International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

An Experimental Data Flow Analysis Annotated-Diagram-based event analysis

Hazaruddin Harun[1], Nurnasran Puteh[2]
[1,2]School of Computing, Universiti Utara Malaysia, 06010 UUM Sintok, Kedah, Malaysia

Abstract: With the adoption of Unified Modeling Language (UML) as the de-facto standard for modeling software systems, several research studies discussed the need for inputs for test automation to use models of systems under evaluation. Event diagrams have recently been used as a framework for test cases to be derived. Current studies concentrate on the study of event control flow. Nonetheless, it is quite easy to analyze the control flow between activities and such analysis alone is inadequate. This research proposes a test case generation methodology that complements a data flow information operation diagram. We performed an observational study into well-known structures in research literature to explore the potential benefits of this methodology. The experimental results were analyzed and compared as an alternative approach to the effectiveness and efficiency of fault detection with a state-of - the-art test suite generation tool. Overall, the results show that the proposed technique outperforms the alternative approach by detecting an average of 27.3 percent more errors. The proposed technique, in particular, yielded the best results in detecting errors related to arithmetic operations or parts used in our context for calculation.

Keywords: model-based testing; diagram-based testing of activity; data flow-annotated diagram of activity; data flow information

1. INTRODUCTION

Testing is an important part of the effort to develop software. Test design is the most demanding and time-consuming part of testing in particular. Testers need to develop cases to prove the presence of defects in order to test software. The key factor that reveals the extent of defects is the design of appropriate test cases. In addition to the adaptation of Unified Modeling Language (UML) diagrams as the de-facto standard for modeling software systems, the use of system models under test (SUT) as inputs for test models [1-3] has become necessary. The research community has therefore shifted its focus on designing and developing test cases based on various structural and behavioral models. In general, researchers emphasized the design of behavioral model-based test cases using activity diagrams (ADs) [4-15], sequence diagrams, state machine diagrams [3,16], and a combination of two or more diagram types [17]. To this end, the AD was considered an important artifact of layout for the identification of test cases [9]. Today, the main focus of existing studies is test automation focused on an AD study to obtain different types of tests of information on flow control. However, it is quite simple and straightforward to examine the control flow between design elements [18, 19]. Testing based solely on the sequence of activities in an AD is probably not enough to detect faults. It is therefore an important ongoing issue to find ways to improve the test quality based on design elements such as an AD. In addition to automatically analyzing its control flow, an activity represents system behavior to ensure its accuracy. In terms of data flow information, these activities also need further analysis. ADs are used to model process behaviors and how these behaviors interact by specifying the sequence of actions between them. Actions are considered to be the main activity capabilities and are central to the activity data flow [2, 17]. Empirical literature studies indicate that AD is among behavioral models the most comprehensive [20] and suitable development artifact describing the control flow between artifacts in an object-oriented system [9]. In addition, an AD is considered to be the best intermediate model between the specification of the program and the code which provides a rich source of data flow analysis details. Thus, for other purposes such as automatic code generation [21-23], ADs are further investigated. We performed an experimental investigation into an AD-based test case generation technique using data flow information (DFI) in this study. This analysis expands our earlier work [24] presented at a conference. To simplify the requirement for data flow coverage (DFC) the AD of a SUT is annotated with DFI. The inclusion of DFI, instead of the AD of a SUT without DFI, enables the analysis of the control flow data, allows the identification of pairs of object variables definition-use through activities that help the generation of highly improved test

