important element of software testing is the use of the adequate test cases. If the outcome of the test case is positive, that means testing did not reveal any fault, then this test case might be considered as inefficient and useless for the tested version of software. Therefore, it is important to check test cases on adequacy, which can be achieved by mutation analysis. This thesis focuses on checking the adequacy of the test cases for Decision-Table-augmented Event Sequence Graphs (ESG-DTs) representation of a system under test by using mutation analysis. Test cases are represented in the Complete Event Sequence (CES) and Faulty CES (FCES) forms. This thesis presents a new set of mutation operators for mutation of contracts represented in Multi-Terminal Binary Decision Diagram (MTBDD). This thesis introduces a new approach for mutation of the ESG-DT model by using the proposed MTBDD mutation operators. The proposed approach is evaluated on three cases. The results for all cases show the drawback of specific FCES test sequences and the relationship between the mutant detection by CES/FCES sequences and proposed mutation operators.

ÖZET

YAZILIM TESTİNDE SPESİFİKASYON TEMELLİ SÖZLEŞMELERİN MUTASYON ANALİZİ

Tıp, finans, havacılık ve uzay, nükleer enerji vb. alanlarda kullanılan yazılımlar güvenilir olması için gereklidir. Bu alanlardaki herhangi bir yazılım arızası, ekonomiye ve sosyal refahına büyük zarar verebilecek insan ve finansal kayıplar gibi feci sonuçlar doğurabilir. Bu nedenle, başlatmadan önce, yazılım titizlikle test edilmelidir. Test, yazılımın işleyemediği koşulları ortaya çıkarabilir. Bu koşullar gelişim sırasında göz ardı edilebilir. Bu nedenle, yazılım testi, geliştirilmekte olan yazılımdaki hataları düzeltmeye işaret eder. Yazılım testinin önemli bir unsuru, yeterli test durumlarının kullanılmasıdır. Test durumunun sonucu pozitifse, bu, testin herhangi bir arıza göstermediği anlamına gelir, daha sonra bu test durumu, test edilen yazılım sürümü için verimsiz ve işe yaramaz olarak kabul edilebilir. Bu nedenle, mutasyon analizi ile elde edilebilecek yeterlilik test durumlarını kontrol etmek önemlidir. Tez, mutasyon analizi kullanılarak test edilen bir sistemin Karar Tablosu ile artırılmış Olay Sıra Çizgeleri (OSÇ-KT'lar) gösterimi için test senaryolarının yeterliliğini kontrol etmeye odaklanır. Test durumları Tam Olay Sırası (TAS) ve Hatalı TAS (HTAS) formlarında temsil edilir. Bu tez, Sözleşme-Çok Uçlu İkili Karar Diyagramında (ÇTUIKD) temsil edilen sözleşmelerin mutasyonu için yeni bir mutasyon operatörleri seti sunar. Tez, önerilen ÇTUIKD mutasyon operatörlerini kullanarak OSÇ-KT modelinin mutasyon için yeni bir yaklaşım sunar. Değerlendirme bölümünde, sunulan mutasyon analizi algoritmasının uygulandığı üç durum sunulmaktadır. Tüm mutantlar için sonuçlar, spesifik HTAS test sıralarının dezavantajını ve TAS/HTAS sıraları tarafından mutant tespiti ile önerilen mutasyon operatörleri arasındaki ilişkiyi göstermektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

Figure **Page**

Figure **Page**

LIST OF TABLES

Table **Page**

Table **Page**

CHAPTER 1

INTRODUCTION

The software development process includes not only the code writing, but also its testing. Testing is not only writing proper tests, but also preparing adequate test set to make sure that the software is ready for the next stage, or deployment. Even a well-tested software does not guarantee that it will not crash in an unexpected situation. Therefore, test cases should not only make sure that the program is ready to fulfill its expectations, but also handle unexpected situations, e.g., handle incorrect input. The effectiveness of a test set can be checked by the technique called *mutation analysis* [1]. The steps involved in mutation analysis are: 1) insertion of the different kinds of faults in the original program by means of *mutation operators*; 2) generating mutant programs; 3) finding distinguished mutants against the provided test set; 4) assessing the adequacy of the provided test set by dividing the number of the distinguished mutants to the total number of mutants. In its origin the mutation analysis is intended to be a code-based technique [1]. Later, mutation analysis was adopted for specification-based testing [2]. As the specification of the system under test (SUT) can be provided in various forms, the mutated specifications permit to test different aspects/properties of the SUT.

Mutating different specification representations demands the construction of corresponding mutation operators. As stated in the last step of mutation analysis, the adequacy is measured by the number of distinguished mutants. High adequacy mark describes the given test set as highly efficient one.

This thesis is the continuation of the [3] and extends the ideas introduced in [4]. The thesis investigates the adequacy of the test set generated from the original specification model by applying it on the mutants generated from the original specification model. The specification model is represented as Decision-Table-Augmented Event Sequence Graph (ESG-DT) [3]. To perform mutation, the contract given as a DT is transformed to a multi-terminal binary decision diagram (MTBDD), then the mutation operators defined

on MTBDDs are applied to the MTBDD and finally the mutated MTBDD is transformed back to a DT, which becomes a mutated DT. This mutant model is tested by test set. The test set is represented as test sequences in two forms: CES and FCES. The CESs and FCESs are generated from the original model. The CES represents the expected behavior, which the specification should correspond, whereas FCES represents the faulty behavior which the specification should not correspond. The thesis proposes a new set of mutation operators for MTBDD mutation. The evaluation is performed on the ESG graph without the contract involvement.

In this study, the quality of the CES and FCES test suites for ESG-DT model representation is assessed. Proposed mutation analysis approach is applied on the three cases. Considering the relation between the mutant detection properties of CES and FCES described in the discussion, one can say that the mutant will be detected if and only if (iff) there is a difference in the model behavior. According to the results, the impact of the mutation operators dealing directly with the terminal nodes of MTBDD is always noticeable by test sequences, because the mutated ESG-DT model will lose an edge and/or acquire a new one, i.e., difference in the model behavior. In case of MTBDD edge mutation, the impact is sometimes noticeable or non-noticeable at all. Another observation obtained from the results is the insensitivity of the certain FCESs to the mutants, of which reason is discussed in the Evaluation chapter.

The thesis is constructed in the following way. The study starts with the literature review in CHAPTER 2. The following CHAPTER 3 describes the fundamental theory about ESG, ESG-DT, MTBDD, mutation analysis and mutation operators for ESGs and DTs. After the review of the thesis foundation, CHAPTER 4 introduces new mutation operators for DT-augmented ESG mutation, and the algorithm used for implementation of mutation analysis. CHAPTER 5 evaluates the application of the proposed algorithm and outcome of the proposed operators on three cases, namely CD player, Cruise Control, Simple Automated Teller Machine. The instruments used for the generation of mutants and test generation are described on CHAPTER 6. The last chapter concludes the thesis and provides further ways of improving and extending the work in this thesis.

CHAPTER 2

RELATED WORK

Testing software is an integral part of the software development cycle. Testing is used for strengthening software, by considering different faults occurring during software exploitation. Therefore, testing is a resource and time-consuming procedure. For this case, mutation testing technique simplifies testing procedure. Mutation testing is originated as white-box/code-based testing technique. Mutation testing embraces different fault domains by inserting faults in the software. Fault insertion is done by mutation operators, which put changes in the source code of the software. After all mutants are tested with the prepared test set. Here, the aim of the test cases in a set is to fail the mutant, in other words killing or distinguishing mutants. The downside of mutation testing is the generation of the equivalent mutants, which any test set is incapable to distinguish from the original version of software.

Unlike in code-based testing, the advantage of mutation analysis in black-box testing is that this process is fully automated and allows to reduce testing domain of system [2] so, that it allows to avoid equivalent mutant problem. The benefit of specification-based testing is that the system specifications are constructed in different representation forms.

An integral part of mutation testing is the set of established mutation operators. Operators in mutation testing inject faults into the testing system. The injected faults represent the specific fault domain, which the corresponding mutation operator represents. As mutation analysis's origin is a code-based testing technique the Yu-Seung Ma et al. in [5] proposed the comprehensive set of mutation operators for class and inter-class mutations in Java language. The authors modified and extended already existed family of mutation operators by considering the faults, which may occur because of object-oriented principles, such as inheritance, polymorphism, and overloading.

Offutt et al. in [6] shows two empirical comparisons between data flow and mutation testing. The aim is to find the powerful sides of both testing techniques or

derivation of a more efficient technique that offers the power of both mutation and data flow testing. Obtained results indicate that, despite of effectiveness of both techniques, mutation provides more stringent testing than data flow does, like mutation-adequate test sets detect more faults [6].

Andrews et al. in [7] use mutation analysis to compare and estimate test suits and criteria in terms of their cost effectiveness. Research involved four common test coverage criteria as Block, Decision, C-Use and P-Use. Where the Block coverage tests the whole module from code by ensuring that each branch is executed at least once, Decision coverage is the validation of every of accessible source code by ensuring that each branch of all possible execution decision is triggered at least once. The P-Use and C-Use are categories of Usage criteria which examines all usages of variable, where c in C-Use stands for computational, inspects all usages of a variable as part of statement, as a function parameter and as output statement and p in P-Use is stands for predicate, inspects all usages of variable in decision making statements. Across the explored criteria the consistent results show that the reliability of mutation operators' usage: generated mutants can be used to predict the detection effectiveness of real faults. The probability of detection faults given a test pool strongly affects the shape of relationship between fault detection, coverage levels, and test set sizes.

Software testing is called positive when the software is tested on proper fulfillment of required tasks. The opposite of positive testing is the negative testing. The importance of negative testing helps to prevent failures by simply handling the erroneous states. Therefore, the demand in test cases supporting negative testing increases. Strug et al. targets the problem by providing a mutation testing-based method for generating negative test cases that can support an assessment of a system ability to handle a wide range of unexpected situations [8]. The method is procedural, systematic, and human-unbiased way of defining the negative test cases without the necessity of any formal or informal description of unexpected situation.

Meyer introduced Design by Contract (DbC) approach in [9], where the author introduces also the contract notion in software development. The contract represents a mutual responsibility between caller and called units, where both promise to fulfill their requirements. One of the main benefits of DbC approach is to help in detecting and locating faults [10]. Traon et al. explores the efficiency of contract by adapting the mutation analysis [11]. Mutation analysis is used as a systematic process for fault injection and the estimation of actual values for the isolated weakness of a component and error

detection probability [11]. The mutations are done simple by injecting errors in the system. Afterwards, if contract is violated during the execution of a faulty system, this implies that the contract has detected an error [11]. Efficiency is measured by checking the contracts' efficiency on the mutants that are distinguished at least by one test case.

Compared to traditional mutation operators defined by Aichernig in [12] and Jiang et al. in [13], operators proposed in [14] are high level contract mutation operators for testing components. Authors also propose a contract-based mutation, which should serve as a test adequacy criterion for component. The reason of creating high level operators is reducing the number of mutants. Indeed, the results given by proposed operators greatly reduce the mutant number in contrast to the traditional operators. Also, application of contract mutation operators in contract-based mutation provides the same ability as that of using traditional mutation operators. Another important discovery is that the test set selected by a contract based mutation can be reused during component regression testing [14].

Another research of contract mutation presented in [15]. Contracts are declared by Spec#, which is an extension for C# programming language. Unlike conventional mutation testing which mutates source code, the proposed [15] approach mutates program contracts and generate test-input data which distinguish the mutated contract from the original one. The test case generation is focused on reported counterexamples. The idea of approach is obtaining test cases which in turn prevent implementation of the "incorrect" contract. An inherent property of the approach is that equivalent mutants are ignored, meaning that no test cases are generated for them [15].

The goals of mutation analysis as a black-box testing technique presented in [2] are determining the ability of fault detection in programs and describing a class specifications, which would benefit from this kind of testing, checking on uniqueness of generated test cases and investigation of generated test sets sizes. Mutation is performed by substitution of language elements of semi-formal specifications for every other element.

Fabri et al. perform the evaluation of mutation analysis criterion on Petri Nets-based specification [16]. As mutation analysis requires changes in original model, operators for Petri Nets mutations are presented. Mutation analysis criterion was performed manually on a test case. The mutant is considered as dead, if the mutant's vector, which is the number of tokens in each place, was different than original model's vector. To reduce testing expenses, authors examined the ideas of alternate mutation

criterions. In constrained mutation criterion a few types of mutants were examined. Randomly selected mutation required only 10% of mutants of each type were investigated. As a result, alternate mutation criteria provide great cost reduction in terms of test sequences and the mutant numbers.

Ammann et al. uses mutation analysis in combination with model checker and test generation [17]. Test cases are defined as a set of inputs and expected results, and this is emphasized as complete test case. Authors declared two classes of mutation operators: those that produce test cases from which a correct implementation must differ and those that produce test cases with which it must agree. By making syntactic errors at the level of the model checker specification, mutation operators define a form of mutation analysis. As a result, the advantages of matching model checker with mutation analysis were automatic test case generation and as opposed to code-based mutation analysis, equivalent mutant identification became also automatic.

Do Rocio Senger et al. provides [18] the mutation testing receipt for the validation of Estelle specifications. A mutation operator set for Estelle are based on the ideas of Interface Mutation. Hence, mutation testing investigates not only validation of behavioral aspect of specification, but also the intermodular communication and specification structure of Estelle. A key point for successful mutation testing of Estelle specification was also the illustration of mutation strategy, validate activity by giving priority to specific types of errors. In Estelle domain, the establishment of an incremental testing strategy becomes feasible.

Fabbri et al. proposes [19] the fundamental mechanism for validation of Statechart-based specifications by mutation testing. Considering specific features of Statechart-based specifications, the corresponding mutation operators set for statechart mutation is proposed. In that scope, mutation operators are considered as a fault model. Strategies based on mutation, incremental and hierarchical testing strategies are provided to explore statechart components separately from different Statechart features, which can cause inaccuracy in validation and testing stages.

Black et al. provides [20] a theoretical and empirical comparison of the effectiveness of mutation operators and the number of mutations the produce by following the combination of mutation analysis with [17]. In proposed method the specification-based coverage metric is used. According to the methodology, all but one copy of inconsistent mutants, which are semantic duplicates of other mutants are excluded. Mutants are distinguished by SMV model checker. The coverage is calculated

by k/N (the number of distinguished and inconsistent mutants, respectively). Among the extensive set of mutation operators only three of them demonstrated high coverage at the smaller number of generated mutants.

With the goal to reduce the number of faults in the actual programs constructed from formal specifications, Wong investigates analytically the relationships between detection conditions for several fault classes and the compares empirically the effectiveness of the mutation operators [21].

Sugeta et al. provides mutation testing mechanism for Specification and Description language (SDL) specifications [22]. Based on the behavioral features of processes, inter-process commutation and specification structure a mutant operator set is defined, which model corresponding errors. For a proper usage of mutant operators on SDL specifications a testing strategy is proposed. The advantage of the proposed strategy is that, if during the application of strategy, the error is found, the specification should be corrected, and the strategy applied again.

Liu et al. presents another specification fault investigation for Object-Z specifications [23]. Authors present five classes of mutation operators, which provide not only an assessing of specification-based test cases, but also introduce an approach of validating the correctness of specifications.

Belli et al. in [24] introduces Decision-Table-augmented Event Sequence Graphs (ESG-DT). Although the work does not contain a mutation testing notion, it introduces first simple insertion and omission mutation operators for mutating ESGs and DTs. Operators generate simple mutants which represent simple faults. Hence, for complex mutants it is enough to use a combination of them. Authors introduce a test set generation algorithm from obtained mutants.

Belli et al. first presents in [25] multiple simple mutation operators for mutation of model-based specifications. Models are represented as Directed Graphs (DG), ESG, Finite-State Machines (FSM), Statecharts (SC). All mutation operators are divided in insertion and omission categories for above listed graph-based models. The advantage these operators bring is in generation of first-order mutants which simulate simple faults. Another advantage of these operators is that they can be combined for simulation of complex faults. The main objective is to assess the fault detection ability of test cases generated from models mutated from proposed operators. Based on empirically obtained results, test sets generated by insertion operators are more effective in revealing faults than those generated by omission operators.

Khalilov et al. extends [4] a mutation operator set for specification-based contracts. Apart from existing DT mutation operators [24], authors introduce a brand new simple mutation operators for Ordered Binary Decision Diagram (OBDD). As OBDDs are limited in the number of terminal nodes this thesis extends OBDD by using Multi-Terminal Binary Decision Diagram (MTBDD).

In this thesis, a set of mutation operators for the contract mutation is proposed. The mutation operators are divided into two categories: insertion and omission. By using these operators, mutated models are generated by mutating contracts in the original model. Test cases are generated from the original model are of two types. Test cases of the first type called CES should not agree with the generated faulty model to detect it. Second type, called FCES should agree with the faulty model, in order to detect it, since both FCES and mutant are the faulty models. The mutants are tested at the level of ESG of the ESG-DT model. Therefore, considering the level of the model being tested and detection properties of the test cases, we can predict which mutants are detectable and which are living and equivalent ones.

CHAPTER 3

FUNDAMENTALS

3.1. Event Sequence Graph (ESG)

Event sequence graph (ESG) is a representation of a model of the system which is used in modeling of system behavior [26]. Modeling is performed simply by retrieving all possible legal and illegal actions, occurring during execution, of the system under consideration from its specifications and establishing all possible sequences between actions. Actions in ESGs are represented by events which occur in system and connections between events are called sequences. An event in ESG is considered as input or stimulus which's execution causes firing of another event. This phenomena helps to predict the next event and control the flow of model execution [26].

Definition 3.1: An event sequence graph ESG = (V, E, Ξ, Γ) 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 $v \neq \xi, \gamma$ [3].

Ξ (ESG), Γ (ESG) represent the entry nodes and exit nodes of a given ESG, respectively [3]. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex "[" $\in V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex "]" $\notin V$ [3]. The semantics of an ESG is as follows [3]. 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 $(v, v') \in E$ [3].

Example 3.1: The ESG depicted in Fig. 3.1, **V** = *{stop, play, pause, load, off}*, **Ξ** = *{stop, play, load}*, **Γ** = *{off}* and **E** = *{*(*stop, stop*), (*stop, play*), (stop, load), (*play, play*), (*play, stop*), (*play, load*), (*play, pause*), (*pause, play*), (*stop, off*), (*play, off*), (*load,*

off), (*load, stop*), (*load, play*), (*load, load*)*}*. As it can be seen none of sets listed above, contain pseudo vertices "[" and "]".

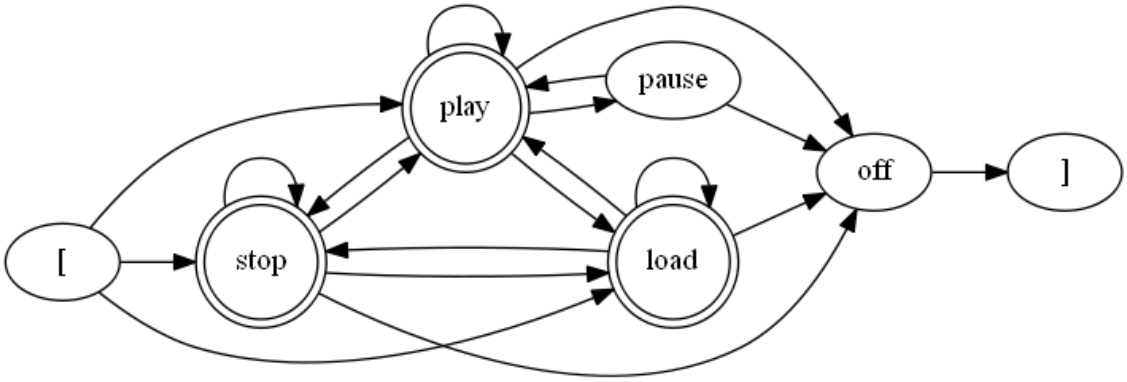

Figure 3.1. ESG of CD player.

Definition 3.2: Let V, E be defined as in Definition 3.1. 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, . . ., k-1 [3].

Definition 3.3: In order to detect entry event and exit event of an ES α (initial) and ω (end) are used, i.e., $\alpha(ES) = v_0$, $\omega(ES) = v_k$. The *successors* set of $\forall v \in V$ is denoted by $N^+(v)$ and the *predecessor* set of $\forall v \in V$ is denoted by $N^-(v)$. The number of vertices of an ES is determined by the function *l(length)*. If $l(ES) = 1$ then $ES = < v_i >$ is an ES of length (1). Each edge of ESG or an $ES = <v_i, v_k>$ of length two (2) represent an ***event pair (EP)***.

Example 3.2: The *length* of *stop – play – pause* ES shown in Fig. 3.1. is 3.

Definition 3.4: An ES is called a complete ES (CES), if $\alpha(ES) = \xi \in \Xi$ is the *entry* and $\omega(ES) = \gamma \in \Gamma$ is the *exit* [3].

Example 3.3: The CES of ESG shown in Fig. 3.1. is *stop – load – off*. This is one of the ways of walking from the start of ESG to its finish.

ESG test cases are represented as CES, the latter is presented in the following form: "(initial) user input(s) → (interim) system responses → … → (final) system response" [3].

3.2. Decision-Table-Augmented Event Sequence Graph (ESG-DT)

3.2.1. Decision Table (DT)

Decision Table (DT) is a popular tool in information processing and widely used in software testing. DT is a combination of possible inputs and corresponding system responses. DT logically connects conditions ("if") with actions ("then"). In scope of this thesis, we consider DT simple, i.e., conditions can accept only T (true) and F (false).

Definition 3.4: DT is a tabular representation of DT = (C, A, R) triple [3]. Where $C \neq \emptyset$ and $C = \{c_1, \ldots, c_n\}$ is a finite set of conditions, $A \neq \emptyset$ and $A = \{a_1, \ldots, a_m\}$ is a finite set of actions and $R \neq \emptyset$ and $R = \{r_1, \ldots, r_k\}$ is a finite set of rules, each of which invoke certain actions depending on a predefined combination of conditions [3].

Definition 3.5: Let R be as declared in Definition 3.4. Then, based on the number of conditions defined for current DT, the maximum number of rules in DT will be $2^{|C|} = 2^n$ [3]. DT with $|R| = 2^n$ are called *complete* DT.

If $|R| > 2^n$, then the DT is inconsistent and should be reconstructed [3].

Definition 3.6: Let R be defined as in Definition 3.4. Then, $\forall r \in R$ can be defined as $r = (C_{true}, C_{false}, A_m)$, where $C_{true} \subseteq C$ is the set of conditions that should be true. $C_{false} \subseteq C$ is the set of conditions that should be false [3]. $A_m \subseteq A$ is the set of actions that should be performed if all $t \in C_{true}$ are resolved to true and all $f \in C_{false}$ are resolved to false [3]. Under regular circumstances: $C_{true} \cup C_{false} = C$ and $C_{true} \cap C_{false} = \emptyset$ [3]. In case if condition is not considered in certain situations it simply denoted as '-' (don't care) in rule [3]. Based on the number of '-' in rule it can simply be calculated the real number of rules of DT by the following way: Let $u < |C|$ be the number of '-' in $r \in R$, then the number of rules substituted by '-' is 2^u [3].

Example 3.4: DT depicted on Table 3.1 is the simple complete DT. C = {*offButtonPressed, isClosed, CDpresent, lastTrackPlayed*} is condition set, A = {*play, stop, load, off*} is action set and R = {R_1, R_2, R_3, R_4, R_5} is a rule set.

For R_5 = ({*offButtonPressed*|*F*}, {*isClosed*|*T, CDPresent*|*T, lastTrackPlayed*|*T*}, {*stop*}) according to R = (C_{false}, C_{true}, A_x) and r ∈ R.

For $C_{lastTrackPlayed}$ = '-' in R3 the real number of rules is $2^1 = 2$. Therefore, R_3 can be substituted DT = (C, A, (R\$R_3$) ∪ {$R_{3.1}$, $R_{3.2}$}), where $R_{3.1}$ = ({*offButtonPressed*|*F, CDpresent*|*F*}, {*isClosed*|*T, lastTrackPlayed*|*T*}, {*stop*}) and $R_{3.2}$ = ({*offButtonPressed*|*F, CDpresent*|*F*}, {*isClosed*|*T, lastTrackPlayed*|*T*}, {*stop*}). Now, the real number of rules in DT "stop" will be 2^3 for R_1, 2^2 for R_2, 2^1 for R_3 => $2^3 + 2^2 + 2^1 + 1 + 1 = 16$. And the maximum possible number of rules according to the $|R| = 2^{|C|} = 2^4 = 16$. Hence, DT "stop" is a complete DT.

Table 3.1. "stop" DT.

stop		Rules				
		R_1	R_2	R_3	R_4	R_5
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	F	T	T	T
	CDpresent	-	-	F	T	T
	lastTrackPlayed	-	-	-	F	T
Actions	play				X	
	stop			X		X
	load		X			
	off	X				

3.2.2. DT as ESG Extension

According to Definition 3.5 the combination of conditions results in $2^{|C|}$, where |C| represents the number of conditions. Each combination of conditions would have to be modeled as vertex and is to be connected with appropriate successor [3]. Thus a DT with n binary conditions subsumes 2^n nodes to realize a thorough evaluation considering all combinations [3]. To avoid this inflation, DT are introduced to refine a node of the ESG, whereas the successors of refined node represent the actions of the DT and vice versa [3].

Definition 3.7: An event $v \in V$ of an ESG is called a *data event* (*DE*) if v is represented by a DT. A *DE* is represented as a DT, which is a contract. In turn, contracts are combined with events of ESG.

Example 3.5: Fig. 3.2 clearly demonstrates how the "stop" DT (Table 3.1) is represented by double circling the event "stop" of ESG depicted in Fig. 3.1. Actions "play", "stop", "load" and "off" indicate the corresponding *play*, *stop*, *load* and *off* events. For instance, rule R_4 says that if both *offButtonPressed* and *lastTrackPlayed* are resolved to false and both *isClosed* and *CDpresent* are resolved to true, then "play" action will be triggered and apparently the *play* will be executed, because it is one of the successors of the current event *stop*.

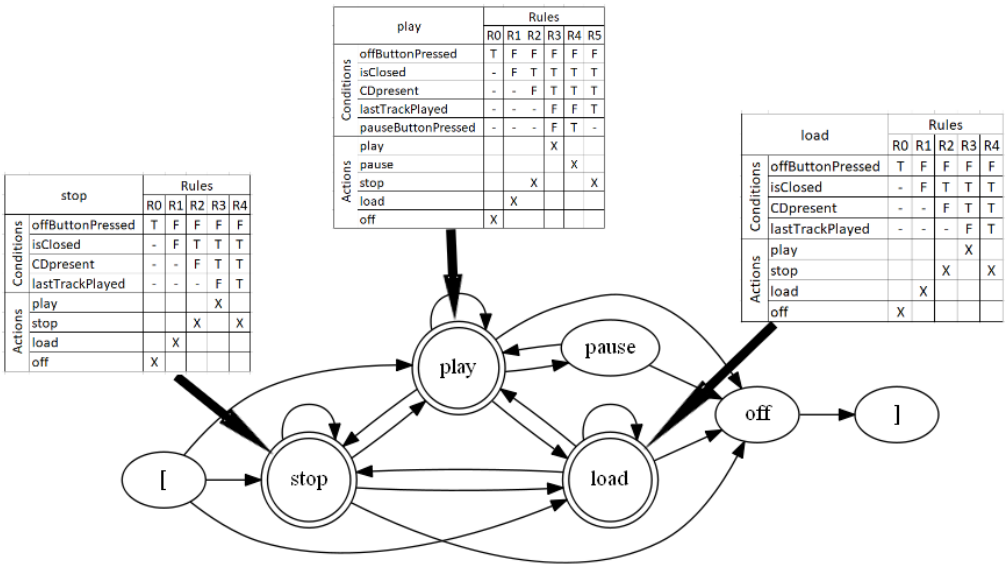

Figure 3.2. DT augmented ESG.

3.3. Multi-Terminal Binary Decision Diagram (MTBDD)

DT represent a set of Conditions which take unique combination of boolean values True and False. The unique condition combination triggers a set of certain Actions. Therefore, DT has a Rule set, where each rule represents an execution of the certain action set called under satisfaction of the given condition set.

Table 3.2. An Example Decision Table.

Example DT		Rules			
		R1	R2	R3	R4
Conditions	C1	F	F	T	T
	C2	F	T	F	T
Actions	A1	F	T	T	F
	A2	T	F	T	T

The R_1 in DT on Table 3.2, in propositional logic is represented in the following form:

$$R_1 := (\neg C_1 \wedge \neg C_2) \wedge (\neg A_1 \wedge A_2) \tag{3.1}$$

where R_1 is read in the following way: R_1 holds iff when the conjunction of conditions $(\neg C_1 \wedge \neg C_2)$ and actions $(\neg A_1 \wedge A_2)$ is satisfiable only if both C_1 and C_2 values are resolved to False and action A_1 is resolved to False and action A_2 is resolved to True.

As the rest of the rules is readable in the following way as R_1, the propositional logic of DT is represented as the disjunction of Rules:

$$DT := R_1 \vee R_2 \vee R_3 \vee R_4 \tag{3.2}$$

which is read: The DT is valid iff one of the given rules is satisfiable.

DT represented on Table II, can be expanded followingly:

$$\begin{aligned} DT := {} & (\neg C_1 \wedge \neg C_2 \wedge \neg A_1 \wedge A_2) \vee (\neg C_1 \wedge C_2 \wedge A_1 \wedge \neg A_2) \\ & \vee (C_1 \wedge \neg C_2 \wedge A_1 \wedge A_2) \vee (C_1 \wedge C_2 \wedge \neg A_1 \wedge A_2) \end{aligned} \tag{3.3}$$

The disadvantage of the DT is that it has fixed size and depending on its size (the number of conditions and actions) the disjunctive normal form (DNF) of DT becomes hardly readable and is takes a lot of time to process. Instead of comprehending whole DNF of DT, it is reasonable to read DT in Shannon Normal Form (SNF), which is represented using IF-THEN-ELSE operator:

$$x => y \mid g \tag{3.4}$$

is read: IF x THEN y ELSE g.

SNF is successfully implemented by Binary Decision Diagrams (BDD), introduced by Bryant in [27]. BDD is a directed acyclic graph data structure representing a boolean function. Also, BDD requires a strict variable ordering, and its leaf nodes are False and True. Figure 3.3 shows a BDD graph, where child nodes do not point on their parent nodes, thus the BDD is acyclic, i.e., no cycles in a graph. Due to acyclicity, BDDs are depicted without arrows on the end of the edges.

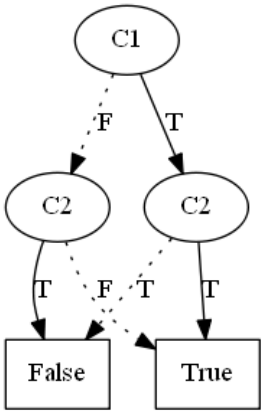

Figure 3.3. Binary Decision Diagram.

Definition 3.7: Let D be a finite set and *Var* be a finite set of Boolean variables equipped with a total ordering $< \subset Var \times Var$ [28]. 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 labeled by an element of *D*, denoted by *value(v)* [28]. Each non-terminal vertex *v* is labelled by a variable $var(v) \in Var$ and has two children $then(v), else(v) \in V$ [28]. 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* [28].

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 [28].

Multi-Terminal Binary Decision Diagram (MTBDD), so called algebraic decision diagrams, extend Binary Decision Diagram (BDD) such that they can represent functions of an arbitrary range, while their domain is still a multidimensional Boolean space [29].

Definition 3.7: Let D be a finite set and *Var* be a finite set of Boolean variables equipped with a total ordering $< \subset Var \times Var$ [28]. 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 labeled by an element of *D*, denoted by *value(v)* [28]. Each non-terminal vertex *v* is labelled by a variable $var(v) \in Var$ and has two children $then(v), else(v) \in V$ [28]. 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* [28].

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 [28].

Example 3.6: Fig. 3.3. shows an ordinary multi-terminal binary decision diagram, where a root vertex is represented by 'A' node, its low and high children represented by 'B' labeled nodes and a set of terminal nodes D = {'T0', 'T1', 'T2'}. According to Definition 3.7, this MTBDD is read by following the expression var(then(v)) > var(v) < else(var(v)):

1) for 'A': var(A) = A; then(A) = B (high);
   else(A) = B (low) => B(high) > A < B(low).
2) for 'B(low)': var(B(low)) = B(low); then(B(low)) = T0(high);
   else(B(low)) = T0(low) => T0(high) > B(low) < T0(low).
3) for 'B (high)': var(B(high)) = B(high);
   then(B(high)) = T2; else(B(high)) = T1 => T2 > B(high) < T1.

Generally, the ordering of the structure is: var(A) < var(B) < var(D), where D is the set of the terminal nodes (see Definition 3.7).

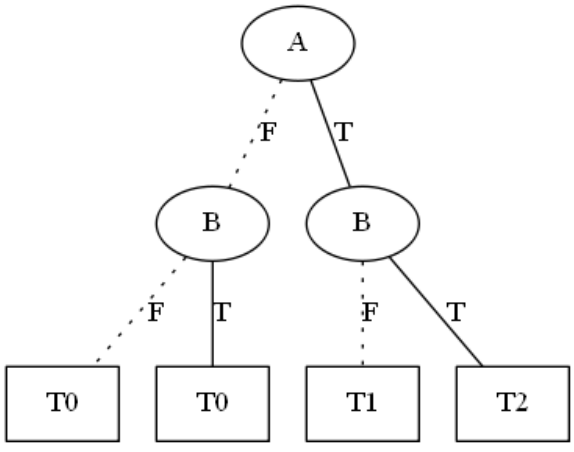

Figure 3.4. Multi-Terminal Binary Decision Diagram

The MTBDD on Figure 3.4 in SNF: MTBDD: = (A => (B => T2 | T1) | (B => T0 | T0)).

Definition 3.8: A MTBDD M is called *reduced* iff

1) for each *non-terminal vertex v* the two children are distinct, i.e. *then(v)* ≠ *else(v)* [28]. Each *terminal vertex v* has a distinct *value(v)* [28].
2) 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'* [28]. Formally, if *var(v)* = *var(v')* and *else(v)* = *else(v')* and *then(v)* = *then(v')*, then *v* = *v'* [28].

Reduced MTBDDs effectively represent DTs as a graph, which is used to generate test cases in the presented approach [28].

Example 3.7: According to the Definition 3.8, MTBDD depicted on Fig. 3.4 is a reduced representation of the Fig. 3.3 MTBDD. None of the non-terminal nodes have the same descendants. Terminal node T0 appears only once, unlike the T0 which occurs twice in Fig. 3.3 MTBDD. Another moment is the absence of the B (low) non-terminal node. This is explained by its uselessness, because both of its test cases are pointing to the same expected value T0. As the result, reduced MTBDD (rMTBDD) preserves the logic and properties embedded in Fig. 3.3 MTBDD, at the same time Fig. 3.4 rMTBDD requires the smaller number of test cases for its full coverage.

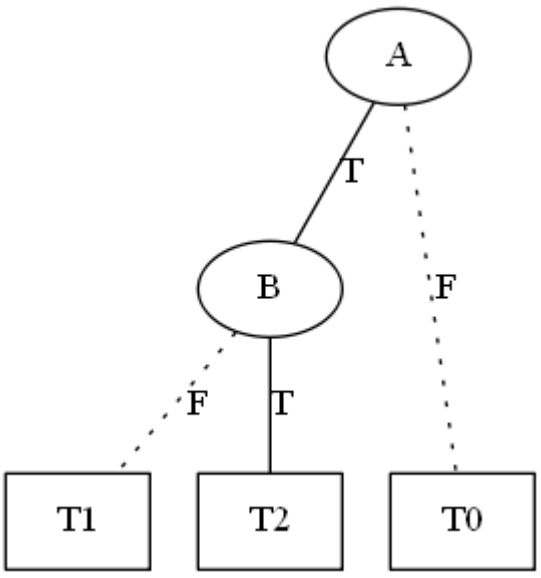

Figure 3.5. Reduced MTBDD.

The MTBDD on Figure 3.5 in SNF: MTBDD: = (A => (B => T2 | T1) | T0).

According to the definition [9] given by Meyer claims that the mutual obligation between caller and callee modules of software is called the contract. As DT represents a contract in ESG event, thereby making that event data event, the MTBDD representing a contract DT should also bear responsibility of contract. It means, that the event containing incoming edge from the current DE can be triggered iff the path starting from the non-terminal root node of MTBDD leading till the terminal node representing the next event is satisfied.

In the scope of this thesis, MTBDD is the graph representation of the contract DT in a data event of ESG. Reducing MTBDD causes the loss of conditions after conversion it back into DT. This thesis considers that the final DT can differ from the initial DT, only after the usage of mutation operators proposed in the Chapter 4.4. Another reason is that mutation operators of insertion type are unable to insert an edge or terminal node in a reduced MTBDD, therefore reducing operation cannot be applied before mutation. Hereby, we introduce the following assumption:

Assumption of MTBDD reduction: It is assumed that MTBDDs are not necessarily reduced.

3.4. Mutation Analysis

The purpose of the software testing is to find faults, for further elimination. For this purpose, a tester needs efficient test cases. The test case which is unable to detect fault is useless for testing. DT augmented ESG test cases are generated from the model itself. It means that the constructed ESG-DT is not necessarily correct. To test ESG-DT, it is important to have test cases which can bring out the model vulnerabilities. Therefore, we need to assess the efficiency of the generated test set. For this purpose, the *Mutation Analysis* is the key technique.

DeMillo et. al. in [30] first proposed the idea of mutation analysis and Budd et. al. gave an extensible explanation for it [1]. Originally it was intended as a white-box technique. The aim of mutation analysis is a generation of program's slight variations and killing them with test sets. It is said that if one test case kills all mutants then the

rest of test cases in a set are considered as effective as a killer test case. By means of mutation analysis the effectiveness of a test set is assessed.

Consider P as an original program, $L \neq Ø$ as a set of mutation operators, $U \neq Ø$ as a set of mutants, $T \neq Ø$ as a set of test sets, a mutant generator function $\Phi(P, L)$, a testing function $Q(U, t \in T)$. Then, the application of the mutation analysis requires the execution of the following steps:

1) $U = \Phi (P, L)$. Generate mutant, by inserting slight deviations in P by means of mutation operator L.
2) $Q (U, t \in T)$. Fail all $u \in U$.

If none of the $t \in T$ can distinguish a behavior of a $u \in U$ from P, then that m is considered as a living mutant or in worst case as equivalent mutant. The equivalency phenomena of $u \in U$ to the P, arouses when there is no such T the $t \in T$ can detect faulty version of P. Such mutants are detected manually.

In black box-based testing mutation analysis is applied on the program's specifications. In the scope of this thesis, the specification of program P is represented as a model M using ESG.

3.5. Mutation Operators

The purpose of the test set is being able to recognize faults in the given model. For this purpose, mutation analysis intentionally inserts different known faults in the model, thereby generates faulty models. After that faulty models are tested with the test cases in the test set. If the result of the original model tested with defined test cases is the same as the result of the faulty model, then the test set is not able to detect the fault in the faulty model. Otherwise, test set is useful because it can detect the specific inserted fault and can be used for the further testing of software. The model is the initial representation of the system under test. For test case qualification, the faulty models are generated from the *original model*. The faulty model in mutation analysis is called a *mutant.* The *faulty model* is generated by insertion of the specific fault in the original model, which is done by means of *mutation operators*. Mutation operators are

instruments representing the specific fault domain. The concise version of the given terms:

To generate mutants from model *M*, the set of mutation operators *L* is required. Mutation operator *l* ($l \in L$) *changes* the structure of *M by* generating a mutant *u* ($u \in U$). Mutation operators imitate the faults which may occur in *P*. Therefore, they are categorized by fault classes. The following group of mutation operators are used in this work.

3.5.1. ESG Mutation Operators

ESG is a graph representation of the system model and the test cases for testing ESG are generated from the model itself. Therefore, we must assess the quality of the generated test set. This sub-chapter revises the existing mutation operators for ESG mutation.

As ESG consists of events and sequences, there is a necessity to imitate the possible faults in ESG. Belli et. al. in [25] proposed the set of mutation operators for ESG mutation, where any changes in ESG results in change of whole model ESG = (V, E, Ξ , Γ) -> ESG* = (V*, E*, Ξ*, Γ*).

1) *eI* -> event insertion operator, $v \notin V$. This results in $\Phi(ESG, eI)$, where $V^* = V \cup \{v\}$ [25].
2) *e0* -> event omission operator, $v \in V$. This results in $\Phi(ESG, eO)$, where $V^* = V \setminus \{v\}$ [25].
3) *sI* -> sequence insertion, inserts sequence which is not presented in ESG [25].
4) *sO* -> sequence omission, omits sequence which is presented in ESG [25].
5) eC -> event corruption operator, substitutes an existing event with a new one. Performed by combination of {eO, eI} [25].
6) sC -> sequence corruption operator, substitutes an existing sequence with a new one. Performed by combination of {sO, sI} [25].

3.5.2. DT Mutation Operators

As DT consist of sets (C, A, R), the following are the mutation operators proposed in [24]:

1) *aI* -> action insertion, inserts a new action a ∉ A => DT = (C, A ∪ {a}, R) [24].
2) *a0* -> action omission, omits an existing action a ∈ A => DT = (C, A\{a}, R) [24].
3) aC -> action corruption, substitutes an existing action with a new one. DT = (C, A\{a} ∪ a', R) [24].
4) *cI* -> condition insertion, inserts a new condition c ∉ C => DT = (C ∪ {c}, A, R) [24].
5) *c0* -> condition omission, omits an existing condition c ∈ C => DT = (C\{c}, A, R) [24].
6) cC -> condition corruption, substitutes an existing condition with a new one. DT = (C\{c} ∪ c', A, R) [24].
7) *rI* -> rule insertion, inserts a new rule r ∉ R => DT = (C, A, R ∪ {r}) [24].
8) *r0* -> rule omission, omits an existing rule r ∈ R => DT = (C, A, R\{r}) [24].
9) rC -> rule corruption, substitutes an existing rule with a new one. DT = (C, A, R\{r} ∪ r') [24].

CHAPTER 4

CONTRACT-BASED MUTATION OPERATORS FOR DECISION-TABLE-AUGMENTED EVENT SEQUENCE GRAPH

4.1. MTBDD-based Mutation Operators for ESG-DT

Apart from ESG and DT mutation operators described in the sections **3.5.1** and **3.5.2** respectively, this thesis proposes new mutation operators for MTBDDs. In Chapter 3.3, the term MTBDD, which is a contract representation of MTBDD, is defined. Khalilov et. al. introduced the mutation operators for Ordered Binary Decision Diagrams (OBDD) in [4]. The OBDD is the a BDD with a strict variable ordering [27]. Therefore, it also implements the SNF. This thesis uses the mutation operators [4] for MTBDD mutation and proposes a set of new operators. The idea of MTBDD mutation remains the same, i.e., executional part consists of two and more terminal nodes and they are attached to the non-terminal nodes same as in OBDD by incoming edges.

According to the [4], node omissions of OBDD can be applied at the terminal and non-terminal levels and they will produce different results. Since the thesis concentrates on the mutation of ESG-DT, the main interest in MTBDD are terminal nodes, because they represent the corresponding events in the model. The present thesis concentrates on omission of the terminal node in MTBDD by *tnO* operator, which in turn produces new DT without corresponding action. This operation produces a new ESG-DT without edge from the current DE to the event represented by the omitted action. Meanwhile, [4] claims that the insertion of the terminal node is possible only if the corresponding non-terminal node at the level adjacent to the terminal has zero or one child. Otherwise, it becomes impossible to insert a terminal node, because the maximum number of outgoing edges of non-terminal nodes in OBDD is two, therefore, due to the limitation of the number of

children a third child addition is an invalid operation. Since the MTBDD is an extension of the OBDD, terminal node insertion takes place due to *tnI* mutation operator.

Since MTBDD is a graph, manipulations can be performed not only on its nodes (vertices), but also on its edges (arcs). This is inspired by [25], where graph-based structures such as DG, FSM, SC and ESG have been changed by removing or adding not only theirs vertices (nodes), but also arcs (edges). Hence, thesis introduces *edgeI* and *edgeO* operators for MTBDD edge related mutations, which are semantically equal to arc, sequence and transition mutation operators in [25].

Both can be applied on any MTBDD edge, but we focus on terminal nodes' incoming edges, because edits on edges at the higher levels will drastically change the resulting contract. *edgeO* operator removes one edge between non-terminal and terminal nodes. This change will reduce the rule number of the final DT by one rule. In case if a terminal node is attached to non-terminal only by one edge (has one incoming edge), *edgeO* will totally disconnect it from MTBDD structure. Consequently, the finite DT will lack not only one rule, but also one action. The opposite to *edgeO*, *edgeI* creates one new connection between existing terminal and non-terminal nodes. The DT obtained afterwards, acquires one new rule, with a corresponding enabled action. The restriction imposed on usage of *edgeI* is the same as for the *tnI*. Again, in MTBDD non-terminal nodes have limitations on the number of children nodes, which cannot exceed two. Consequently, those non-terminal nodes which have one child or have not got any, can end the path passing through it by the *edgeI* application.

The representation of MTBDD diminishes the size of MTBDD structure and decreases the required test set size for it. The drawback of reducing MTBDD is the impossibility of *tnI* and *edgeI* usage. The terminal node corruption (*tnC*) and edge corruption (*edgeC*) mutation operators fix this disadvantage. The essence of corruption mutation operators is in replacement of the existing parts of MTBDD structure. The *edgeC* allows to replace existing edge with a new one, simply by removing it with *edgeO* and using *edgeI* operator. The *edgeC* can redirect existing edge to the different terminal node. Mutation operator *tnC* replaces the existing terminal node with a new terminal node, by preserving all incoming edges of the previous terminal node for a new node. The edge switcher (*edgeS*) mutation operator switches the outgoing edges of the non-terminal node by involving the application of *edgeC* operator twice for each edge. First, it removes both outgoing edges, then descendant nodes get the opposite valued

edges. This is explained by the inability of the *edgeC*'s single usage for inverting the edge value, because non-terminal nodes cannot have equal values outgoing edges.