cases. For example, it can be used to identify irregularities in the data flow that can also validate the template itself before extracting and executing test cases and deficiencies which need more accurate oracles such as calculation. It is mapped to an intermediate template, a so-called data flow graph (DFG), after annotating the AD with DFI, which is comparatively simple and more suitable for automated manipulation. Subsequently, the test paths are generated by DFG using specific DFC criteria in a depth-first search (DFS) manner, and the concrete tests are performed using the given oracles and input values. Ultimately, against predicted outcomes, the findings are presented and evaluated. The experimental investigation was conducted with the commonly used software systems to compare and evaluate new and existing literature testing tools and techniques. A comparative experimental investigation analysis using two techniques was conducted, namely data flow annotated activity diagram-based testing (DFAAD) and a so-called state-of - the-art test suites generation tool (EvoSuite) [25]. The impact of the proposed approach on effectiveness and efficiency in detecting faults has been discussed. Effectiveness in this analysis means revealing the maximum number of faults without considering the number of tests performed, whereas reliability means revealing the maximum number of faults with the minimum number of tests performed. The experimental investigation was conducted to further verify the arguments regarding efficacy and efficiency in detecting faults. Therefore, we aimed to address the general research question (RQ): how does the proposed DFAAD-based testing methodology perform in terms of the effectiveness of fault detection compared to a well-practiced alternative? Our general RQ was further broken down into sub-questions in Section 4. The rest of the analysis is structured as follows: Section 2 addresses related work and contrasts existing strategies for the generation of test cases based on AD. Section 3 provides the main approach including the basic concepts and meanings, examples to explain the overall concepts, a simple description to define and annotate DFI, as well as steps to separate the DFG from the DFAAD. Section 4 offers an observational analysis to examine the potential benefits of this method from an alternative research technique. Sections 5 and 6, respectively, address validity findings and risks. Finally, the conclusion and possible directions for the study are given in Section 7.

2. RELATED WORK

This section addresses related work and offers a simple comparison of existing literature studies involving techniques for producing AD-based test cases. ADs are used to model the SUT's complex behavior and are commonly used for testing support. These models are very useful and offer a significant testing opportunity as they accurately describe the SUT's functionality in a manner that can be easily manipulated by automation [26]. As well as the internal logic of a complex network, ADs can be used to model a system from a high-level business process to each individual unit of the system. Overall, there are a range of benefits associated with AD-based research, such as early generation of test cases during software design, good reporting of test cases, early detection of specification defects, and cost and effort reduction. There are several literature studies with different strategies that used ADs for the creation of test cases. An AD-based test case generation approach has been introduced [14] using UML 2.0 with a use case context based on a high-level abstraction AD. The study's aim is to identify lower synchronization and additional loop failures following guidelines for coverage of the activity path. An AD-based test method [11] builds trees for condition classification by collecting information on control flow from decision points and con ditions for guarding. The technique [10] known as automated test generation, by interpreting the AD, takes into account both the control flow and the information stream. By gathering data members to provide input for a new symbolic model checker (NUSMV), the analysis extracts and analyzes the structure of the AD. Nonetheless, the mapping of data members to the NUSMV input [27] and their contribution to the generation of test cases is not clear. The AD-based test case generation technique [5] generates test cases directly from the AD. The possible values of the input / output parameters were generated using a category-partition method to identify any differences between implementation and design. Similar to our research, this method creates test cases that can be used to test the system at code level. However, the technique still focuses on the operations / method sequences control flow in an AD and applies the criteria for the basic path coverage. In our study, we annotated the SUT's AD explicitly with data flow information and the criteria used to cover data flow. A rule-based approach [4] was presented to derive a combinatorial test design model from ADs in order to improve test effectiveness. The main idea is to provide rules with their corresponding values and constraints to identify the parameters by parsing the AD. Table 1 provides a simple comparison of the various perspectives, such as the technique used, the existing work, for further information on AD-based testing. The current studies are compared on the basis of our perception based on the

research objective, the intermediate model used to generate test cases, and the coverage criteria used, as well as issues related to these techniques.