Reducing of MTBDDs leads to the omission of the redundant nodes. The subsequent DT after this operation may have decreased number of conditions, although the mutations are performed on terminal nodes, not on non-terminal ones.

A summary on the MTBDD mutation operators is given below:

1) *tnI* (nodeI) -> terminal node insertion operator. Inserts a new terminal node in MTBDD by connecting it with a new edge [4]. Consider *newTNode* as a new terminal node and *NTNode* as an existing non-terminal node, then *tnI*(MTBDD, newTNode, NTNode) => MTBDD* = (C ∪ (A ∪ {newTNode}), E ∪ {NTNode, newTNode}). Fig. 4.1 represents MTBDD and Fig. 4.2 *MTBDD after being mutated by *tnI*.

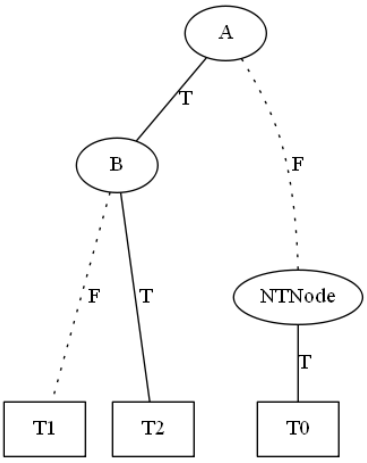

Figure 4.1. MTBDD before the *tnI* application.

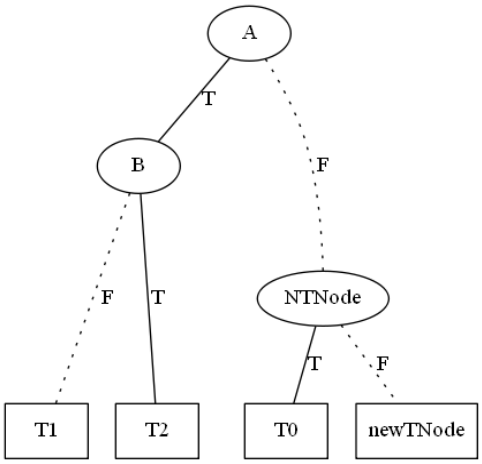

Figure 4.2. MTBDD* after the application of *tnI*.

2) *tnO* (nodeO) -> terminal node omission operator. Omits an existing terminal node from MTBDD and subsequently all of its incoming edges [4]. Consider *TNode* as a terminal node being removed and its incoming edge set E_r, which is $E_r \in E$, then *tnO*(MTBDD, TNode) => MTBDD* = $(C \cup (A \setminus \{TNode\}), E \setminus E_r)$. Fig.4.3 represents the original MTBDD before being mutated by *tnO* and the mutated *MTBDD is depicted on Fig.4.4.

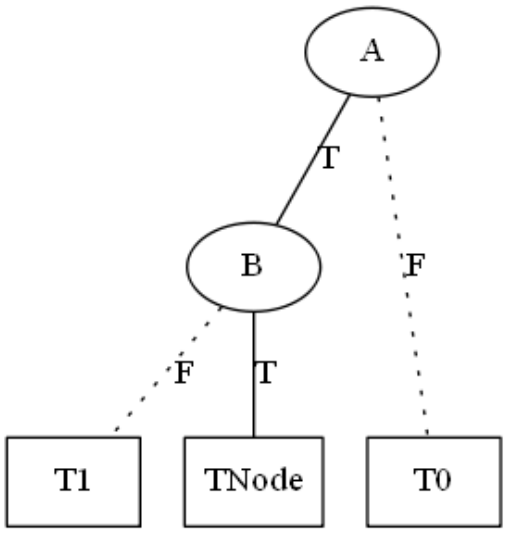

Figure 4.3. MTBDD before the *tnO* application.

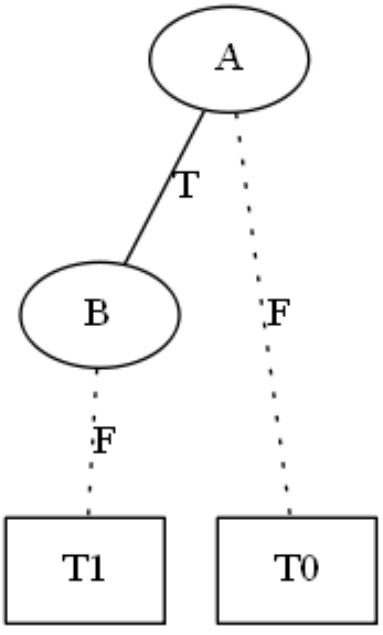

Figure 4.4. MTBDD* after the application of *tnO*.

3) *tnC* -> terminal node corruption. Replaces the existing terminal node with a new one, by preserving all incoming edges of the old node for the new node. Consider TNodeOld and TNodeNew as old and new terminal nodes, respectively. Then *tnC*(MTBDD, TNodeOld, TNodeNew) => MTBDD* = (C ∪ A\{TNodeOld} ∪ {TNodeNew}, E). *MTBDD shown on Fig.4.6 is a mutant derived after mutation of Fig.4.5. MTBDD by *tnC*.

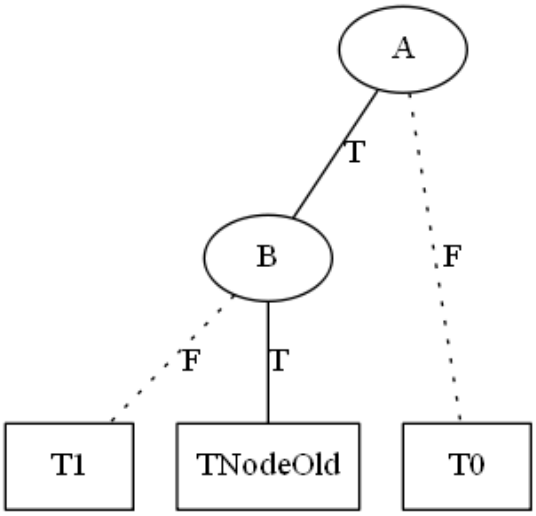

Figure 4.5. MTBDD before the *tnC* application.

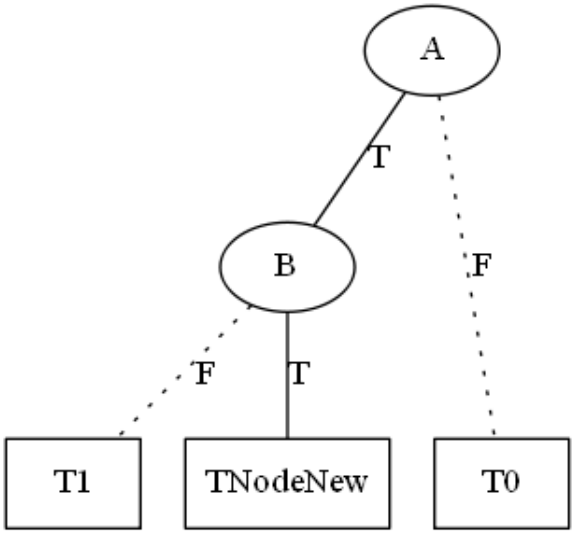

Figure 4.6. MTBDD* after the application of *tnC*.

4) *edgeI* -> edge insertion. Inserts a new edge, by connecting an existing terminal node to the non-terminal node. Consider NTNode as an existing non-terminal

node which has only one descendant and TNode as an existing terminal node. *edgeI*(MTBDD, NTNode, TNode) => MTBDD* = (C ∪ A, E ∪ {NTNode, TNode}). The *edgeI* operator explicitly demonstrates the result of edge insertion to the original MTBDD on Fig.4.7. and as a result, obtaining mutant depicted on Fig.4.8.

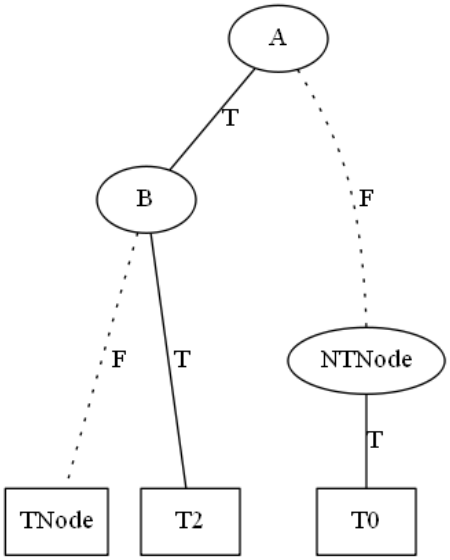

Figure 4.7. MTBDD before the *edgeI* application.

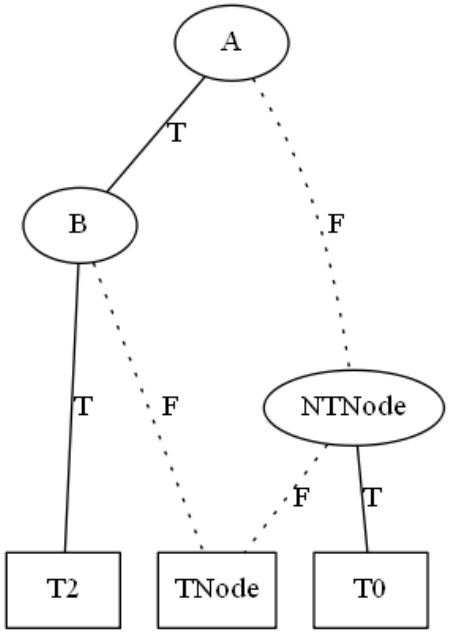

Figure 4.8. MTBDD* after the application of *edgeI*.

5) *edgeO* -> edge omission. Omits an existing incoming edge of a certain terminal node, may cause in total disconnection of a terminal node from MTBDD.

Consider NTNode as an existing non-terminal node and TNode as an existing terminal node. *edgeO*(MTBDD, NTNode, TNode) => MTBDD* = (C ∪ A, E \ {NTNode, TNode}). The result of the edge removal by *edgeO* is shown clearly on Fig.4.10.

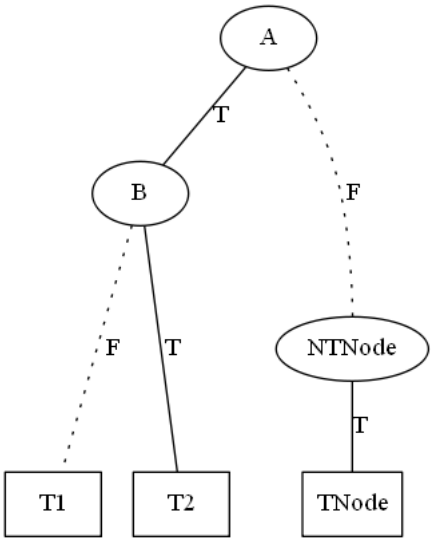

Figure 4.9. MTBDD before the *edgeO* application.

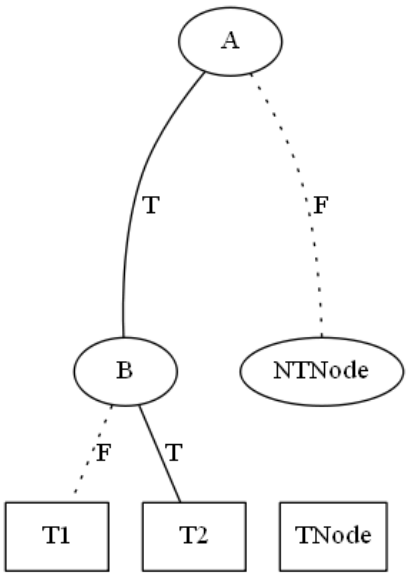

Figure 4.10. MTBDD* after the application of *edgeO*.

6) *edgeC* -> edge corruption. Redirects an existing outgoing edge of a certain non-terminal node from one terminal node to another one. The edge corruption operator involves the execution of *edgeO* and *edgeI*. Consider TNode1 and TNode2 as terminal nodes and NTNode as nonterminal node, where TNode1 has 2 incoming edges. If apply *edgeC* on the {NTNode, TNode1} edge and redirect it

to the TNode2, the resulting MTBDD: *edgeC*(MTBDD, NTNode, TNode1, TNode2) => MTBDD* = (C ∪ A, E \ {NTNode, TNode1} ∪ {NTNode, TNode2}), for redirecting. The Fig. 4.11 shows the outcome (Fig.4.12) of the *edgeC* operator application on the structure on Fig.4.11.

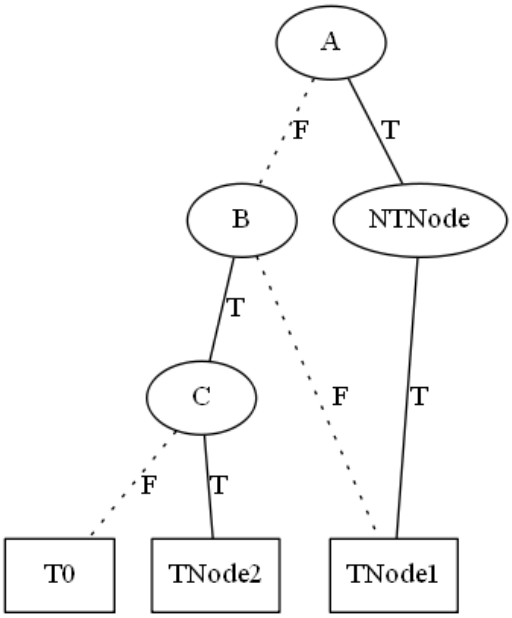

Figure 4.11. Case 1: MTBDD before the *edgeC* application.

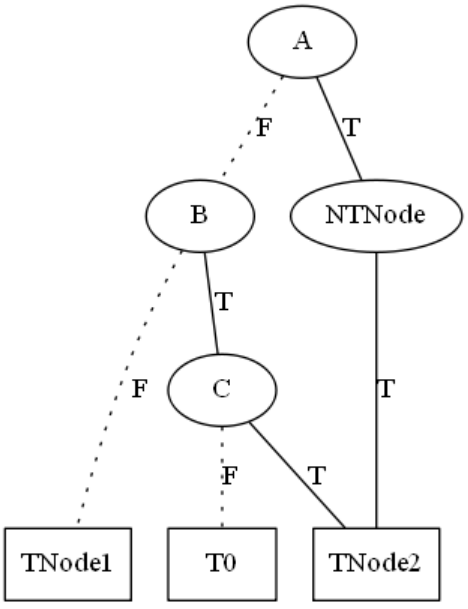

Figure 4.12. Case 1: MTBDD* after the application of *edgeC*.

In case if TNode1 has only one incoming edge then edgeC in redirecting operation will totally disconnect TNode1 from reduced MTBDD structure. edgeC(MTBDD, NTNode, TNode1, TNode2) => MTBDD* = (C ∪

A\{TNode1}, E \ {NTNode, TNode1} ∪ {NTNode, TNode2}). This case is explicitly shown on Fig.4.13 and the result on Fig.14.

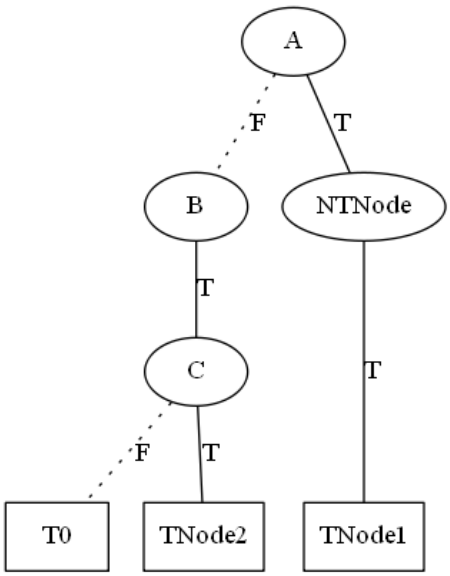

Figure 4.13. Case 2: MTBDD before the *edgeC* application.

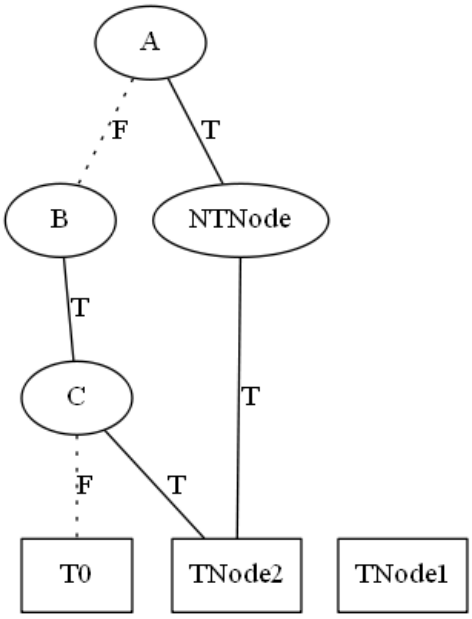

Figure 4.14. Case 2: MTBDD* after the application of *edgeC*.

7) *edgeS* -> edge switcher. Switches the outgoing edges of the existing non-terminal node, so that its descendants get the inverted valued edges. This operation involves the application of two *edgeC* operators. *edgeS*(MTBDD, NTNode, TNode1, TNode2) => MTBDD* = (C ∪ A, E \ {{NTNode, TNode1}, {NTNode,

TNode2}} ∪ {!{NTNode, TNode1}, !{NTNode, TNode2}}). The result of *edgeS* mutation operator usage is represented on Fig.4.16.

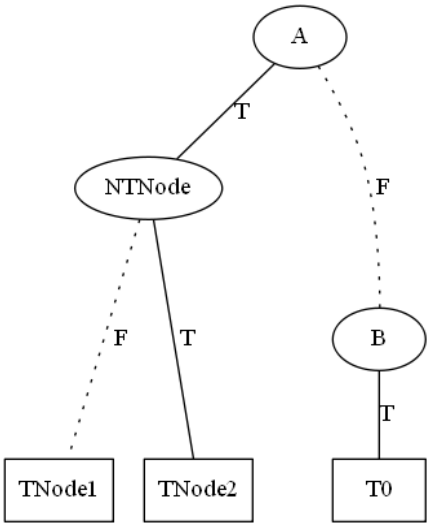

Figure 4.15. MTBDD before the *edgeS* application.

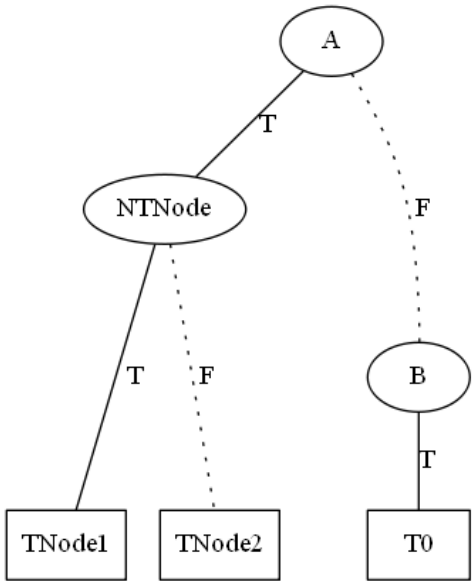

Figure 4.16. MTBDD* after the application of *edgeS*.

From the above-mentioned formulations of proposed MTBDD mutation operators, the need in operators' categorization arises. As mutation operators intend to change the original specifications either slightly or considerably, we place operators into two groups: *Simple Mutation* and *Composed Mutation*:

Definition 4.1: Simple Mutation group operators aim to modify MTBDD by either removing or adding **one** component (edge or terminal node) once per mutation.

Definition 4.2: Composed Mutation group operators will rearrange MTBDD component or components. This action involves the application of two mutation operators at least one time from Simple Mutation group.

Table 4.1. Composition of mutation operators.

Mutation Operators	Simple Mutation	Composed Mutation
tnI	+	
tnO	+	
tnC		+
edgeI	+	
edgeO	+	
edgeC		+
edgeS		+

Based on the Definitions 4.1 and 4.2, the Table 4.1 clearly separates these two categories. The simple mutation operator set is {*tnI*, *tnO*, *edgeO*, *edgeI*} and {*tnC*, *edgeC*, *edgeS*} are elements of the composed mutation operators set. As *tnI* (Fig. 4.17), *edgeI* (Fig. 4.18) just insert corresponding components per mutation and *tnO* (Fig. 4.19), *edgeO* (Fig. 4.20) omit corresponding components without involvement of any other operation, hence they belong to Simple Mutation group. On the other hand, *tnC* and *edgeC* involve corresponding insertion and omission operators by replacing corresponding components. The *tnC* operator first removes the corrupted terminal node by means of *tnO*, then engages *tnI* to insert a "fresh" terminal node, which is a new expected result. Fig. 4.21 and Fig. 4.22 clearly explicitly shows steps of *tnC* and *edgeC* execution corresponding ESGs. Finally, the *edgeS* operator targets on switching outgoing edges of nonterminal node, where at least one edge attaches the terminal node. This operation involves performing of two *edgeI* and two *edgeO* operators. Initially, edges are removed by two *edgeO* operators, then inserted into opposite positions by the execution of *edgeI* operators. This process is shown on Fig. 4.23 by ESG of *edgeS*.

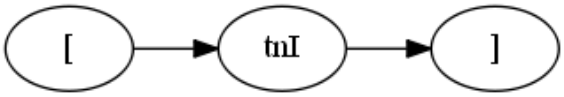

Figure 4.17. ESG of *tnI* operator.

Figure 4.18. ESG of *edgeI* operator.

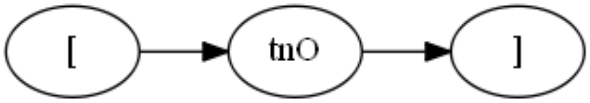

Figure 4.19. ESG of *tnO* operator.

Figure 4.20. ESG of *edgeO* operator.

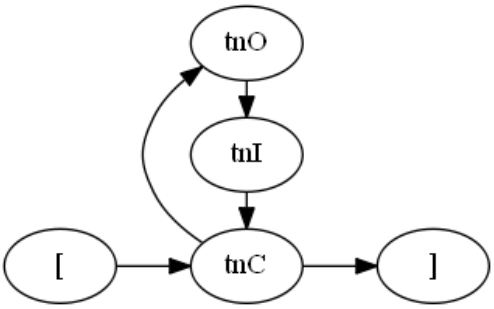

Figure 4.21. ESG of *tnC* operator.

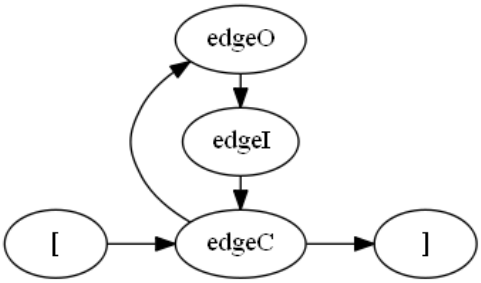

Figure 4.22. ESG of *edgeC* operator.

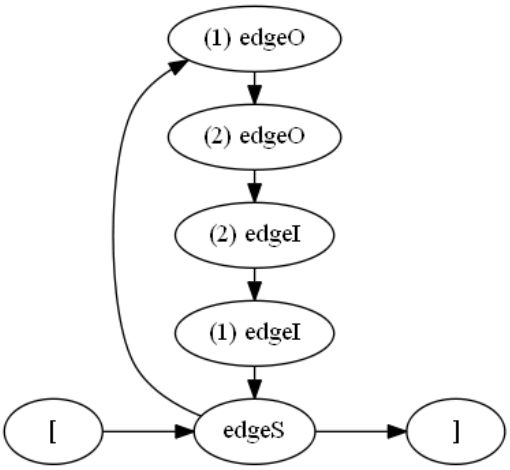

Figure 4.23. ESG of *edgeS* operator.

4.2. Mutant Generation

As the research field of this thesis is DT-augmented ESGs, all mutations are performed on the level of data events (DEs), i.e., Event with DT (EwDT). Considering that DT in ESG represents a refined event, based on input values passed to it, triggers next predefined successor, DT and MTBDD mutation operators provided in 3.5.2 and 4.1.1, respectively, provide a more accurate way of managing the sequence flow.

To get a mutated ESG, by mutating event with DT it is necessary to aim on

1) complete removal of any notion of the action in all rules which leads to the omission of the corresponding sequence in ESG.
2) complementing DT action set with a new action which should be activated in at least one, possibly new, rule, which leads to the new sequence in ESG.
3) application in strict sequence of the ways 1, 2 to change the direction of sequence, also leads to the creation of the new ESG.

Meanwhile, obtaining a new ESG, by mutating event with MTBDD it is enough to perform:

1) completely detach a terminal node from MTBDD, which is responsible for certain sequence. Hence, this operation effects on existence of certain sequence in ESG. This can be achieved by either removal of the terminal node's only incoming edge or complete omission of the terminal node.

2) insert, if possible, a new terminal node into MTBDD, which creates a new sequence in ESG.

Here it can be concluded that, the sequences of ESG depends on connectivity of the terminal nodes to the MTBDD.

As thesis relies on simple mutations, i.e., one mutation applied by one mutation operator at a time, the DT based event mutation may not always produce a new ESG, different from an original one. This can be explained by cases such that application of *rO* operator not necessarily completely disables an action in DT or *aI* always requires *rI* operator. MTBDD mutation, on the other hand, can provide simple mutation by insertion of action element without requiring another MTBDD mutation operator. Therefore, MTBDD is considered as an extension for DT mutations.

To start mutating MTBDD based event is it important to convert MTBDD into DT and after mutation convert it back into DT. By using a DT mutation operator and newly proposed MTBDD mutation operators research offers an algorithm for generation of faulty models.

The aim of mutation analysis is measuring the ability of fault detection of the test cases. Test cases in DT augmented ESG are represented as CES. To generate new CESs from mutants of DT augmented ESG, the generation of a faulty models are required. Considering Δ as a DT and MTBDD mutation operators set, where Δ = *{rI, rO, tnI, tnO, edgeI, edgeO}*, DT augmented ESG as an original model *M*, MM as a mutated model set, which is MM = {MM_1, MM_2, …, MM_k}, test cases generated from M as T = {CES_1, CES_2, …, CES_n} and MM based test set as MT = {CES^*_1, …, CES^*_m}, the algorithm 1 describes thoroughly a method of mutant generation and subsequently the test case generation.

Algorithm 1.

Input: M: = DT augmented ESG.

Output: The quality of the generated CES and FCES.

1.BEGIN

2. Generate CESs and FCESs from M.

3. FOREACH DE in M.

4. Convert DE into MTBDD

5. FOREACH MTBDD mutation operator from Δ set

6. BEGIN

7. MM = Δ^{MTBDD}(M);

8. END

9. END

10. FOREACH MM* in MM

11. FOREACH CES* in CES test set

12. BEGIN

13. If MM* fails CES*

14. Test passed.

15. END

16. FOREACH FCES* in FCES test set

17. BEGIN

18. If MM* passes FCES*

19. Test passed

20. END

21. END

Example for mutant generation: ESG-DT on Fig. 4.24, consists of events *a*, *b*, *c,* and *d*, where only event *a* is a data event, i.e., has a DT.

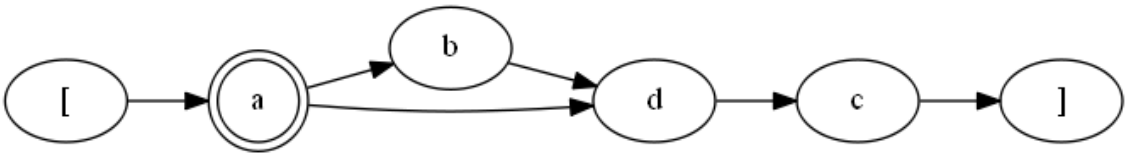

Figure 4.24. Dummy ESG.

A DT in the event *a* is shown on Table 4.2, consists of two rules, two conditions and two actions representing corresponding events in ESG-DT on Fig. 4.24. The corresponding MTBDD of Table 4.2 DT is depicted on Fig. 4.25.

Table 4.2. DT "a" of event *a*.

DT a	Rules	
	R1	R2

(cont. on next page)

Table 4.2 (cont.)

Condition	C1	F	T
	C2	T	F
Action	b	X	
	d		X

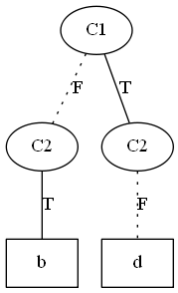

Figure 4.25. MTBDD contract representation in “a” DE.

According to Algorithm 1, the event which has a contract is picked for the further mutation. Therefore, the DT on Table III, which is contract in the event *a* is used for the further mutation process. Before mutation starts, it should be transformed into MTBDD (Fig. 4.25), on which the proposed in Chapter 4.1 mutation operators are applied only once. For example, here *tnI* operator is applied on Fig. 4.25. MTBDD adds **c** node, which represents event *c*. The result of application is on Fig. 4.26. Table 4.3 DT is obtained from transformation of MTBDD into DT form.

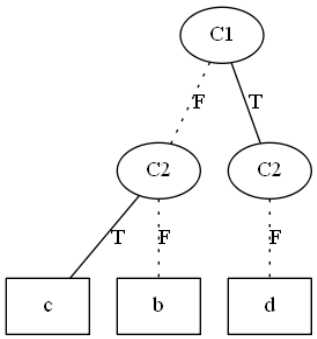

Figure 4.26. Mutated MTBDD of contract in DE “a”.

Table 4.3. Mutated “a” DT.

DT a		Rules		
		R1	R2	R3
Condition	C1	F	T	F
	C2	T	F	F
Action	b	X		
	d		X	
	c			X

The final ESG-DT will look as it is depicted on Fig. 4.27:

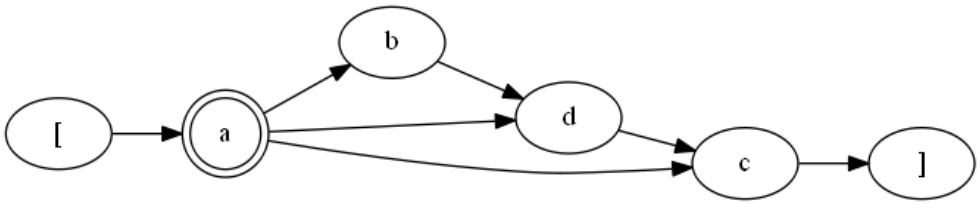

Figure 4.27. Final ESG-DT mutant.

Figure 4.28 shows same mutant as in Fig. 4.27 but shows the changes which took place after mutation of MTBDD. The change is represented by the dashed edge, outgoing from event *a* to event *c*. Further changes in ESG-DT models are represented by dashed and dotted lines, which represent inserted and omitted edges, respectively.

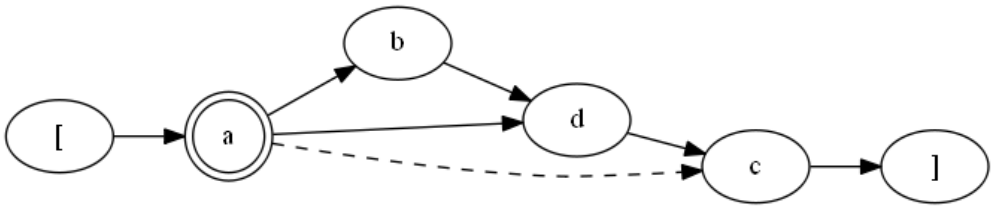

Figure 4.28. Changes of the resulting final ESG-DT mutant.

Algorithm 1 is the extended and modified version of the “Algorithm 2. K-Robustness testing process” from [24]. Like its predecessor, Algorithm 1 also involves mutation operators for model mutation and generates test cases represented in CES and

FCES forms from the obtained mutants. The modification includes the enlarged list of mutation operators for seeding faults into MTBDD and additional mutation of the ESG-DT by converting DT in DE into MTBDD.

4.3. Equivalent Mutants

As the side effect of mutation testing is equivalent mutants, which cannot be distinguished by any test set, we have chosen certain mutation operators based on assumption given in Chapter 4.2. As the manipulations with sequences are performed on DE level, we claim that the mutations performed in following cases will always produce non-equivalent mutants:

1) If action in DT is active only in one rule, then omission of that rule completely removes the corresponding sequence in DT augmented ESG.
2) If terminal node in MTBDD has only one incoming edge, then omission of that edge will completely remove action and corresponding rule from DT and as consequence whole sequence from DT augmented ESG.
3) The omission of terminal node in MTBDD will also remove all its incoming edges and in resulting DT corresponding action with all corresponding rules will disappear. As a result, it leads to the omission of respective sequence.
4) The insertion of a new terminal node in MTBDD will add a new edge and a final DT will have a new action and a new corresponding rule. Finally, DT augmented ESG will have one new sequence.

CHAPTER 5

EVALUATION

This chapter provides an evaluation of ESG-DT mutation analysis for the newly developed mutation operators in this thesis on three cases, namely CD player, Cruise Control, and Simple Automated Teller Machine (ATM). Initially, each of these cases has its original model, which is consistent with the corresponding original system. The models of these systems are represented as DT augmented ESGs. As mutant generation being the main part of mutation analysis, this thesis involves the application of the proposed MTBDD operators along with the proposed DT mutation operators. The instruction for obtaining mutants and generating test sequences from them are described in Algorithm 1.

It is necessary to note the following representations on ESG-DT mutants:

- The omitted edges in ESG-DT mutants are represented as dotted arrows - - ►.
- The inserted edges in ESG-DT mutants are represented as dashed arrows - – ►.

5.1. CD Player

5.1.1. CD Player ESG-DT Model

The ESG-DT model of CD player [31] is presented as the first case. Fig. 5.1 model contains five nodes, where *stop, play, load* are events with DT and nodes *pause* and *off* are simple nodes without contracts inside.

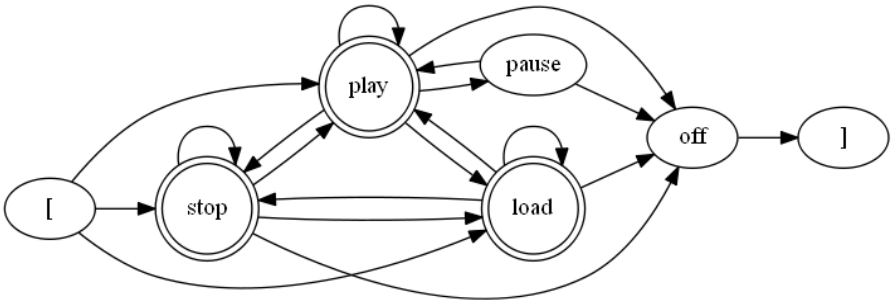

Figure 5.1. CD Player ESG-DT.

Table 5.1 shows the "stop" event DT, which corresponds to the *stop* node of ESG-DT in Fig. 5.1. For instance, it indicates that, if the first condition "offButtonPressed" is resolved to be *true*, then, no matter of what the value of the other conditions will be, the *off* action will be triggered. Therefore, in ESG-DT model, the next possible event is certainly the *off* event. The "load" event DT in Table 5.2 represents the *load* event of ESG-DT in Fig. 5.1.

Table 5.1. The original "stop" DT.

stop		Rules				
		R0	R1	R2	R3	R4
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	F	T	T	T
	Cdpresent	-	-	F	T	T
	lastTrackPlayed	-	-	-	F	T
Actions	play				X	
	stop			X		X
	load		X			
	off	X				

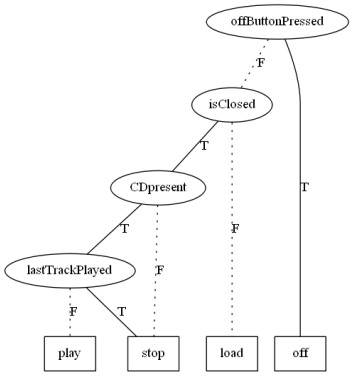

Figure 5.2. MTBDD of the original "stop" DT.

Table 5.2. The original “load” DT.

load		Rules				
		R0	R1	R2	R3	R4
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	F	T	T	T
	CDpresent	-	-	F	T	T
	lastTrackPlayed	-	-	-	F	T
Actions	play				X	
	stop			X		X
	load		X			
	off	X				

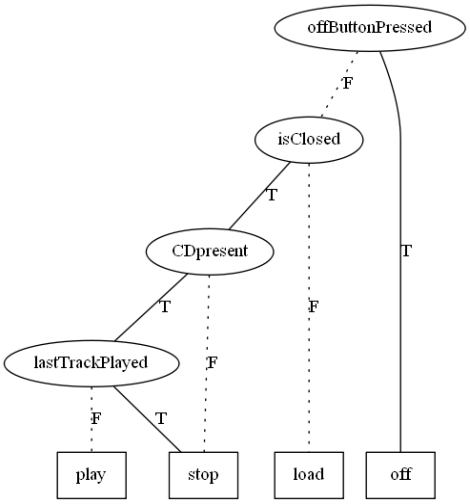

Figure 5.3. MTBDD of the original “load” DT.

Table 5.3. The original “play” DT.

play		Rules					
		R0	R1	R2	R3	R4	R5
Conditions	offButtonPressed	T	F	F	F	F	F
	isClosed	-	F	T	T	T	T
	CDpresent	-	-	F	T	T	T
	lastTrackPlayed	-	-	-	F	F	T
	pauseButtonPressed	-	-	-	F	T	-
Actions	play				X		
	pause					X	
	stop			X			X
	load		X				
	off	X					

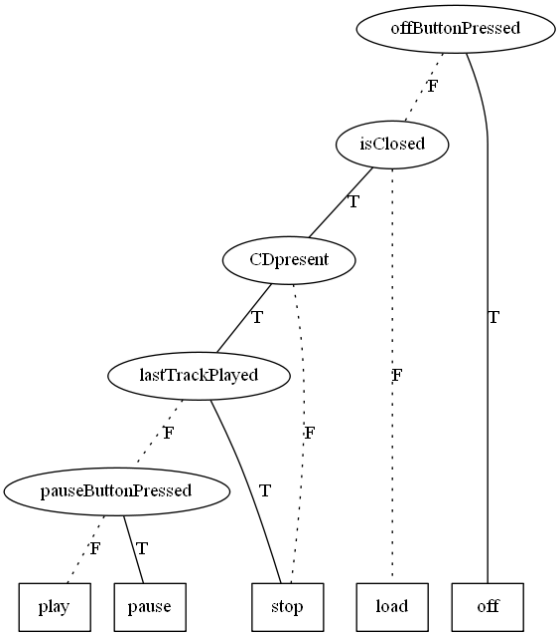

Figure 5.4. MTBDD of the original “play” DT.

The test sequences are generated from the original ESG of the CD player. As the model is in the ESG form, the corresponding test sequences will be in the form CES and FCES. The tool Test Suite Designer (TSD) [3], [32] generates test sequences from the ESG in Fig. 5.1 without taking DTs into consideration. This may be considered as generating test sequences with respect to branch coverage from source code, although condition coverage is possible but costly. TSD provides four CESs consisting of 20 events (given in Table 5.4) and twelve FCESs consisting of 30 events (given in Table 5.5).

Table 5.4. CESs of CD Player ESG-DT.

5 nodes CES:	[, play, play, pause, play, off,]
5 nodes CES:	[, stop, load, stop, stop, off,]
3 nodes CES:	[, load, load, off,]
7 nodes CES:	[, play, stop, play, load, play, pause, off,]

Table 5.5. FCESs of CD Player ESG-DT.

2 nodes FCES:	[, stop, pause,
3 nodes FCES:	[, play, pause, stop,
3 nodes FCES:	[, play, pause, pause,
3 nodes FCES:	[, play, pause, load,
2 nodes FCES:	[, load, pause,
3 nodes FCES:	[, play, off, stop,
3 nodes FCES:	[, play, off, play,
3 nodes FCES:	[, play, off, pause,
3 nodes FCES:	[, play, off, load,
3 nodes FCES:	[, play, off, off,
1 nodes FCES:	[, pause,
1 nodes FCES:	[, off,

5.1.2. CD Player Mutation Analysis

According to the Algorithm 1, the mutation process starts by traversing all events containing DTs. In case of CD player, they are *stop*, *play*, and *load*, which means that operations will be performed on these nodes. To show how changes in DT influence on the resulting ESG-DT we apply decision table mutation operators. In thesis, only *rI* and *rO* operators are involved for DT mutation.

Table 5.6. “stop” DT after the application of *rO*.

Stop		Rules			
		R0	R1	R2	R3
Conditions	offButtonPressed	T	F	F	F
	isClosed	-	F	T	T
	Cdpresent	-	-	F	T
	lastTrackPlayed	-	-	-	F
Actions	play				X
	stop			X	
	load		X		
	off	X			

Table 5.7. “load” DT after the application of *rO*

load		Rules			
		R0	R1	R2	R4
Conditions	offButtonPressed	T	F	F	F
	isClosed	-	F	T	T
	CDpresent	-	-	F	T
	lastTrackPlayed	-	-	-	T
Actions	play				
	stop			X	X
	load		X		
	off	X			

Although the fourth rule has been omitted in Table 5.6 DT, all actions are working in the remained rule, therefore, the resulting graph (Fig. 5.5) is not changed. In case of Table 5.7. DT, the rule R3 is omitted, where the action *play* was working, action play became inactive. Therefore, the resulting Fig. 5.6 ESG-DT does not have the edge from “*load*” to “*play*” event.

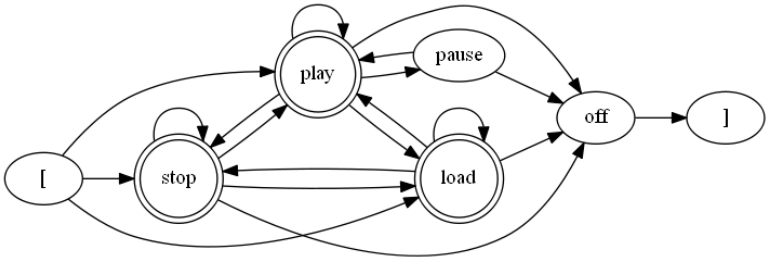

Figure 5.5. The mutant ESG-DT obtained after the application *of rO* on “stop” DT (note that it is equivalent to the original ESG layer).

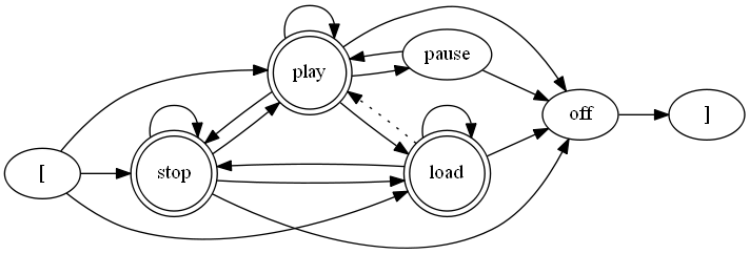

Figure 5.6. The mutant ESG-DT derived after the application of *rO* on “load” DT.

The application of the *rO* on the R4 rule of Table 5.3 DT leads to the DT in Table 5.8, which generated the mutated ESG in Fig. 5.7, with the omitted edge from *play* to *pause*.

Table 5.8. "play" DT after the application of *rO*.

play		Rules				
		R0	R1	R2	R3	R5
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	F	T	T	T
	CDpresent	-	-	F	T	T
	lastTrackPlayed	-	-	-	F	T
	pauseButtonPressed	-	-	-	F	-
Actions	play				X	
	pause					
	stop			X		X
	load		X			
	off	X				

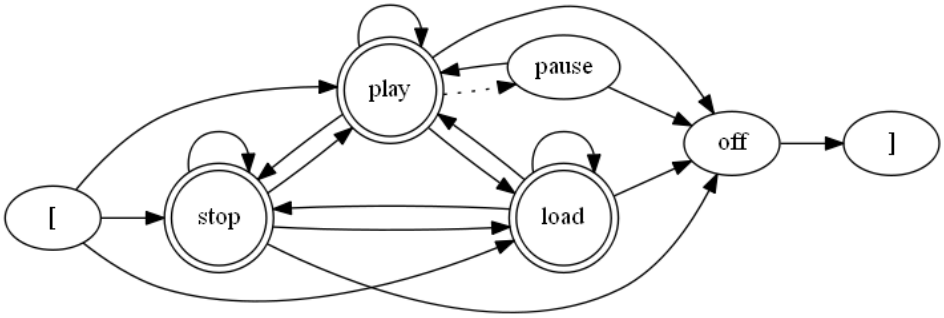

Figure 5.7. The ESG-DT mutant obtained after the application of *rO* on the "play" DT.

Mutant ESGs generated by DT mutation are shown above. Now, we convert the original DTs into MTBDD form and generate mutant ESGs using MTBDD mutation operators proposed in this thesis. The Fig. 5.4 represents MTBDD of the DT in Table 5.3. Here, the transformed contracts (from DT to MTBDD) are mutated by newly proposed mutation operators. This step embraces the application of all MTBDD mutation operators. The obtained mutated MTBDDs are converted back into DT representation form so that the effect of mutation can be reflected on the ESG.

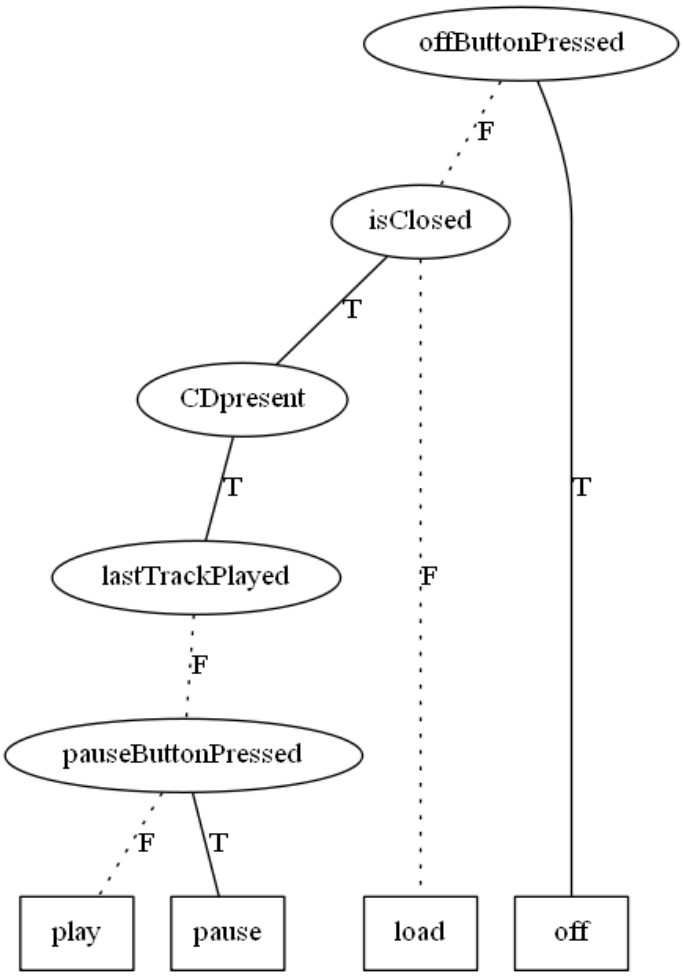

Figure 5.8. “play” MTBDD after the application of the *tnO*.