Table 1. A comparison of activity diagram-based test case generation techniques

Study	Techniques	Objective	Intermediate	Coverage criteria	Related issues
[14]	UML 2.0 modeling capabilities with use case scope	Detecting synchronization and loop faults	Active Graph	Active path	High level of abstraction, missing details of individual activity
[11]	Condition classification tree	Test automation, generate tests early duration development	Condition classification tree	Decision point	Difficulties in identifying all feasible paths with complex control flow and their nested combination(loops)
[5]	Gray box	Test automation, find inconsistency between implementation and design	None	Basic path	Test cases are generated based on assumption that concurrent activity stated will not access the same object and only execute asynchronously
[6]	XML-based	Test automation, save time and effor	Activity dependency table(ADT)	Branch, predicate basic path	Lacking validation of fault detection capability with reduced set of generated test paths
[4]	Combinational test design model	Test automation, reduce effort, improve effectiveness	CTDM model	Parameter- value	Difficulties on identifying constrains from linking the parameters and values
[7,8]	I/O explicit activity diagram	Minimize number of TC	Directed Graph	All paths	Generalizability and vagueness on identification of input/ output activity, domain specific
[9]	Classificaton of control constructs	Identification of all possible scenarios	Intermediatee testable model	Selection loop adequacy	Generating and running too many test cases to cover every possible path is not feasible as it causes path explosion and reduces tes efficiency
[10]	Automated test generation using model checking	Test automation, reduce time and validation effort	Formal model(NUSMV input)	Activity, Transition Key_ path	Leading to state explosion, ambiguity on using data members for test generation

[12]	Automatically generate random TC	Test automaton, minimize number of TC consistency checking	None	Activity transition, simple path	Randomness limits the reliability of the generated test cases
[13]	Construct activity dependency table	To achieve all path coverage. Improve fault detection	Activity convert [8]	All paths	Manually generating AC grammar and feasible paths with complex AD including loops, detecting only design errors

The purpose of the comparative analysis is to provide a general overview of the current AD-based testing status. Although several related issues of the current AD-based testing techniques have been highlighted, we have not addressed every issue. The findings presented in Table 1 suggest that current methods focused primarily on test automation centered on the study of different control information objects in an AD. However, it is not enough to examine only the control flow between the activities to test the entire SUT. Though test automation based on the analysis of control flow information is a good idea to boost test output, it is more important to test reliability. In other words, the test cases generated should be able to detect errors. Accordingly, an ongoing challenge is improved detection capability of conventional AD-based testing. Through annotating ADs with DFI, the experimental investigation in this study strengthened the exposure criterion for increased ability to detect failures. To supplement traditional control flow-based testing techniques [28, 29], data flow-based testing techniques were initially introduced. Several research have examined the integration of DFI into a model-based test environment, using UML class diagrams [30] and state machine diagrams [19, 31, 32] for example. However, class diagrams are limited to the dynamic behavior of the static view of SUT and miss. State machine diagrams are also limited to representing the interaction of complex objects and do not represent all of the SUT's properties [33]. This thesis explores how annotating DFI can improve the ability to identify faults in the sense of an AD as opposed to existing studies.

3. DATA FLOW ANNOTATED ACTIVITYDIAGRAM-BASED TESTING

We define the basic concepts and meanings used in this analysis in this chapter and provide an overview of DFAAD-based testing. We also describe in more detail each of the activities, such as the circumstances to identify and annotate DFI and the steps to extract a DFG based on an annotated AD, along with a running example.

Basic Concepts and Definitions

Most of the definitions and terms used in this study are derived from standard textbook testing software [18, 34], existing documentation, and slightly modified research studies [2, 9,10].