Table 5.9. “play” DT the application of *tnO* on the “play” MTBDD.

play		Rules			
		R0	R1	R2	R3
Conditions	offButtonPressed	F	F	F	T
	isClosed	T	T	F	-
	Cdpresent	T	T	-	-
	lastTrackPlayed	F	F	-	-
	pauseButtonPressed	F	T	-	-
Actions	play	X			
	pause		X		
	load			X	
	off				X

As there is no rule with active “stop” action in the Table 5.9 DT, the edge going from event *play* to event *stop* must be omitted and the model on Fig. 5.1 becomes ESG-DT on Fig. 5.9.

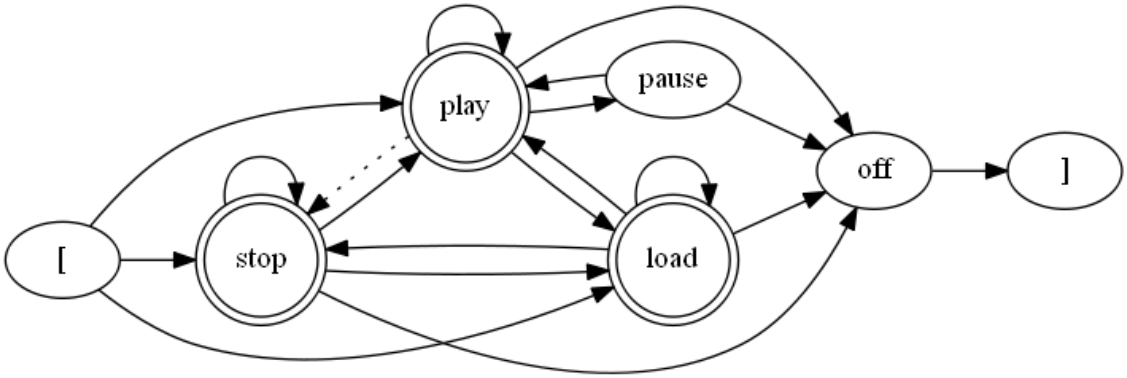

Figure 5.9. “CD Player” ESG-DT after the *tnO* usage.

As the FCESs of the original model are considered to be faulty models [26], FCES is said to be efficient if it is able to distinguish the mutant by matching to it. Matching means that FCES exists as a part of an existing path in an ESG.

According to the ESG mutation testing, the generated mutant is tested with CESs. A CES distinguishes a mutant only if the CES does not match to the mutant, which means that there is at least one difference between the CES and the event sequence of the mutant. If a given ESG does not contain same event sequences as a given CES, then the CES is not sensitive enough to recognize ESG as mutant. If none of the CESs in the test set can distinguish the mutant, then this test set is said to be not sensitive enough or ineffective. For instance, the mutant shown in Fig. 5.9 fails the *[, play, stop, play, load, play, pause, off,]* CES. At the same time, FCES test set cannot distinguish this mutant from the original, because the mutant shows the same behavior during testing as the original graph.

The *edgeO* mutation operator transforms the original MTBDD in Fig. 5.4 into the one in Fig. 5.10.

Table 5.10. “play” DT the application of *edgeO* on the “play” MTBDD.

play		Rules				
		R0	R2	R3	R4	R5
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	T	T	T	T
	CDpresent	-	F	T	T	T
	lastTrackPlayed	-	-	F	F	T
	pauseButtonPressed	-	-	F	T	-
Actions	play			X		
	pause				X	
	stop		X			X
	off	X				

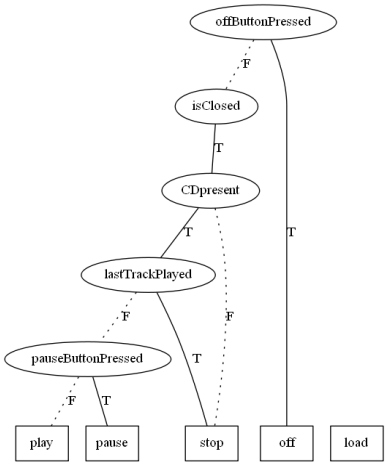

Figure 5.10. “play” MTBDD after the application of the *edgeO*.

The DT presented in Table 5.10 have not got the load action, therefore, it considers the original model on Fig.1 without the edge between *play* and *load* events, as shown on Fig.5.11:

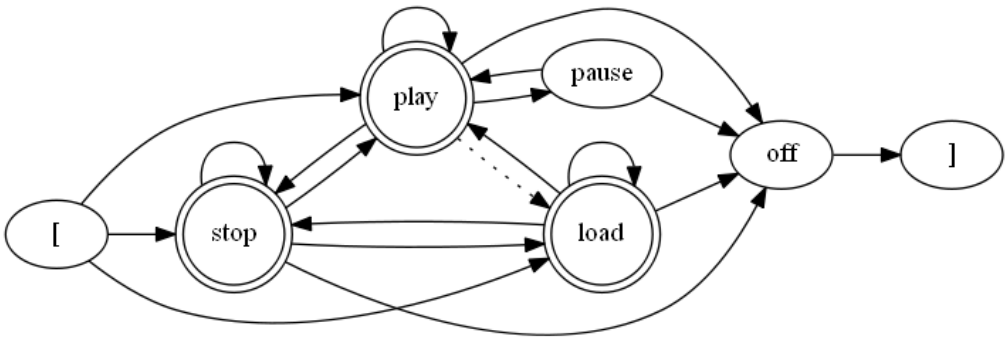

Figure 5.11. “CD Player” ESG-DT after the *edgeO* usage.

The mutant on Fig. 5.11 has omitted edge from event *play* to *load*. This happened due to application of *edgeO* mutation operator on MTBDD on Fig. 5.4, which results in Fig. 5.10. The corresponding ESG-DT mutant is distinguished only by *[, play, stop, play, load, play, pause, off,]* CES of original model. This is the same test sequence from the same test set applied on the mutated model in Fig. 5.9. FCESs also cannot distinguish the graph of the mutant ESG-DT from the original.

As the *eC* operator redirects the edge from one terminal node to another, the resulting mutated "play" DT and MTBDD are

Table 5.11. "play" DT the application of *edgeC* on the "play" MTBDD.

play		Rules					
		R0	R1	R2	R3	R4	R5
Conditions	offButtonPressed	T	F	F	F	F	F
	isClosed	-	F	T	T	T	T
	CDpresent	-	-	F	T	T	T
	lastTrackPlayed	-	-	-	F	F	T
	pauseButtonPressed	-	-	-	F	T	-
Actions	play				X		
	stop			X			X
	load		X				
	off	X				X	

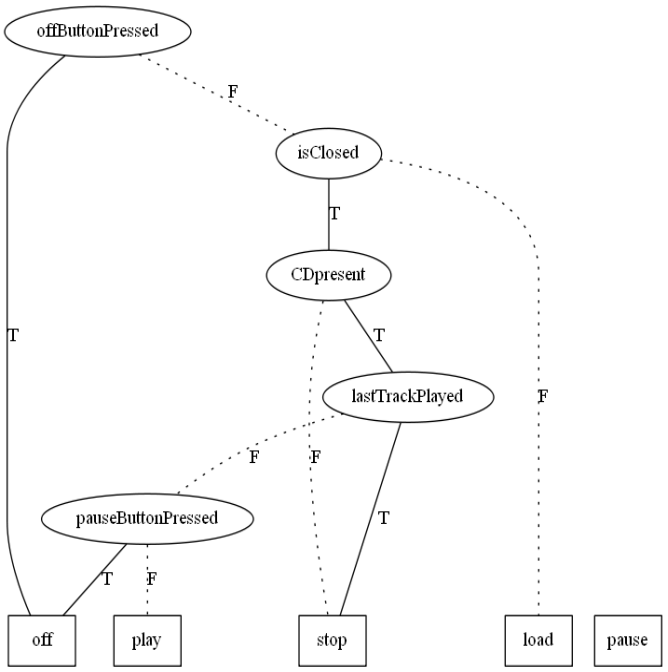

Figure 5.12. "play" MTBDD after the application of the *edgeC*.

The *edgeC* redirects the edge from *pause* to *off* and the resulting DT loses the *pause* action and the edge from event *play* to *pause* also is removed. The mutant on Fig. 5.13 is distinguished from the original model by CESs *[, play, play, pause, play, off,]* and *[, play, stop, play, load, play, pause, off,]*. FCES could not eliminate the mutated model on the Fig. 5.13, because the event sequences do not match.

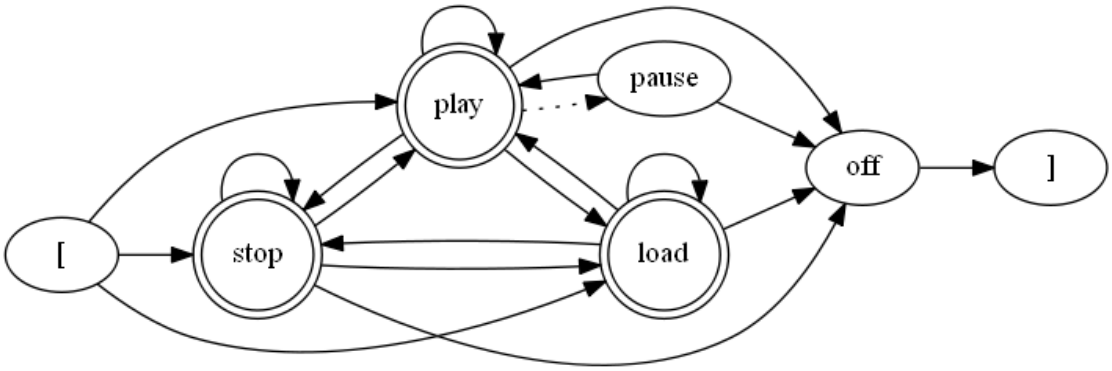

Figure 5.13. “CD Player” ESG-DT after the *edgeC* usage.

The MTBDD on Fig. 5.14 is the result of tnC application on Fig 5.2:

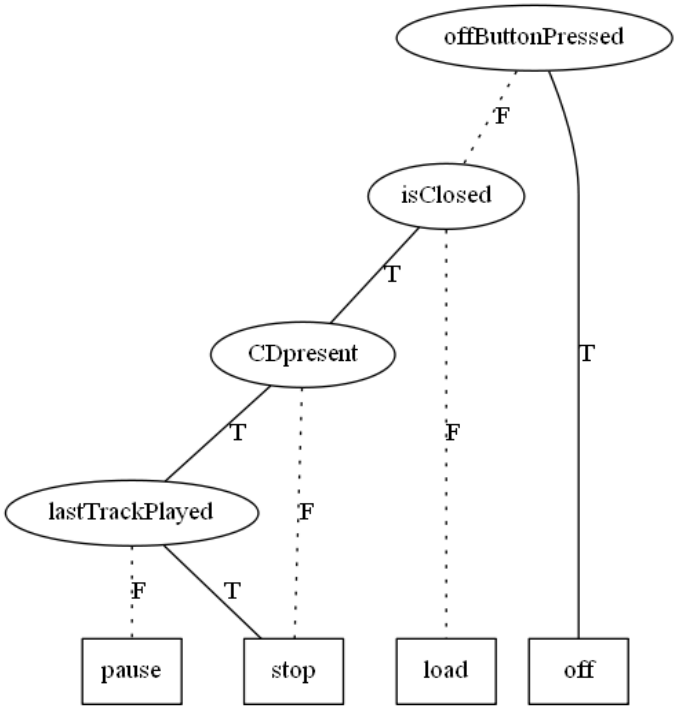

Figure 5.14. “stop” MTBDD after the application of *tnC*.

Table 5.12. “stop” DT the application of *tnC* on the “stop” MTBDD.

stop		Rules				
		R0	R1	R2	R3	R4
Conditions	offButtonPressed	T	F	F	F	F
	isClosed	-	F	T	T	T
	CDpresent	-	-	F	T	T
	lastTrackPlayed	-	-	-	F	T
Actions	pause				X	
	stop			X		X
	load		X			
	off	X				

The resulting ESG-DT on Fig.5.15, made from usage of *tnC* operator on Fig. 5.2 MTBDD, acquired a new edge *stop -> pause* (dashed arrow) and lost *stop -> play*. For Fig. 5.15, only the CES *[, play, stop, play, load, play, pause, off,]* kills mutated ESG-DT. In case of FCES set, only *[, stop, pause,* FCES test sequence is covered.

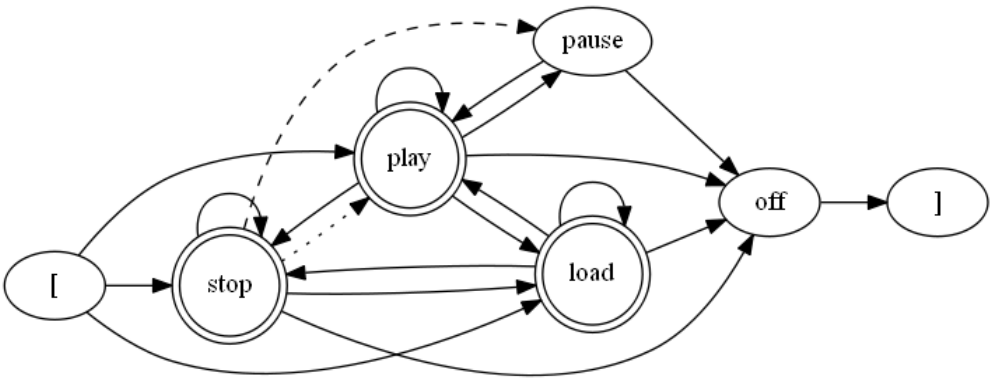

Figure 5.15. “CD Player” ESG-DT after the *tnC* usage.

The *edgeS* operator influence on Fig. 5.4 MTBDD is depicted on Fig. 5.17, which shows inversed outgoing edges going from *lastTrackPlayed* non-terminal node to *stop* and *pauseButtonPressed* nodes:

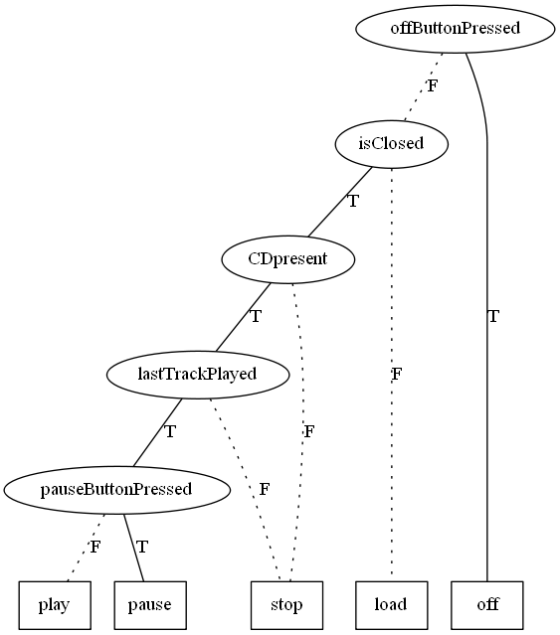

Figure 5.16. “play” MTBDD after the application of edgeS.

Table 5.13. “play” DT the application of *edgeS* on the “play” MTBDD.

play		Rules					
		R0	R1	R2	R3	R4	R5
Conditions	offButton Pressed	T	F	F	F	F	F
	isClosed	-	F	T	T	T	T
	CDpresent	-	-	F	T	T	T
	lastTrack Played	-	-	-	T	T	F
	pauseButton Pressed	-	-	-	F	T	-
Actions	play				X		
	pause					X	
	stop			X			X
	load		X				
	off	X					

The resulting ESG in Fig. 5.17, generated by using *edgeS* mutation operator, has not changed at all. This mutant's graph is equivalent to the original graph.

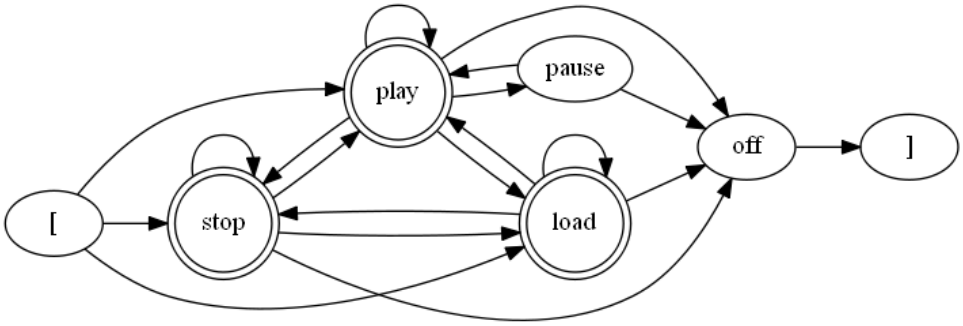

Figure 5.17. "CD Player" after the *edgeS* usage.

By applying *tnO*, *tnC*, *edgeO*, *edgeC* and *edgeS* MTBDD mutation operators on MTBDDs obtained from DT in Fig.5.1. ESG, we generated 46 mutated ESG-DTs. The Table 5.14 provides the number of ESG-DT mutants per mutation operator.

Table 5.14. The number of mutants per operator for "CDplayer" ESG-DT.

edgeC	edgeI	edgeO	edgeS	tnC	tnI	tnO
6	NA	16	3	8	NA	13

According to the Algorithm 1, we must test these mutants via test sequences generated from the original model, represented in Tables 5.4 and 5.5. A mutant is distinguished when at least one CES can distinguish a mutant iff there is no sequence in mutant that can match a sequence of CES. In case of FCES, a mutant is distinguished when at least one FCES matches the mutant, because FCES is a faulty model of the original ESG-DT.

The Table 5.15 shows the number of mutants per operator, for each original MTBDD:

Table 5.15. The number of mutants of all original MTBDDs.

ESG-DT mutants	edgeC	edgeI	edgeO	edgeS	tnC	tnI	tnO	Total
play	2	NA	6	1	NA	NA	5	14
stop	2	NA	5	1	4	NA	4	16
load	2	NA	5	1	4	NA	4	16

In total, 16 ESG-DT mutants were obtained after mutating the *play* MTBDD by the listed operators. For the mutation of the *play* MTBDD on Fig. 5.4 only *edgeC*, *edgeO*, *edgeS* and *tnO* operators are used. The application of the *edgeI* and *tnI* operators is impossible because all non-terminal nodes already have both descendant nodes. The application of *tnC* is not reasonable in the event *play*, since it already has outgoing edges with all existing nodes. Four out of six ESG-DT mutants derived from *edgeO* are detected by the CESs in the first and fourth rows in Table 5.4. However, the FCESs are not able to distinguish any mutants. CES in the first row of Table 5.4 distinguishes only one *edgeC* derived mutant, when the other mutants pass all CESs. But both mutants fail all FCESs, therefore are not distinguished. Same first and fourth CESs detects *tnO* derived mutants and none of the FCESs can distinguish them. In case of *edgeS* obtained ESG-DT mutant test sequences in neither CES and nor FCES test sets can see mutant in it.

In total, 16 ESG-DT mutants were obtained after mutating the *stop* MTBDD by the listed operators. The *stop* MTBDD in Fig. 5.2 is mutated by *edgeC*, *edgeO*, *edgeS, tnC* and *tnO*. The *edgeI* and *tnI* operator usage is unreasonable, as all non-terminal nodes already have both descendant nodes. The mutants generated by *tnC* were detected by test sequences from both CESs and FCESs, where the test sequences of the second and fourth rows in Table 5.4. and the first FCES in Table 5.5 were able to kill them. However, *edgeS* generated an undetectable mutant. The fourth CES could detect half of the mutants generated by edgeC, another half was not distinguished and none of the FCESs could say that they are mutants. Three out five of *edgeO* and all *tnO* generated mutants were detected by the second and the fourth CES test sequences, but FCES test set was not able to distinguish mutants in them.

Totally 16 ESG-DT mutated models are generated by mutating “load” MTBDD. ESG-DT mutants are obtained from mutation of the “load” MTBDD (Fig. 5.3) by involving *edgeC*, *edgeO*, *edgeS*, *tnC* and *tnO* operators. Simple insertions operators are

not used, because all non-terminal nodes already have both descendants, therefore insertion of the new edge or terminal node becomes impossible. Mutated ESG-DT obtained with *tnC* operator are detected by both CESs in the second, third and fourth rows of Table 5.4 and the fifth FCES. The mutants obtained by the rest of the operators are not distinguished with FCES test set. Test cases two, three and four in Table 5.4 can distinguish faulty models constructed with *tnO* operator. CESs three and four could detect three out of five *edgeO* made mutants, whereas only the fourth CES could find half of mutants created with *edgeC* operator. The *edgeS* made faulty ESG-DT went unnoticed.

The results are presented in Table 5.16:

Table 5.16. Test results

Operator	CES	FCES
tnO	13 mutants are distinguished	No mutant is distinguished
tnC	8 mutants are distinguished	8 mutants are distinguished
edgeO	10 mutants are distinguished	No mutant is distinguished
edgeC	3 mutants are distinguished	No mutant is distinguished
edgeS	No mutant is distinguished	No mutant is distinguished

According to the results provided in Table 5.16, the mutants obtained after *tnC* usage are distinguished by both CESs and FCESs from the original model. On the contrary, the graphs of the *edgeS* derived mutants are completely identical to the graphs of the original model, as the result are not distinguished from behavior of the original ESG. Mutants derived from *tnO*, *edgeO* and *edgeC* are detected only by CESs.

Every mutant is distinguished only once by a certain CES. It means that mutants are not distinguished by two and more different FCESs Table 5.18 shows the number of detected mutants per mutation operator. It also shows the number of undetected mutants by certain FCES and the number of undetected mutants per mutation operator.

Table 5.17. CES detected number of mutants per mutation operator.

CES ID	CES	tnO	tnC	edgeO	edgeC	edgeS	Number of undetected mutants
1	[, play, play, pause, play, off,]	2	0	2	1	0	41
2	[, stop, load, stop, stop, off,]	4	4	2	0	0	36
3	[, load, load, off,]	2	2	2	0	0	40
4	[, play, stop, play, load, play, pause, off,]	5	2	4	2	0	33
-	Undetected mutant number per operator	0	0	6	3	3	-

Every mutant is distinguished only once by certain FCES. It means that mutants are not distinguished by two and more different FCESs. Table 5.18 shows the number of detected mutants per mutation operator. It also shows the number of undetected mutants by certain FCES and the number of undetected mutants per mutation operator.

Table 5.18. FCES detected number of mutants per mutation operator.

FCES ID	FCES	tnO	tnC	edgeO	edgeC	edgeS	Number of undetected mutants
1	[, stop, pause,	0	4	0	0	0	42
5	[, load, pause,	0	4	0	0	0	42
-	Undetected mutant number per operator	13	0	16	6	3	-

The mutation score is calculated by the following formula:

$$Score = \frac{Number\ of\ distinguished\ mutants}{Total\ number\ of\ mutants} * 100 \quad (5.1)$$

Mutation score of all CESs and FCESs, that distinguished at least one mutant is presented on Table 5.19.

Table 5.19. CD Player ESG-DT test sequences mutation score.

CES ID	Score	FCES ID	Score
1	≈10,87 %	1	≈ 9.52 %
2	≈21,74 %	5	≈ 9.52 %
3	≈13,04 %	-	-
4	≈28,26 %	-	-

5.2. Cruise Control

Cruise Control's specification is taken from [17]. Originally it is represented as transition table, which in turn is transformed into DT augmented ESG. Each mode is represented by events in ESG-DT and all inputs are used in corresponding DE in DTs. In total Cruise Control ESG-DT consists of four *off*, *inactive*, *cruise*, *override*.

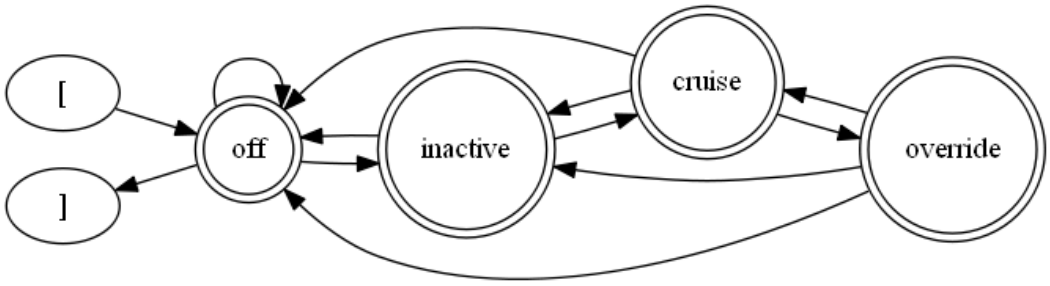

Figure 5.18. Cruise Control ESG-DT.

For Cruise Control ESG-DT, one CES (Table 5.20) and nine FCESs (Table 5.21) are generated, consisting of 18 and 22 events, respectively.

Table 5.20. CES of the Cruise Control ESG-DT.

CES	[, off, off, inactive, off, inactive, cruise, inactive, cruise, off, inactive, cruise, override, cruise, override, inactive, cruise, override, off,]

Table 5.21. FCESs of the Cruise Control ESG-DT.

2	[, off, cruise,
2	[, off, override,
3	[, off, inactive, inactive,
3	[, off, inactive, override,
4	[, off, inactive, cruise, cruise,
5	[, off, inactive, cruise, override, override,
1	[, inactive,
1	[, cruise,
1	[, override,

Table 5.22. The original "off" DT.

off		Rules	
		R0	R1
Conditions	Ignited	T	F
Actions	inactive	X	
	off		X

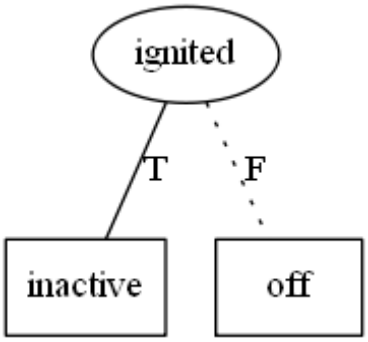

Figure 5.19. MTBDD of the original "off" DT.

For “off” MTBDD on Fig.5.19 *edgeO*, *edgeS*, *tnC* and *tnO* mutation operators are used. As it has only one non-terminal node with two outgoing edges the usage of *edgeI* and *tnI* operators is impossible and using *edgeC* also becomes pointless, because same child node for one non-terminal operator is considered as a redundant test. In total, seven ESG-DT mutants are derived by mutating “off” MTBDD. The Table 5.23 shows the number of mutants per operator:

Table 5.23. The number of Cruise Control mutants of all original MTBDDs.

MTBDD	edgeC	edgeI	edgeO	edgeS	tnC	tnI	tnO
off	NA	NA	2	1	2	NA	2
inactive	1	1	2	1	4	2	2
cruise	3	3	3	NA	3	1	3
override	3	1	3	1	3	1	3

The ESG-DT mutants derived from *edgeO*, are distinguished by CES, but FCESs are not able to distinguishe them. *edgeS* derived mutant is not distinguished by neither CES nor FCESs. The ESG-DTs derived by *tnC* are distinguished by failing CES and passing FCESs in the first and second rows in Table 5.21. Finally, testing of *tnO* derived mutant shows same result as for *edgeO*.

Table 5.24. The original “inactive” DT.

inactive		Rules	
		R0	R1
Conditins	Ignited	F	T
	EngRun	-	F
Actions	off	X	
	cruise		X

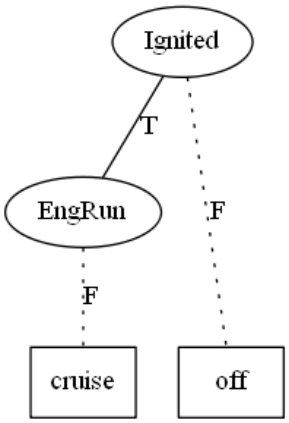

Figure 5.20. MTBDD of the original “inactive” DT.

The “inactive” MTBDD on Fig.5.20 is mutated by means of all presented operators. In total, 13 ESG-DT mutants are derived by mutating “inactive” MTBDD. The ESG-DT mutants derived by using *edgeC*, *edgeO*, and *tnO* are distinguished by CES by not passing test, meanwhile none of the FCESs are able to match them. On the contrary, mutants made by *tnI* are not distinguished by CES, but they are distinguished by the FCESs in the third and fourth rows of Table 5.21. The mutants generated by *edgeI* and *edgeS* operators are not distinguishable by both CES and FCES test set. Finally, *tnC* operator generates mutants which are distinguished by both CES and FCESs in the third and fourth rows of Table 5.21.

Table 5.25. The original “cruise” DT.

cruise		Rules		
		R0	R1	R2
Conditions	Ignited	F	T	T
	EngRun	-	F	T
	TooFast	-	T	F
	Brake	-	-	T
	Deactivate	-	-	T
Actions	off	X		
	inactive		X	
	override			X

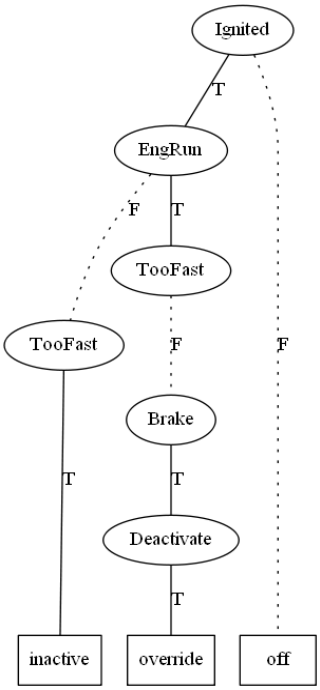

Figure 5.21. MTBDD of the original “cruise” DT.

Mutation of “cruise” MTBDD is done by all introduced MTBDD operators, except *edgeS*. The reason is the single outgoing edges connecting terminal nodes with corresponding non-terminal nodes. In total, 16 ESG-DT mutants are derived by mutating “cruise” MTBDD. Mutants obtained by means of *edgeC*, *edgeO* and *tnO* are detected similarly, with sole CES, by not passing it and are not distinguished at all by any test in a FCES set. The behavior of *tnI* derived mutant shows opposite result: despite of model being changed, the CES cannot distinguish the mutant, but FCES in the fifth row of the Table 5.21 is able to distinguish it. The *tnC*-based mutant is distinguished by both CES and FCES in fifth row of the Table 5.21. In the end, *edgeI* created mutant is not distinguished by any test sequence.

Table 5.26. The original “override” DT.

override		Rules		
		R0	R1	R2
Conditions	Ignited	F	T	T
	EngRun	-	F	T
	TooFast	-	-	F
	Brake	-	-	F

(cont. on next page)

Table 5.26. (cont.)

	Activate	-	-	T
	Resume	-	-	T
Actions	off	X		
	inactive		X	
	cruise			X

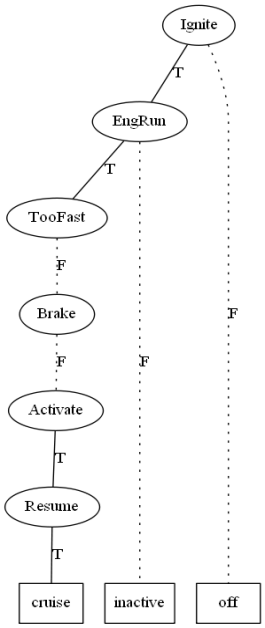

Figure 5.22. MTBDD of the original “override” DT.

For “override” MTBDD mutation all mutation operators are involved. In total, 15 ESG-DT mutants are derived by mutating “override” MTBDD. The testing of the *edgeC*, *edgeO* and *tnO* derived ESG-DT mutants, show same outcome, because they fail CES and therefore are distinguished, but FCES test set is unable to detect them. The *tnI* derived mutant is distinguished by the sixth FCES, but not distinguished by CES. However, *tnC*-based mutant is distinguished by both CES and FCES located in the sixth row of the Table 5.21. The mutants generated after *edgeI* and *edgeS* application on “override” are not distinguished by CES and FCES test sequences, therefore these mutants are living ones.

Each mutant is distinguished only once by the certain CES. It means that mutants are not distinguished by two and more different CESs Table 5.27 shows the number of distinguished mutants per mutation operator. It also shows the number of

undetected mutants by the CES and the number of undetected mutants per mutation operator.

Table 5.27. CES detected number of mutants per mutation operator.

CES ID	CES	tnO	tnI	tnC	edgeI	edgeO	edgeC	edgeS	Number of undetected mutants
1	[, off, off, inactive, off, inactive, cruise, inactive, cruise, off, inactive, cruise, override, cruise, override, inactive, cruise, override, off,]	10	0	12	0	10	7	0	12
-	Undetected mutant number per operator	0	4	0	7	0	0	3	-

Each FCES sequence can detect only one mutant. It means that mutants are not distinguished by two and more different FCESs Table 5.28 shows the number of detected mutants per mutation operator. It also shows the number of undetected mutants by certain FCES and the number of undetected mutants per mutation operator.

Table 5.28. FCES detected number of mutants per mutation operator.

FCES ID	FCES	tnO	tnI	tnC	edgeI	edgeO	edgeC	edgeS	Number of undetected mutants
1	[, off, cruise,	0	0	1	0	0	0	0	50
2	[, off, override,	0	0	1	0	0	0	0	50
3	[, off, inactive, inactive,	0	1	2	0	0	0	0	48
4	[, off, inactive, override,	0	1	2	0	0	0	0	48
5	[, off, inactive, cruise, cruise,	0	1	3	0	0	0	0	47
6	[, off, inactive, cruise, override, override,	0	1	3	0	0	0	0	47
-	Undetected mutant number per operator	10	0	0	5	10	7	3	-

The mutation score is calculated by the following formula:

$$Score = \frac{Number\ of\ distinguished\ mutants}{Total\ number\ of\ mutants} * 100 \quad (5.2)$$

Table 5.29. Cruise Control ESG-DT test sequences mutation score.

CES ID	Score	FCES ID	Score
1	≈ 76,47 %	1	≈ 1.96 %
-	-	2	≈ 1.96 %
-	-	3	≈ 5.88 %
-	-	4	≈ 5.88 %
-	-	5	≈ 7.84 %
-	-	6	≈ 7.84 %

Table 5.29 presents a mutation score of each test sequence used for this case.

5.3. Simple Automated Teller Machine

The Simple Automated Teller Machine (SATM) is a simplified model of the real ATMs. The model used in thesis consists of eight events *Insert Card*, *Account*, *Deposit*, *Withdrawal*, *Insert Envelope*, *Cancel*, *Proceed*, *Withdraw Card*, where only two events *Insert Card*, *Withdrawal* hold DTs.

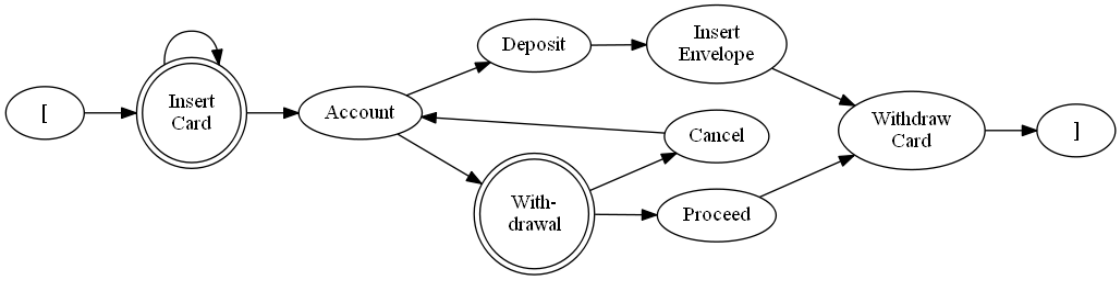

Figure 5.23. Simple ATM ESG-DT.

Table 5.30. CESs of the Simple ATM ESG-DT.

5 nodes CES	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card,],
9 nodes CES	[, Insert Card, Insert Card, Account, Withdrawal, Cancel, Account, Withdrawal, Proceed, Withdraw Card,]

Table 5.31. FCESs of the Simple ATM ESG-DT.

2	[, Insert Card, Withdrawal,
2	[, Insert Card, Insert Envelope,
2	[, Insert Card, Cancel,
2	[, Insert Card, Proceed,
2	[, Insert Card, Withdraw Card,
2	[, Insert Card, Deposit,
3	[, Insert Card, Account, Insert Card,

(cont. on next page)

Table 5.31. (cont.)

3	[, Insert Card, Account, Account,
3	[, Insert Card, Account, Insert Envelope,
3	[, Insert Card, Account, Cancel,
3	[, Insert Card, Account, Proceed,
3	[, Insert Card, Account, Withdraw Card,
4	[, Insert Card, Account, Withdrawal, Insert Card,
4	[, Insert Card, Account, Withdrawal, Account,
4	[, Insert Card, Account, Withdrawal, Withdrawal,
4	[, Insert Card, Account, Withdrawal, Insert Envelope,
4	[, Insert Card, Account, Withdrawal, Withdraw Card,
4	[, Insert Card, Account, Withdrawal, Deposit,
5	[, Insert Card, Account, Deposit, Insert Envelope, Insert Card,
5	[, Insert Card, Account, Deposit, Insert Envelope, Account,
5	[, Insert Card, Account, Deposit, Insert Envelope, Withdrawal,
5	[, Insert Card, Account, Deposit, Insert Envelope, Insert Envelope,
5	[, Insert Card, Account, Deposit, Insert Envelope, Cancel,
5	[, Insert Card, Account, Deposit, Insert Envelope, Proceed,
5	[, Insert Card, Account, Deposit, Insert Envelope, Deposit,
5	[, Insert Card, Account, Withdrawal, Cancel, Insert Card,
5	[, Insert Card, Account, Withdrawal, Cancel, Withdrawal,
5	[, Insert Card, Account, Withdrawal, Cancel, Insert Envelope,
5	[, Insert Card, Account, Withdrawal, Cancel, Cancel,
5	[, Insert Card, Account, Withdrawal, Cancel, Proceed,
5	[, Insert Card, Account, Withdrawal, Cancel, Withdraw Card,
5	[, Insert Card, Account, Withdrawal, Cancel, Deposit,
5	[, Insert Card, Account, Withdrawal, Proceed, Insert Card,
5	[, Insert Card, Account, Withdrawal, Proceed, Account,
5	[, Insert Card, Account, Withdrawal, Proceed, Withdrawal,
5	[, Insert Card, Account, Withdrawal, Proceed, Insert Envelope,
5	[, Insert Card, Account, Withdrawal, Proceed, Cancel,
5	[, Insert Card, Account, Withdrawal, Proceed, Proceed,
5	[, Insert Card, Account, Withdrawal, Proceed, Deposit,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Insert Card,

(cont. on next page)

Table 5.31. (cont.)

6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Account,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Withdrawal,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Insert Envelope,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Cancel,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Proceed,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Withdraw Card,
6	[, Insert Card, Account, Deposit, Insert Envelope, Withdraw Card, Deposit,
4	[, Insert Card, Account, Deposit, Insert Card,
4	[, Insert Card, Account, Deposit, Account,
4	[, Insert Card, Account, Deposit, Withdrawal,
4	[, Insert Card, Account, Deposit, Cancel,
4	[, Insert Card, Account, Deposit, Proceed,
4	[, Insert Card, Account, Deposit, Withdraw Card,
4	[, Insert Card, Account, Deposit, Deposit,
1	[, Account,
1	[, Withdrawal,
1	[, Insert Envelope,
1	[, Cancel,
1	[, Proceed,
1	[, Withdraw Card,
1	[, Deposit,

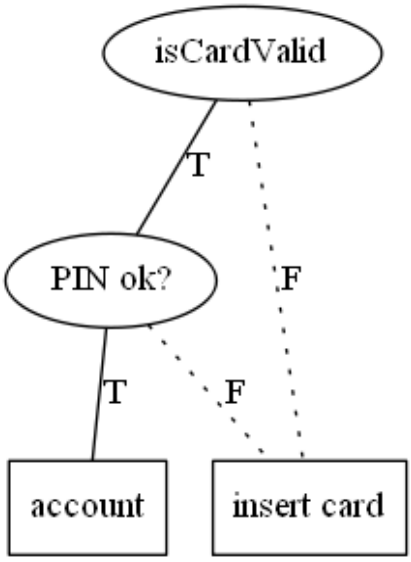

Figure 5.24. MTBDD of the original “insert card” DT.

Table 5.32. The original “insert card” DT.

insert card		Rules		
		R0	R1	R2
Conditions	isCardValid?	F	T	T
	PIN ok?	-	F	T
Actions	insert card	X	X	
	account			X

The mutation operators *edgeO*, *edgeS*, *tnC* and *tnO* are involved in “insert card” MTBDD, depicted in Fig. 5.24, mutation. *edgeI* and *tnI* are used because non-terminal nodes have both low and high child nodes.

According to the Table 5.33, 16 ESG-DT mutants are derived by mutating “insert card” MTBDD. All mutants obtained after the application of *tnC* operator are detected by both CES and FCESs test sets. Half of the mutants are detected by all CESs and another half by only the second CES in Table 5.30. FCESs which are successfully passed by 12 mutants are in the first, second, third, fourth, fifth and sixth rows of the Table 5.31. On the other hand, neither of CESs and FCESs can find *edgeS* derived mutant, because it passes all CESs and fails all FCESs as the original model. The whole CES test set can detect the *edgeO* derived mutant, but not FCES. Finally, the first

mutant obtained by *tnO* operator successfully fails all CESs, whereas the second mutant is detected only by the CES in the second row of Table 5.30. Both mutants are not distinguished by FCES test set.

Table 5.33. The number of Simple ATM mutants of all original MTBDDs.

MTBDD	edgeC	edgeI	edgeO	edgeS	tnC	tnI	tnO
insert card	NA	NA	1	1	12	NA	2
withdrawal	2	2	2	NA	12	6	2

For "withdrawal" MTBDD mutation (Fig. 5.25) are involved all introduced operators except *edgeS*. In accordance with Table 5.32, 26 faulty ESG-DTs are generated by mutating "withdrawal" contract.

Table 5.34. The original "withdrawal" DT.

withdrawal		Rules		
		R0	R1	R2
Conditions	amount <= balance	F	T	T
	buttonProceedPressed	-	T	F
	buttonCancelPressed	-	F	T
Actions	cancel	X		X
	proceed		X	

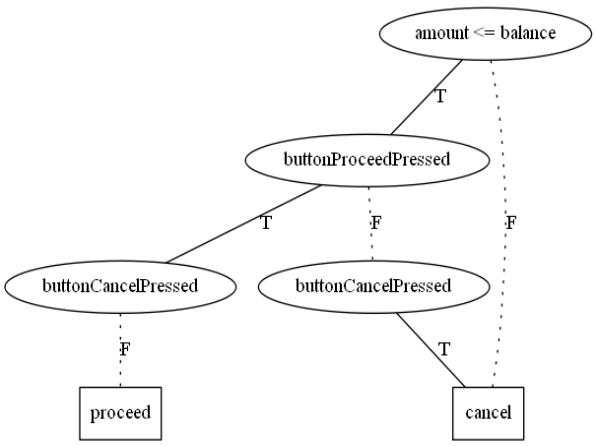

Figure 5.25. MTBDD of the original "withdrawal" DT.

Unlike the results provided for testing of *tnC* derived mutants for “insert card” MTBDD, mutants generated with the same operator for “withdrawal” are distinhuised only by the second CES in Table 5.30. For FCESs, among of 61 test sequences, the 13th, 14th, 15th, 16th, 17th, and 18th test sequences are passed successfully, therefore these faulty models are distinguished from the original model. The *tnI* derived ESG-DTs, are detected by the same FCESs used in *tnC*. However, the CESs are not able to distinguish it from the original model. *edgeC* and *edgeO* show same results, where their mutants fail all FCESs, like the original model. In case, of testing with CES half of their mutants are distinguished by all tests in a set and another half by only with the second test sequence. Whereas *tnO* operator mutants are distinguished by only with the second CES (Table 5.30). The notorious testing results are obtained after testing of *edgeI*-based mutants, where all CESs are passed and FCESs are failed.

Every mutant is distinguished by a unique CES. It means that mutants are not distinguished by two and more different CESs Table 5.35 shows the number of detected mutants per mutation operator. It also shows the number of undetected mutants by certain CES and the number of undetected mutants per mutation operator.

Table 5.35. CES detected number of mutants per mutation operator.

CES ID	tnO	tnI	tnC	edgeI	edgeO	edgeC	edgeS	Number of undetected mutants
1	0	0	0	0	0	0	0	42
2	3	0	18	0	1	1	0	19
-	1	6	6	2	2	1	1	-

Each mutant is distinguished by one unique FCES. It means that any mutant is not distinguished by two and more different FCESs. Table 5.36 shows the number of detected mutants per mutation operator. It also shows the number of undetected mutants by certain FCES and the number of undetected mutants per mutation operator.

Table 5.36. FCES detected number of mutants per mutation operator.