Definition 1. Data flow-annotated activity diagram
DFAAD is an extension of the original activity diagram representing the sequence of actions that explicitly mark the flow of data across activities. A DFAAD can be formally described as a graph, G = (A, E, C) where:
A is a collection of actions / activities including A_0 and A_f, in which each A excluding A_0 and A_f is either annotated with d, u, cu or a combination thereof following the name of the stereotype notation data members in which d is specified, U stands for use, and *cu* stands for calculation use, and A_0 represents initial activity, where $A_0 \subseteq A$, and $A_0 \neq \emptyset$, and A_f is a set of final activities, where $A_f \subseteq A$ and A and $A_f \neq \emptyset$.
E refers to a set of edges in which E is a subset of A x A
$C = D_n \cup J_n \cup F_n \cup M_n$ is a set of control nodes, so D_n is a set of decision nodes, J_n is a set of join nodes, F_n is a set of fork nodes, and M_n is a set of fusion nodes. D_n is close to p-use, which stands for predicated use in terms of data flow.
Both the AD and the data flow graph that contains a single initial node are restricted.
Definition 2. Data flow graph A DFG is a simplified representation of a formally definable annotated activity diagram (AAD) as:
a set N of nodes, where each node is explicitly marked with DFId, u, cu, and p-use
a set N_0 of initial nodes, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
a set N_f of final nodes, where $N_f \subseteq N$ and $N_f \neq \emptyset$
a set E of edges, where E is a subset of $N \times N$

International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

Definition 3. All d-use path coverage (ADUPC) TR includes each direction d in S [18] for each array of def-pairs S = d-use (ni, nj, v). TR stands for test criteria, d-use for description use, and ni and nj, respectively, represent nodes I and j.

Overview of DFAAD-based Testing

Each paragraph provides an overview of DFAAD-based research and a detailed description of the main activities. According to [1], a standard MBT process involves three main tasks: developing a functional test model to reflect the SUT's expected operational behaviour, deciding requirements for test generation to limit the number of tests produced, and generating fully automated tests. We followed a medical MBT in this study in which the SUT's AD was established on a test basis. Compared to an MBT approach, there are three main activities in the DFAAD-based research, namely behavioral test template development, test generation, and test execution. The DFI was established and annotated within the SUT's AD to promote DFC requirements. For simplification and further automation, the AAD was mapped into an intermediate model known as the DFG.

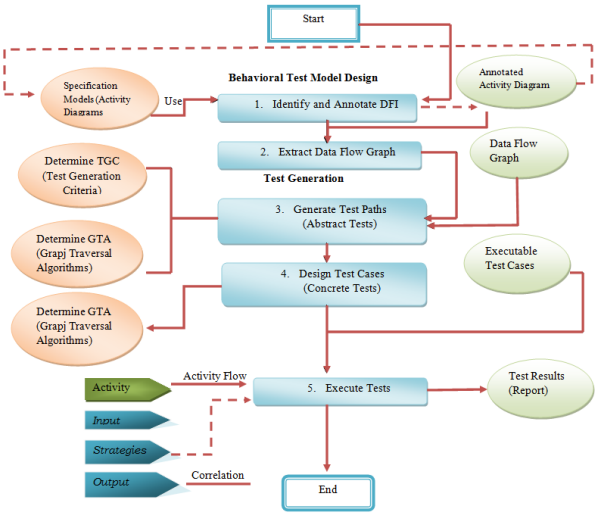


Figure 1. DFAAD-based testing overview

An intermediate test model used to create test paths based on existing graph traversal algorithms such as DFS or BFS was the deliverable of this operation. Testers need to establish the test generation requirements (e.g. dupath coverage) to limit the number of test cases created in order to generate the tests. Next, testers provided oracles and input values to adapt the test cases generated. The generated tests were finally carried out and the results of the tests were reported. Figure provides an overview of DFAAD-based testing. 1.

Identification and Annotation of DFI

This paragraph discusses the situations in which an AD will classify and annotate the DFI. The AD shows the control flows between operations, and ADs can model a system from a high-level business process to communication and state changes between activities, return values, and computations. They depict among activities the sequence of acts. For any significant capabilities, actions are needed and are essential to an AD's data flow aspect [2]. The sequence of acts between activities provides useful communications data such as the recipient and the receiver entity, state changes, parameter exchange, return values, and conditions of guard. Other supporting references also include specification documents, under particular the application case specification which is the basis of an AD. The measures and situations in which the DFI can be identified and annotated with the AD are as follows:

1) Identification of data members participating in an AD: first, annotation of an AD with DFI requires identification of the data members participating in the AD. There are several ways in which a data member can be identified through activities as mentioned above. Using the information in the guard condition is the easiest way to identify a data member. In addition, the input or output parameters specified in the Action Pin help to identify the members of the data.

Detection of DU pairs across activities: After the participating data members in an AD are identified, there are different situations in which the DU pairs of data members can be detected. The descriptive name and types of the action depict an action behavior. Thus, one approach is to analyze each action's encapsulated behavior. In addition, the input or output parameters specified in the Action Pin help to identify the members of the data. Detection of DU pairs across activities: There are different situations where the DU pairs of data members can be detected after the participating data members are identified in an AD.

2) The descriptive name and forms of the action represent an activity conduct. Thus, one approach is to analyze each action's encapsulated behavior. It is conceivable that the different DU pairs of data members across activities can be identified in the following situations: (a) A description can occur in an A in the following situations: in an Executable Node, which is the Activity Node origin, which outputs the variable objects (defines by input). In an executable Node (defined by assignment) in which the variable objects are initialized.

(b) Use in A may occur in the following situations: in an Executable Node invoked by other input data behaviors and operations, supply or return the information to other activities by means of outgoing edges without alteration. The data is obtained from the object of the recipient and transferred or returned to the object of the receiver without alteration. (C) In A, c-use may occur under the following conditions: in an Executable Node performing a subordinate behavior (e.g. arithmetic computation, object contents manipulation). (D) In A, p-use may occur in the following situations: in an AD node with Dn and a guard condition, C nodes are excluded without conditions of guard (e.g. Jn, Fn, Mn). DFI's annotation in an AD.

3) This information is annotated with the AD using assumptions, for instance, if a variable is specified in a particular activity it is interpreted as < < < d (variable name) > > after the data members are recognized and their DU pairs are detected. You can find additional information in the running instance and the chapter on case study.

Extraction of DFG

The AD syntax can be easily correlated with the DFG syntax with reference to the UML documentation and current studies. A DFG routinely encapsulates AD for further automation [14]. We can either extract test cases directly from an AD or convert them to an intermediate model DFG and generate test cases through the graph. We prefer to convert the AD into a DFG for simplification purposes, which simplifies the concepts by encapsulating different syntax of an AD into DFG nodes. The DU pairs annotated in an AD are labelled with the DFG's corresponding nodes. Since both the AD and the DFG are directed graphs, it is straightforward to map the annotated AD into the DFG and essentially involves the following steps:

The operation set A is mapped to the DFG node set N ($A_{Activity} \rightarrow N_{node}$) and the occurrence of d-use data is included with the respective nodes.

→An AD's A_0 node is connected to a DFG's N_0 node (A_0 N_0)

An AD's A_f node is mapped to a DFG's N_f ($A_f \rightarrow N_f$)

An AD's C_n nodes are mapped to a DFG's N (C_n N)

An AD's edge E is mapped to a DFG's edge E

Graph theory is a common concept of software testing which provides testers with a major simplification mechanism. An advantage of converting AD to DFG is that the structure of the DFG is more simplified. It is therefore relatively easy to generate test cases based on a DFG. Also, in the case of a DFG, there are already many algorithms to generate tests to traverse the graph.

Generation of Test Paths

The test paths are generated on the basis of the DFG intermediate test model. It is necessary to determine appropriate test coverage criteria (TCC) in order to limit the number of tests generated. The most commonly adopted coverage criteria include branch coverage, coverage of decisions, simple path coverage, and coverage of statements that require the test cases to cover and execute each branch, decision, statement, or path. We are not, however, able to discover most common mistakes [28].