FCES ID	tnO	tnI	tnC	edgeI	edgeO	edgeC	edgeS	Number of undetected mutants
1	0	0	2	0	0	0	0	40
2	0	0	2	0	0	0	0	40
3	0	0	2	0	0	0	0	40
4	0	0	2	0	0	0	0	40
5	0	0	2	0	0	0	0	40
6	0	0	2	0	0	0	0	40
13	0	1	2	0	0	0	0	39
14	0	1	2	0	0	0	0	39
15	0	1	2	0	0	0	0	39
16	0	1	2	0	0	0	0	39
17	0	1	2	0	0	0	0	39
18	0	1	2	0	0	0	0	39
-	4	0	0	2	3	2	1	-

The mutation score of each CES and FCES is presented on Table 5.37 and calculated by the following formula:

$$Score = \frac{Number\ of\ distinguished\ mutants}{Total\ number\ of\ mutants} * 100 \quad (5.3)$$

Table 5.37. Simple ATM ESG-DT test sequences mutation score

CES ID	Score	FCES ID	Score
1	≈ 0 %	1	≈ 4.76 %
2	≈ 54,76 %	2	≈4.76 %
-	-	3	≈4.76 %
-	-	4	≈4.76 %
-	-	5	≈4.76 %
-	-	6	≈4.76 %
-	-	13	≈ 7,14 %
-	-	14	≈ 7,14 %

(cont. on next page)

Table 5.37. (cont.)

CES ID	Score	FCES ID	Score
-	-	15	≈ 7,14 %
-	-	16	≈ 7,14 %
-	-	17	≈ 7,14 %
-	-	18	≈ 7,14 %

5.4. Discussion

The mutation analysis of three cases indicated that some operators produce undetectable mutants, so that neither CES nor FCES test sets can distinguish them. Some operators produce a set of mutants of which detection depends on certain conditions. The reason of this phenomenon lies in the number of connections between terminal and non-terminal nodes of MTBDD.

The existence of an edge in ESG-DT from node A to node B depends on the existence of at least one rule containing the corresponding action B in the event A's DT. The DT rules are represented as paths to the terminal node in MTBDD. The terminal node can have at least one incoming edge to be considered a part of MTBDD. If there is only one incoming edge, then omission of that edge leads to the detaching of the corresponding terminal node and the DT after conversion will have one less rule and absent action. The final ESG-DT will not have the corresponding edge. Otherwise, when the number of incoming edges is minimum two, then the omission of one of the selected edges will reduce the number of rules by one and remain the corresponding action in DT. The final ESG-DT graph will preserve all its edges. So, whenever the *edgeC* redirects an edge or *edgeO* omits the edge in MTBDD, preservation of the terminal node will depend only on the number of its incoming edges.

The insertion and swapping of the edges in MTBDD results in the increase of the number of rules in DT and the exchange of the triggered actions of two rules in DT, respectively. The application of *edgeI* and *edgeS* operators changes the contracts in data events, i.e., events with DT, but does not change the edge number or direction of ESG-DT graph.

The operations on terminal nodes have inevitable impact on the resulting ESG-DT graph. The omission of the terminal node in MTBDD results in the omission of the

corresponding action from DT and all related rules. Therefore, the edge going from the respective data event to the event representing the omitted action will be removed. The *tnO* operator is responsible for this change. Insertion of the terminal node attaches a new terminal node to MTBDD and adds a new action with corresponding rule in DT. The final ESG-DT will acquire a new edge going from the current event to another event, represented as a new terminal node. This change is performed by *tnI* operator. Finally, the substitution of the corrupted terminal node replaces the chosen terminal node with a new one, by preserving all incoming edges. The event DT will have an updated action. The ESG-DT will redirect the edge from the "corrupted" event to the event, represented by the new terminal node. The *tnC* operator is responsible for edge redirection in ESG-DT. Similar results are already shown in [25] by for *sO*, *sI* and *sC* operators.

The detection of mutants by CES and FCES depends on the connections between events in mutant ESG-DTs. According to the results presented by three cases the commonality between mutants detected by CES is the absence of the certain edge. A FCES, on the contrary, distinguishes a mutant only if there is a certain edge connecting two events, which is introduced in FCES generation.

Based on the observations on impact of presented MTBDD operator on ESG-DT, omission, or redirection of the only incoming edge of the terminal node, omission, or replacement of the terminal node in MTBDD will generated mutant ESG-DT, which will be 100% distinguishable by the CES. This is achieved by *edgeO*, *edgeC* and *tnO* operators. On the other hand, insertion, or replacement of the terminal node in MTBDD will create 100% distinguishable by FCES mutant ESG-DT. Hence, only *tnI* and *tnC* operators can generate such mutants.

The remaining *edgeI* and *edgeS* operators will always generate the living mutants, as the resulting ESG-DT mutants are not distinguishable by both CES and FCES.

Therefore, it is reasonable to say, that there exists a hierarchy for presented mutants, where:

1) the operators, which generate mutants that are always detectable by both CES and FCES, are presented in category **I** in Table 5.38.
2) the operators, which generate mutants that are always detectable by either CES or FCES, are presented in category **II** in Table 5.38.

3) the operators, which generate mutants that are sometimes detectable by either CES or FCES, are presented in category **III** in Table 5.38.
4) the operators, which generate mutants that are never detectable by neither CES nor FCES, are presented in category **IV** in Table 5.38.

Table 5.38. The hierarchy of the MTBDD mutation operators.

Category	Mutation operators
I	*tnC*
II	*tnI*, *tnO*
III	*edgeO*, *edgeC*
IV	*edgeI*, *edgeS*

The drawback in the FCES test sequences for ESG-DT mutation analysis is that, those FCESs which second event (following pseudo-event *ſ*) which originally does not come after the starting *ſ* pseudo-event in ESG-DT, cannot distinguish mutants generated using the approach presented in thesis. As stated in the paragraph above, FCES can distinguish a mutant if it sees the new edge introduced in its own generation in the sequence of a mutant. However, mutation using MTBDD cannot deal with pseudo-events, because the ESG-DT does not know under which conditions or what kind of input or action can trigger the execution of the certain event in ESG-DT. This is explained in Chapter 3.2.1 in **Definition 3.4**, where the Condition set cannot be empty <=> $C \neq \emptyset$.

It follows that initial pseudo-event cannot be used for mutation using presented approach, since the corresponding MTBDD for it cannot be obtained, because the respective DT does not exist! Hence, FCES with second event which is not the starting event in ESG-DT will never distinguish a model-mutant from original model.

CHAPTER 6

TOOL SUPPORT

The approach proposed in this thesis is implemented on the pure Java without the usage of the side libraries or frameworks and the Test Suite Designer (TSD) is used for test sequence generation. This chapter describes the Java implementation and the application of test sequences on produced mutant set.

6.1. Java SE

As the implementation is built on Java programming language, the minimum requirement is the already installed Java 1.8.0 runtime environment or the newer version. The reason is that the software is written on the Java 8, because the implementation involves the usage of Java Stream API (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html).

6.2. IntelliJ Idea

IntelliJ Idea (https://www.jetbrains.com/idea) is the well-known powerful Integrated Development Environment (IDE) mainly used for Java development. For running the software, it is not important to use the specific version of this IDE, as it was updated multiple times. It is publicly available, free of charge, proprietary IDE and can be downloaded from https://www.jetbrains.com/idea/download.

6.3. Test Suite Designer

This is a non-commercial, free of charge tool designed for software analysis. TSD is based on event sequence graphs. The tool is accessible from http://download.ivknet.de. The purpose of the TSD usage is the generation of test sequences CES and FCES, because the ESG is the main area of interest. First of all, it is important to pass the ESG, simply by opening the pre-saved in filesystem ESG or by creating from scratch a new one. The tool has a GUI support (Fig. 6.1):

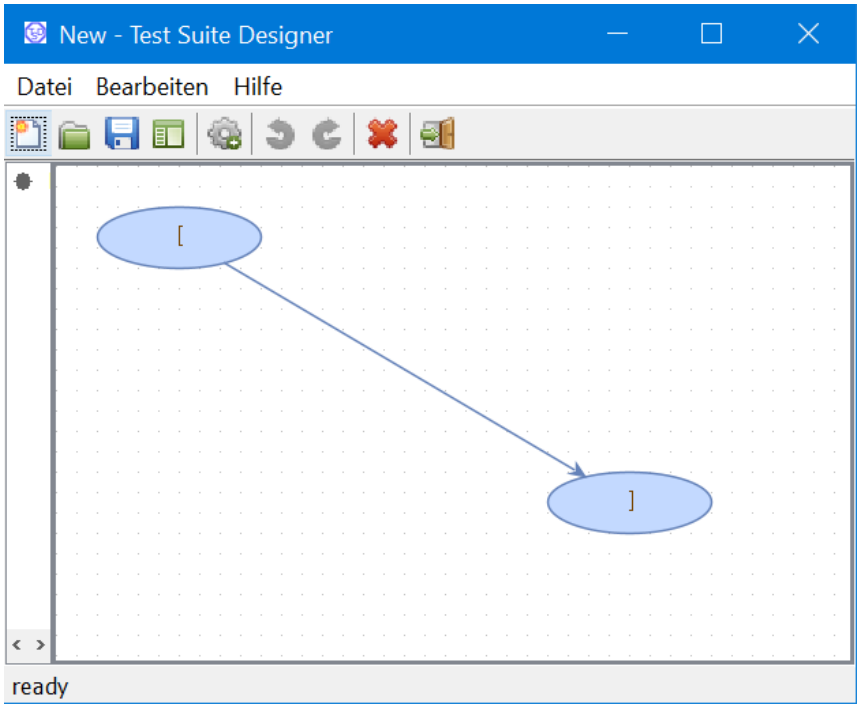

Figure 6.1. TSD initial window.

The example on Fig. 6.2 shows the Cruise Control ESG, which is presented in chapter 5.2, built in TSD tool.

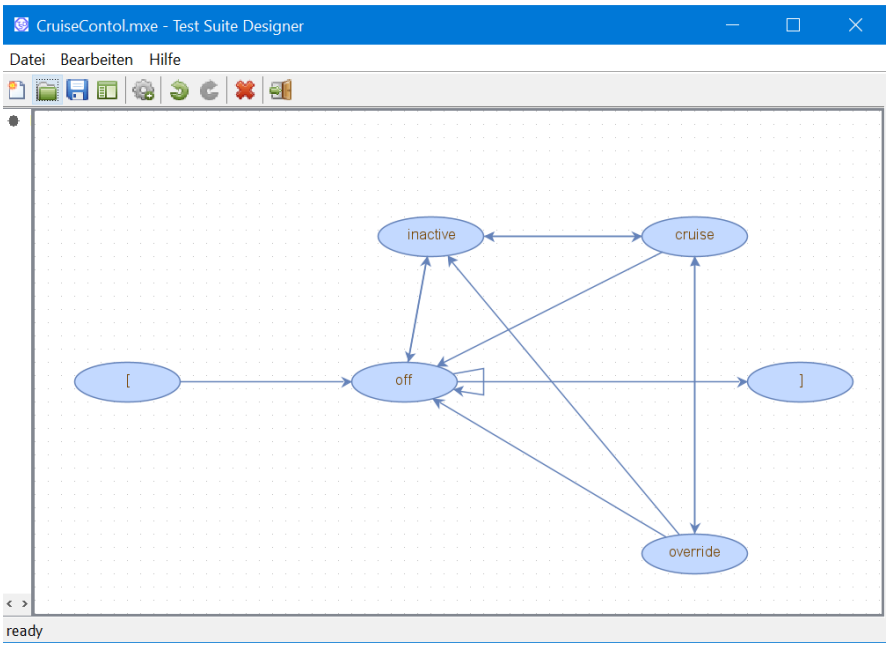

Figure 6.2. CruiseControl ESG.

Having the ready ESG, we can generate the test sequence by simply pressing the *Generiere Test-Skripte*, which will open the window on Fig. 6.3. The important part is leaving *positive Testfälle (CES) generieren* and *negative Testfälle (FCES) generieren* enabled.

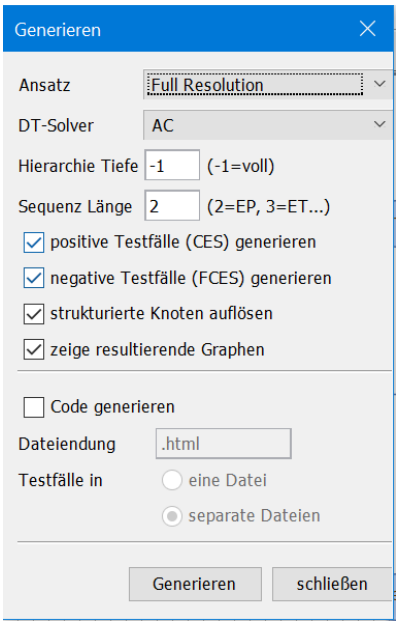

Figure 6.3. Test generation menu.

After clicking *Generieren* button in the bottom, the tool will provide the window (Fig. 6.4) containing all possible CESs and FCESs for the given ESG.

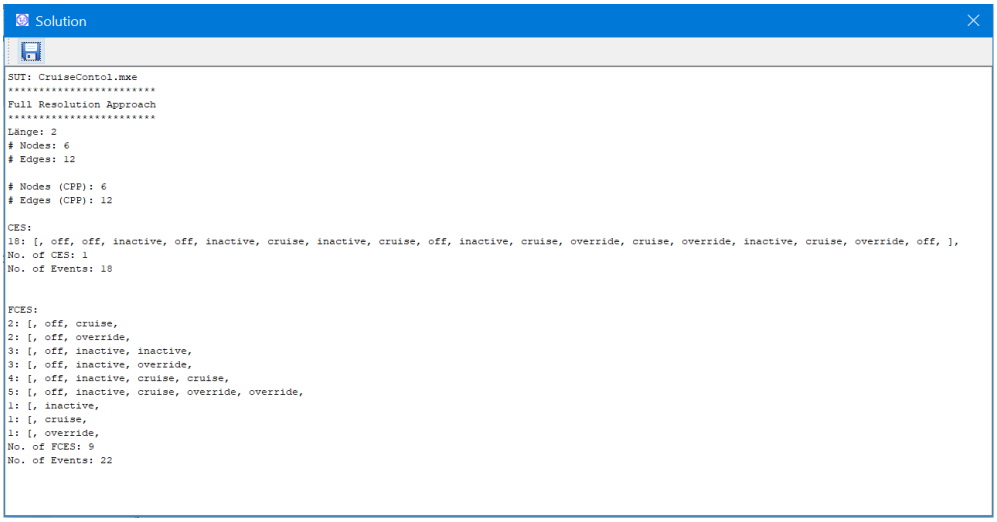

Figure 6.4. Generated CESs and FCESs.

6.4. Mutant Generator Software

This tool is written in Java programming language and represents a console application. The tool provides the facility for transformation of Decision Tables (DT) into Multi-Terminal Binary Decision Diagrams (MTBDD) and MTBDDs into DTs. Besides transformations, it also offers mutation of MTBDDs. Another feature proposed by this tool is the DT augmented Event Sequence Graph (ESG) processing. The work with ESG-DTs involves only the graph modification in accordance with a mutated contract in the corresponding event. The tool accepts DT in a text file format.

The following class diagrams represent the software architecture. Classes responsible for DT construction are *Action*, *Condition*, *Rule*, *Pair*, *DTFileReader* (Fig. 6.9), *Utility* (Fig 6.10), and *DecisionTable* (Fig. 6.11).

Classes Action.java (Fig. 6.5) and Condition.java (Fig. 6.6) represent the Actions and Conditions correspondingly in DT.

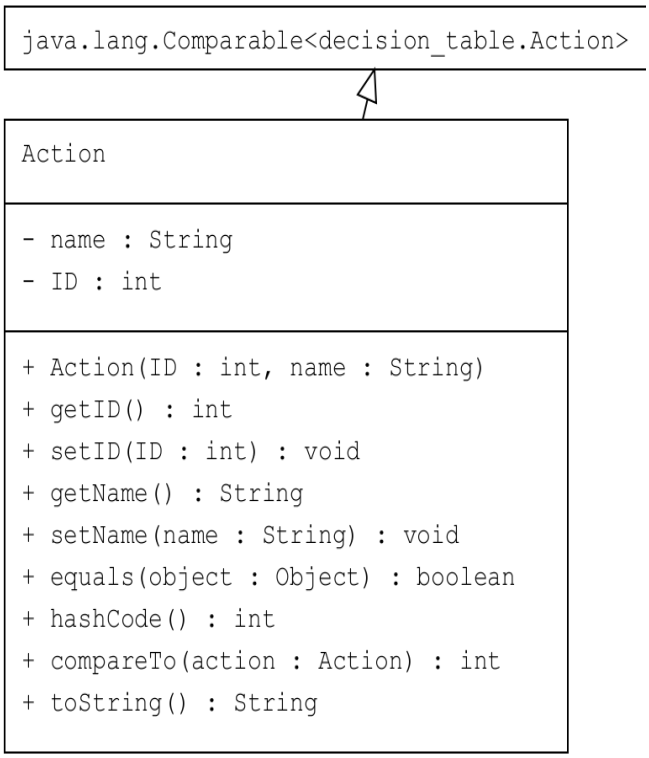

Figure 6.5. Action.java class diagram

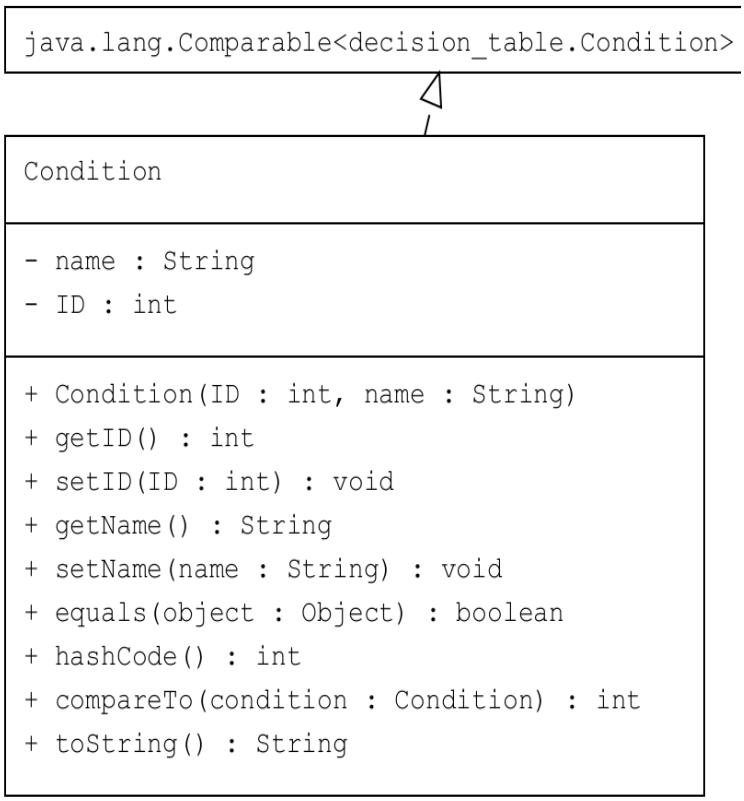

Figure 6.6. Condition.java class diagram.

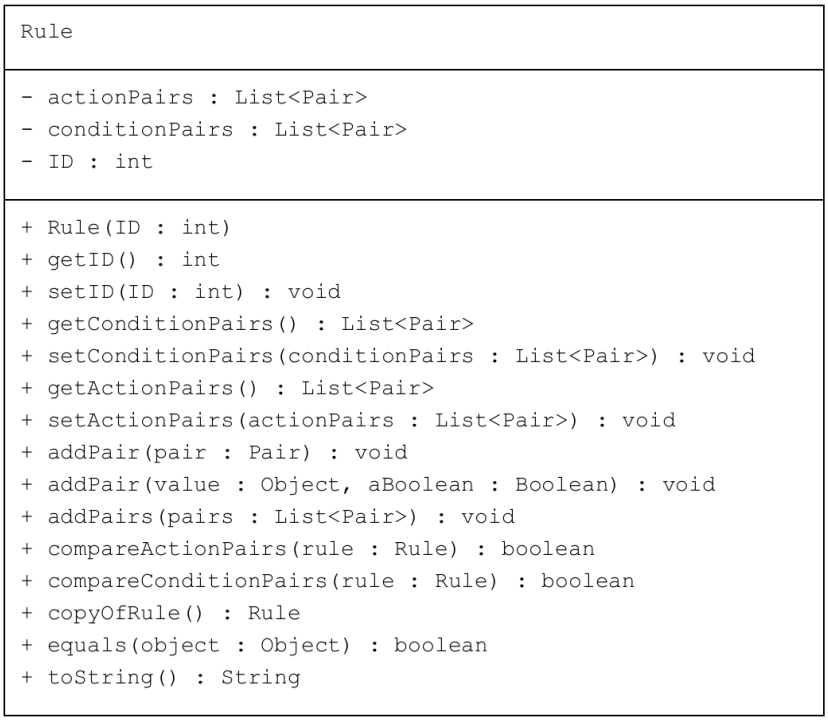

Figure 6.7. Rule.java class diagram.

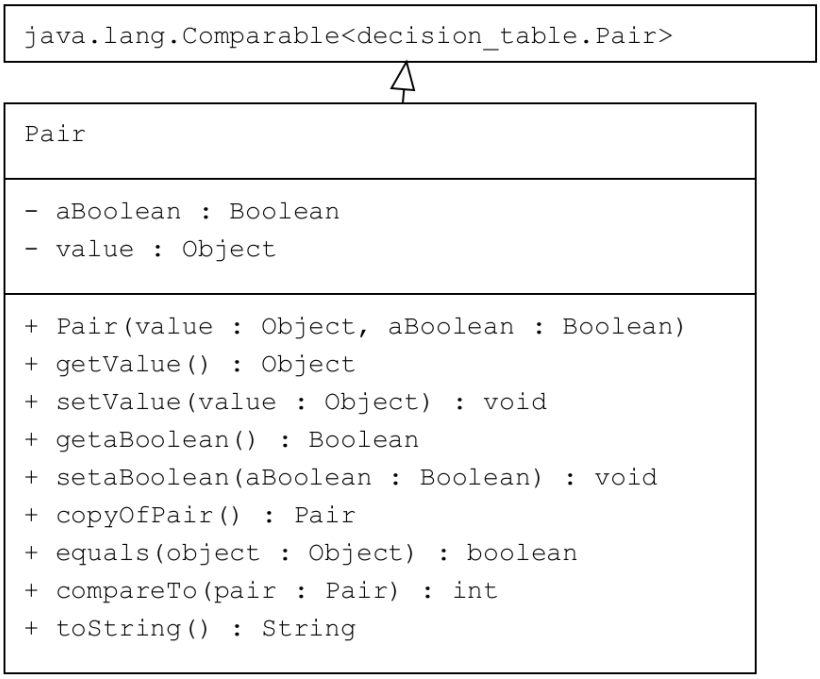

Figure 6.8. Pair.java class diagram.

Class Pair.java (Fig. 6.8) represents a pair of either Action and Boolean or Condition and Boolean. A Boolean is chosen as wrapper not primitive, in order to introduce ‘don’t care’ as null in Java. Also, it returns a copy of itself. Class Rule.java (Fig. 6.7) represents a container of Condition Pairs list and Action Pairs list.

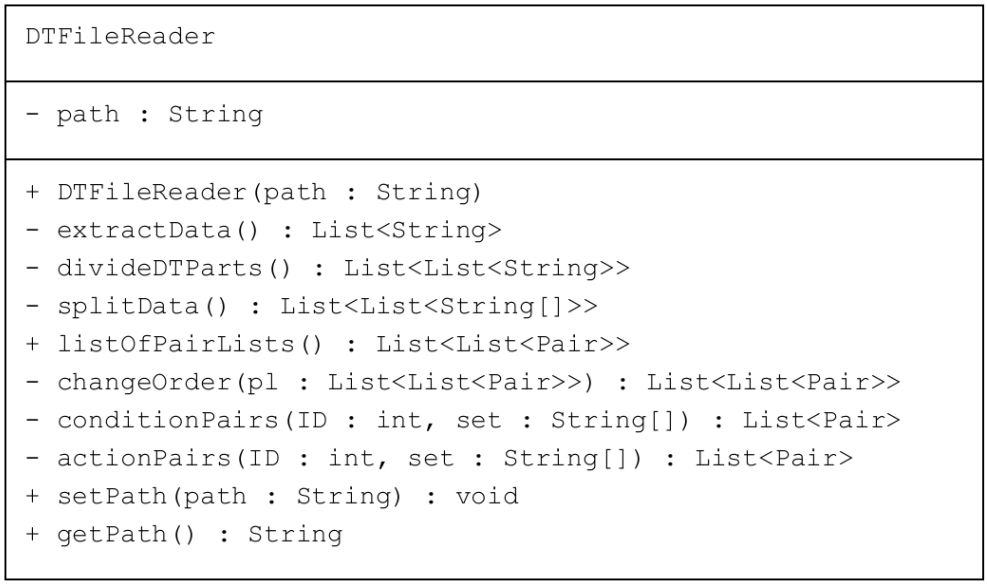

Figure 6.9. DTFileReader class diagram.

Class DTFileReader.java (Fig. 6.9) reads a DT from a text file. The method which prepares a conditions and actions for DT is *listOfPairLists()*.

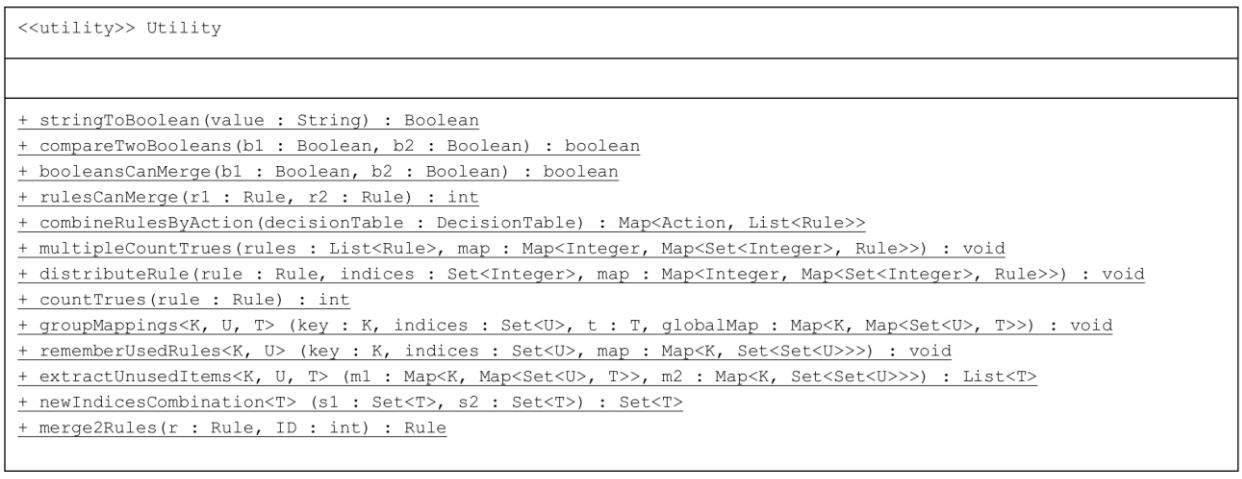

Figure 6.10. Utility.java class diagram.

The Utility.java (Fig. 6.10) class serves as a helper for the DecisionTable.java (6.11) class. The DecisionTable.java class represents a DT and provides such methods as *shortenDT()* which simplifies DT (merges Rules if possible), *expandDT()* shows DT with all of its Rules (the opposite of *shortenDT()*), creates a deep copy of DT.

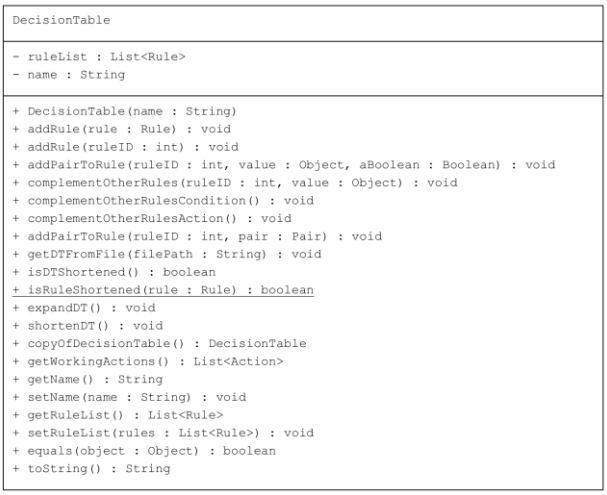

Figure 6.11. DecisionTable.java class diagram.

MTBDD is based on the following classes: *MTBDD*, *MTBDDBuilder*, *Node*. The relationship between them is represented on Figure 6.12.

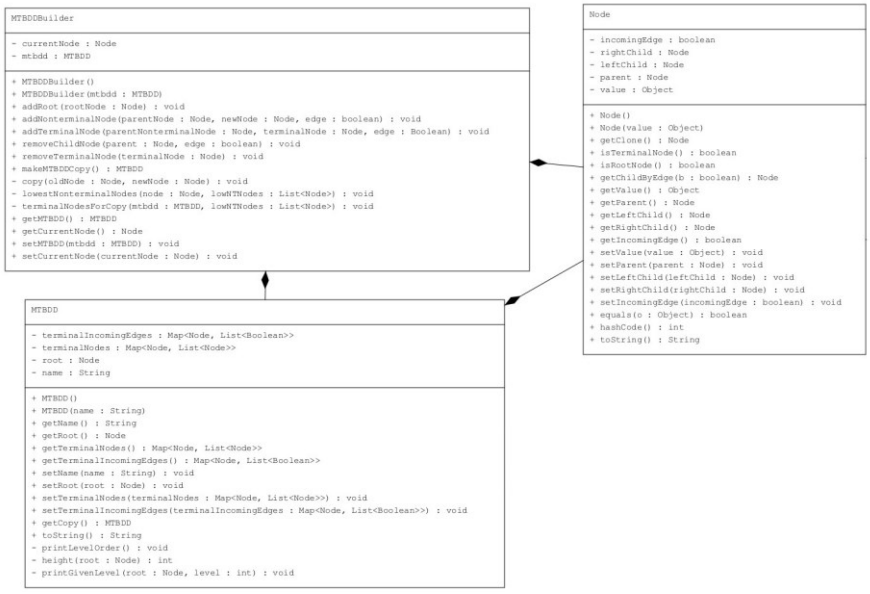

Figure 6.12. MTBDD representation class relation diagram.

Class *Converter* is used for converting DT represented by DecisionTable.java into MTBDD represented by MTBDD.java and vices versa. Figure 6.13 represents a class structure of Converter.java. It consists of two main methods *DT_into_MTBDD()* and *MTBDD_into_DT()* for converting DT into MTBDD and vice versa. Methods *DTs_into_MTBDDs()* and *MTBDDS_into_DTs()* use two beforementioned corresponding methods for conversion of multiple MTBDDs into DTs and vice versa.

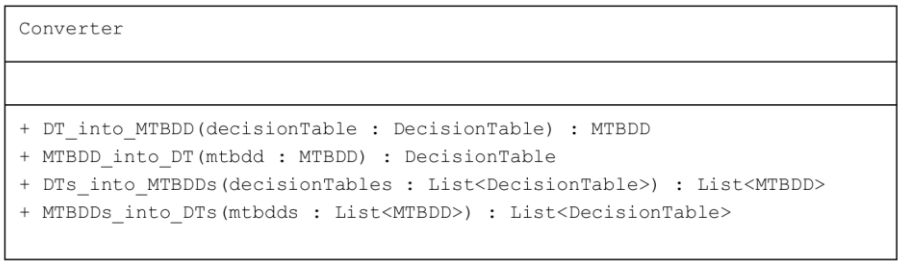

Figure 6.13. Converter class diagram.

MTBDDMutationOperators (Fig 6.14) class provides a set of methods which represent proposed mutation operators.

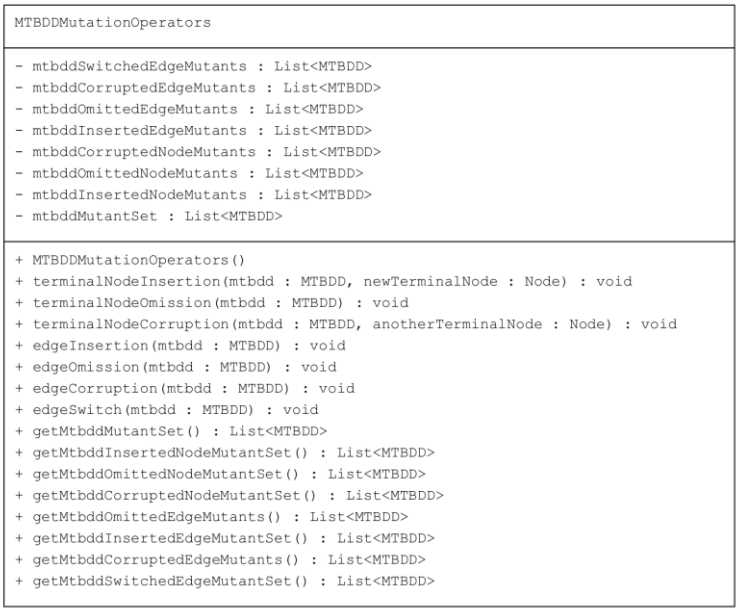

Figure 6.14. MTBDD mutation operators class diagram.

The ESG-DT is built on the ESG_DT class. The class diagram on Fig. 6.15 represents an event in ESG.

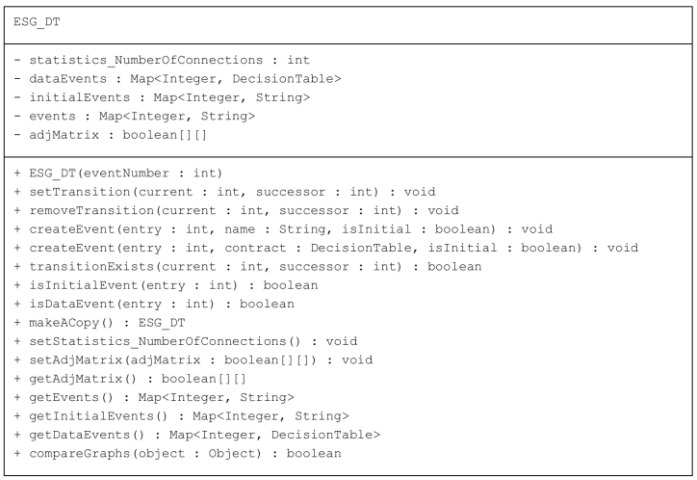

Figure 6.15. ESG_DT class diagram.

Class responsible for the model mutation is shown on Figure 6.19. Its purpose is only the mutation of the given model.

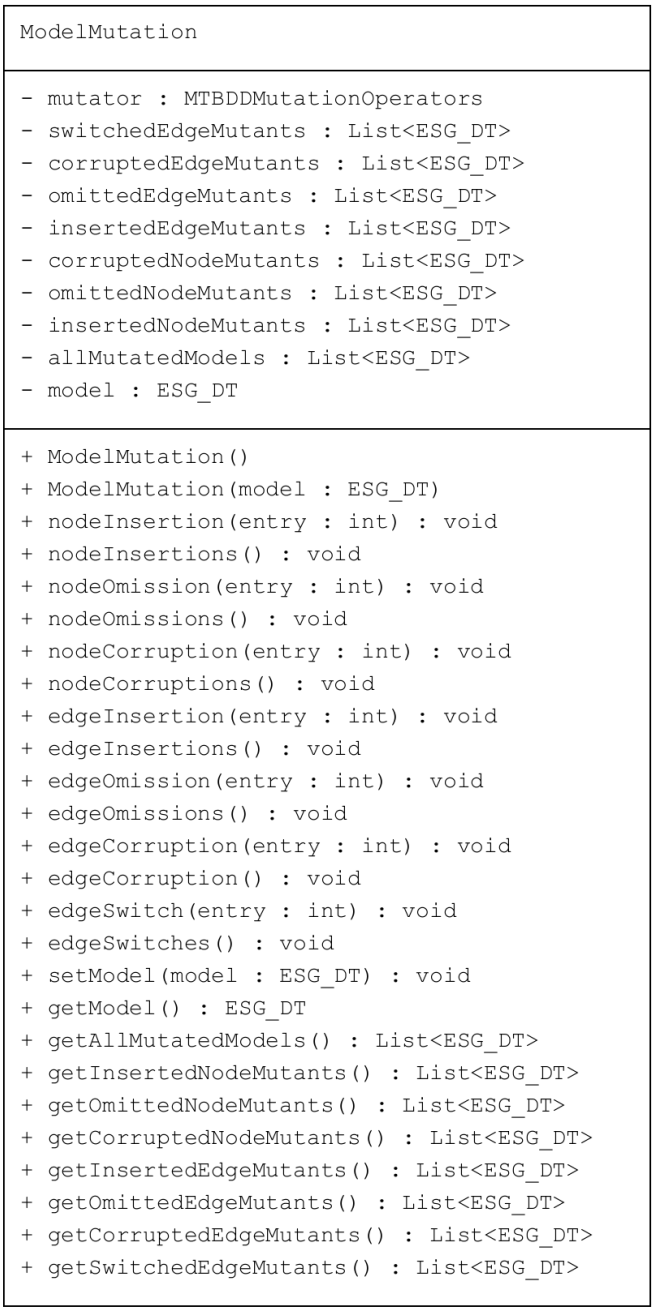

Figure 6.16. Model Mutator class diagram

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, mutation analysis is proposed for the specifications modeled using the Decision Table augmented ESGs. For the generation of mutants from this model representation, we proposed mutation operators for the contracts represented in Multi-Terminal Binary Decision Diagram form, obtained by translating the Decision Table inside the event. The presented mutation operators are: *edgeI*, *edgeO*, *edgeC*, *edgeS*, *tnI*, *tnO*, and *tnC*. The test cases are generated from the original ESG-DT graph and called test sequences. Test sequences are presented as CES and FCES. The proposed operators mutate MTBDD, thereby change the resulting DT.

The evaluation performed on the three cases shows that the ESG-DT mutants obtained after *tnC* application are always detected by both CES and FCES test sequences. The *tnO* and *tnI* operators return ESG-DT mutants detectable only by CES and FCES, respectively. On the contrary, mutants generated after the usage of *edgeI* and *edgeS* operators are never distinguishable by both CES and FCES test sets. The reason behind is that test sequences can reveal a mutant only at the ESG level of the ESG-DT. Depending on the number of the incoming edges of MTBDD terminal node, the *edgeO* and *edgeC* operators produce both distinguishable (only by CES) and indistinguishable by both CESs and FCESs. For conclusion, the detection of mutated model depends on the existence of the terminal node. That is why, *tnI*, *tnO* and *tnC* generated faulty models are always distinguished.

One direction for future work would be to propose an enhanced test sequence for detecting faults considered at the contract level of ESG-DT, as the existing test sequences are insufficient for this purpose. Another future work would be to improve the existing test sequence generation tool.

It is also noticed that the proposed mutation operators are unable to deal with the pseudo events *[* and *]* of the DT augmented ESG. The reason behind of this is that the contracts can be operated only with real events. The FCESs which start with [and

continue with the event which originally is not supposed to be one of the initial events, are never triggered. One other future work could be dedicated to the exploration of this phenomenon.

REFERENCES

(1) Budd, T. A.; Lipton, R. J.; DeMillo, R. A.; Sayward, F. G. *Mutation Analysis.*; YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE, 1979.

(2) Murnane, T.; Reed, K. On the Effectiveness of Mutation Analysis as a Black Box Testing Technique. In *Proceedings 2001 Australian Software Engineering Conference*; 2001; pp 12–20. https://doi.org/10.1109/ASWEC.2001.948492.

(3) Tuglular, T.; Belli, F.; Linschulte, M. Input Contract Testing of Graphical User Interfaces. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26* (02), 183–215. https://doi.org/10.1142/S0218194016500091.

(4) Khalilov, A.; Tuglular, T.; Belli, F. Mutation Operators for Decision Table-Based Contracts Used in Software Testing. In *2020 Turkish National Software Engineering Symposium (UYMS)*; 2020; pp 1–6. https://doi.org/10.1109/UYMS50627.2020.9247061.

(5) Yu-Seung Ma; Yong-Rae Kwon; Offutt, J. Inter-Class Mutation Operators for Java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*; 2002; pp 352–363. https://doi.org/10.1109/ISSRE.2002.1173287.

(6) Offutt, A. J.; Pan, J.; Tewary, K.; Zhang, T. An Experimental Evaluation of Data Flow and Mutation Testing. *Softw. Pract. Exp.* **1996**, *26* (2), 165–176. https://doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K.

(7) Andrews, J. H.; Briand, L. C.; Labiche, Y.; Namin, A. S. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Softw. Eng.* **2006**, *32* (8), 608–624. https://doi.org/10.1109/TSE.2006.83.

(8) Mutation Testing Approach to Negative Testing https://www.hindawi.com/journals/je/2016/6589140/ (accessed 2020 -11 -04).

(9) Meyer, B. Applying "Design by Contract." *Computer* **1992**, *25* (10), 40–51. https://doi.org/10.1109/2.161279.

(10) Jezequel, J.-M.; Deveaux, D.; Traon, Y. L. Reliable Objects: A Lightweight Approach Applied to Java. In *N O 4, July/August 2001*; 2001; pp 76–83.

(11) Traon, Y. L.; Baudry, B.; Jezequel, J.-. Design by Contract to Improve Software Vigilance. *IEEE Trans. Softw. Eng.* **2006**, *32* (8), 571–586. https://doi.org/10.1109/TSE.2006.79.

(12) Aichernig, B. K. Mutation Testing in the Refinement Calculus. *Form. Asp. Comput.* **2003**, *15* (2–3), 280–295. https://doi.org/10.1007/s00165-003-0011-8.

(13) Jiang, Y.; Xin, G.-M.; Shan, J.-H.; Zhang, L.; Xie, B.; Yang, F.-Q. Method of Automated Test Data Generation for Web Service. **2005**, *28*, 568–577.

(14) Ying Jiang; Shan-Shan Hou; Jin-Hui Shan; Lu Zhang; Bing Xie. Contract-Based Mutation for Testing Components. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*; 2005; pp 483–492. https://doi.org/10.1109/ICSM.2005.36.

(15) Krenn, W.; Aichernig, B. Test Case Generation by Contract Mutation in Spec#. *Electron. Notes Theor. Comput. Sci.* **2009**, *253*, 71–86. https://doi.org/10.1016/j.entcs.2009.09.052.

(16) Fabbri, S. C. P. F.; Maldonado, J. C.; Masiero, P. C.; Delamaro, M. E.; Wong, E. Mutation Testing Applied to Validate Specifications Based on Petri Nets. In *Formal Description Techniques VIII*; Bochmann, G. v., Dssouli, R., Rafiq, O., Eds.; IFIP Advances in Information and Communication Technology; Springer US: Boston, MA, 1996; pp 329–337. https://doi.org/10.1007/978-0-387-34945-9_24.

(17) Ammann, P. E.; Black, P. E.; Majurski, W. Using Model Checking to Generate Tests from Specifications. In *Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241)*; IEEE Comput. Soc: Brisbane, Qld., Australia, 1998; pp 46–54. https://doi.org/10.1109/ICFEM.1998.730569.

(18) Do Rocio Senger De Souza, S.; Maldonado, J. C.; Fabbri, S. C. P. F.; Lopes de Souza, W. Mutation Testing Applied to Estelle Specifications. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*; IEEE Comput. Soc: Maui, HI, USA, 2000; Vol. vol.1, p 10. https://doi.org/10.1109/HICSS.2000.926973.

(19) Fabbri, S. C. P. F.; Maldonado, J. C.; Sugeta, T.; Masiero, P. C. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*; IEEE Comput. Soc: Boca Raton, FL, USA, 1999; pp 210–219. https://doi.org/10.1109/ISSRE.1999.809326.

(20) Black, P. E.; Okun, V.; Yesha, Y. Mutation Operators for Specifications. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*; IEEE: Grenoble, France, 2000; pp 81–88. https://doi.org/10.1109/ASE.2000.873653.

(21) Black, P. E.; Okun, V.; Yesha, Y. Mutation of Model Checker Specifications for Test Generation and Evaluation. In *Mutation Testing for the New Century*; Wong, W. E., Ed.; Springer US: Boston, MA, 2001; pp 14–20. https://doi.org/10.1007/978-1-4757-5939-6_5.

(22) Sugeta, T.; Maldonado, J. C.; Wong, W. E. Mutation Testing Applied to Validate SDL Specifications. In *Testing of Communicating Systems*; Groz, R., Hierons, R. M., Eds.; Lecture Notes in Computer Science; Springer Berlin Heidelberg: Berlin, Heidelberg, 2004; Vol. 2978, pp 193–208. https://doi.org/10.1007/978-3-540-24704-3_13.

(23) Ling Liu; Huaikou Miao. Mutation Operators for Object-Z Specification. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*; IEEE: Shanghai, China, 2005; pp 498–506. https://doi.org/10.1109/ICECCS.2005.65.

(24) Belli, F.; Hollmann, A.; Wong, W. E. Towards Scalable Robustness Testing. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*; IEEE: Singapore, Singapore, 2010; pp 208–216. https://doi.org/10.1109/SSIRI.2010.14.

(25) Belli, F.; Budnik, C. J.; Hollmann, A.; Tuglular, T.; Wong, W. E. Model-Based Mutation Testing—Approach and Case Studies. *Sci. Comput. Program.* **2016**, *120*, 25–48. https://doi.org/10.1016/j.scico.2016.01.003.

(26) Belli, F.; Nissanke, N.; Budnik, C. J.; Mathur, A. Test Generation Using Event Sequence Graphs. *Softw. Eng.* **2005**, 52.

(27) Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* **1986**, *C–35* (8), 677–691. https://doi.org/10.1109/TC.1986.1676819.

(28) Tuglular, T.; Muftuoglu, A.; Belli, F.; Linschulte, M. Model-Based Contract Testing of Graphical User Interfaces. *IEICE Trans. Inf. Syst.* **2015**, *E98-D* (7), 1297–1305.

(29) Hermanns, H.; Meyer-Kayser, J.; Siegle, M. *Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains*; 1999.

(30) DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* **1978**, *11* (4), 34–41. https://doi.org/10.1109/C-M.1978.218136.

(31) Behaviour and State Change Models II https://personal.cis.strath.ac.uk/sotirios.terzis/classes/52.234_old/Behaviour%20and%20State%20Change%20Models_B.htm (accessed 2021 -04 -06).

(32) *On the Role of Test Sequence Length, Model Refinement, and Test Coverage for Reliability*; 2013.