The DFC is therefore added to complement the requirements for the control flow coverage, and the most well-known is the coverage of the DU road. We applied ADUPC to the DFC requirements in our approach. We are not, however, able to discover most common mistakes [28]. The basic concepts and definitions are set out in Section III-1, and some examples are given in Section III-3. A graph traversal algorithm (GTA) is required after determining the TCC to extract the abstract tests based on the selected criteria. DFS and breadth-first search (BFS) are the most common graph traversal algorithms. We crossed the DFG in a DFS way in our situation. By providing additional information such as oracles and input values, these paths are used to design concrete tests after generating the relevant test paths.

Design Test Cases

After generating and obtaining the abstract tests, test oracles and input values are needed to transform the abstract tests into concrete tests. This activity is an important aspect of the activity of test generation, which needs special attention. There are currently many literature guidelines and oracle testing strategies. Several oracle techniques have been documented for model-based testing and their ability to detect faults has been investigated [35]. Although our main concern is not the test oracle strategy, we have followed existing strategies and guidelines for designing the concrete tests.

Execution of Tests

Test cases were conducted against the SUT as the final operation, and the test results were published. There are currently several test output systems that can be used to conduct planned test cases. We used J Unit as an execution tool in this study to facilitate comparative

analysis against the alternative approach, but any other tools can be used to perform the generated executable tests.

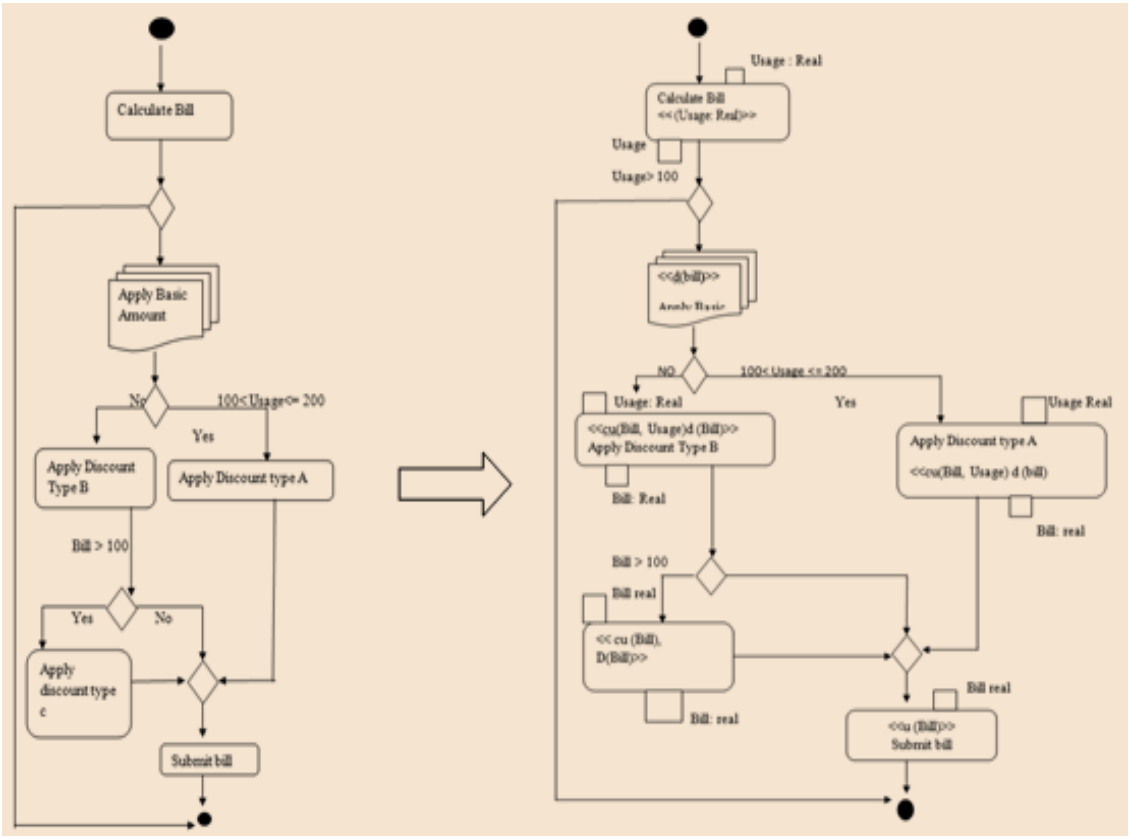


Figure 2. (a) Customer billing service AD; (b) Customer billing service DFAAD

4. RUNNING EXAMPLES

We provide two running examples in this section, using a customer billing service application and a triangle problem, for ease of understanding. The first example illustrates the data flow description in an AD and the latter also demonstrates DFG generation and test cases.

Example 1

We used a modified version of a cellular service customer billing software as a first running example [36]. The software measures billing on the basis of the amount of use of the customer and provides three different types of discount. The following is a basic case specification definition for the use of "Calculate Billing Service":

Normal flows

The application receives the amount of use from the system actor

If the use reaches zero, the bill will be determined based on the type of discount.
A discount type A (50 cents for each additional minute) is applied if the consumption is between 100 and 200.
If the use exceeds 200, a discount type B (10 cents for each additional minute) will be applied.

Alternate/exceptional flows

1 If the use is below or equal to zero, the bill shall be zero
1 If the bill reaches 100, a Type C discount (10% discount from the total amount) is applied.
The definition of use case is the basis for the creation of an AD. It is also considered a rich knowledge source to identify DFI in an AD. Figure shows the original AD for billing operation. 2(a), and Figure displays the corresponding AAD. 2(b). The AD is the series of actions needed to calculate the amount of the billing. Acts are known to be the main activity capabilities and are central to the aspects of information flow [2]. We can use Action Pins to represent transmitted data values to and from an Action. We use the three previously defined measures to

annotate the AD with DFI. The first step is to identify participating data members through activities using the information in the conditions of the guard and the parameters put in / out specified in the pins of action. In the picture. 2(b), by examining the action pins and conditions of the guard, a total of two data members (Usage, Bill) were defined. Subsequently, through examining the encapsulated actions of Activities, we detected the DU pairs of each identified data member through activities. The AD was annotated in the final step with the established DU pairs. Table 2 shows the DU pairs associated with the information members in a simple format through different activities for better understanding. Table 2 also provides abstract and concrete comparisons of input values and predicted outputs. In this case, only dummy values used for demonstration purposes are the input and expected values for Table 2. The abstract test cases are collected directly through the DFAAD manually. Table 2 reveals that the Bill parameter appears as define-definine, which is a double description known as a possible error. Detection in the design of such anomalies helps to prevent different code anomalies.

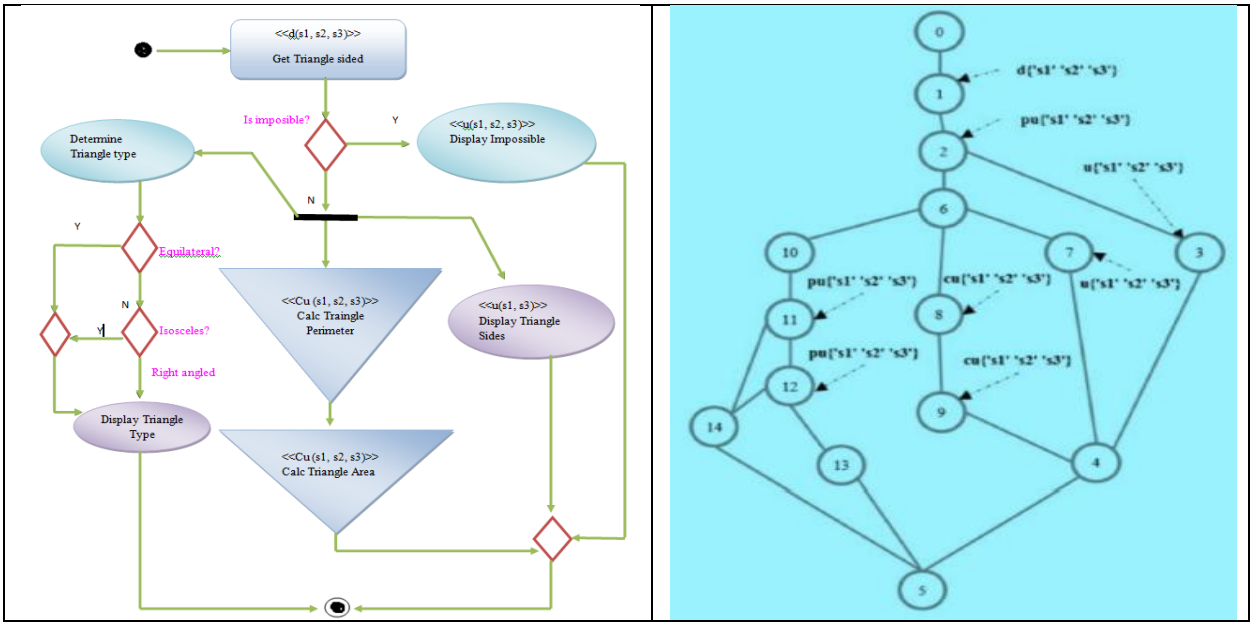
Table 2. du-pairs of data members across activities, the abstract and concrete tests

Variable	Activity Nodes									
	Calculate Bill	Decision (node 1)	Basic Amount	Decision (node 2)	Discount Type A	Discount Type B	Decision (node 3)	Discount Type c	Merge node	Submit Bill
Usage Bill	d d	pu	d	pu	Cu Cu, d	Cu Cu, d	pu	Cu, d		u
	Abstract Tests								Concrete tests	
									Input Value	Expected output
Usage	Calculate Bill → A		Decision node 1→		Decision node 2 →		Discount Type		199	89.5
du-pairs	Calculate Bill → B		Decision node 1 →		Decision node 2 →		Discount Type		900	99.0
Bill	Apply basic Amount → Discount Type A								150	140
du pairs	Apply basic Amount → Discount Type B								230	110
Discount type A → Submit bill									120	120
Discount Type B → Decision node 3 → Submit bill									600	400
Discount Type B → Decision → Discount Type C → Submit bill									200	120
Discount Type C → Submit bill									300	150

International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

Example 2

This example shows the annotation of the data flow, the conversion of DFAAD to DFG using triangle problem. Specification of the triangle problem: three positive total numbers are taken as input by the triangle problem. Classify it as an equilateral triangle if all three sides have the same length;
When two sides have the same size, classify it as a triangle of isosceles;
If a right angle is an angle, mark it as a right angle triangle;
Classify it as a scalene triangle if all sides have different lengths and no right angles;



(a) (b)

Figure. 3 a) DFAAD triangle problem, b) DFG triangle problem

Table. 3 Concrete analysis and reliability findings for the triangle problem

Coverage	TC ID	Test Paths	Input Values			Expected outputs	Result
			s1	s2	s3		
ADUPC	TC1	1-2-3- pu(s1,s2, s3)	20	0	10	Impossible	P
	TC2	1-2-6-7u (s1, s2, s3)	40	50	20	40,50,20	P
	TC3	1-2-6-8cu (s1, s2, s3)	60	70	80	210	P
			60	0	80	-1	P
	TC4	1-2-6-8-9 cu(s1, s2, s3)	40	60	60	1131137.085	F
	TC5	1-2-6-10-11-12pu(s1, s2, s3)	200	200	200	Equilateral	P
	TC6	1-2-6-11-14-5pu(s1,s2,s3)	13	55	55	Isosceles	F
	TC7	1-2-6-10-11-12-13-5pu(s1, s2,s3)	12	21	33	Impossible	F

If there is no triangle on the given sides, mark it as impossible;
Calculate the perimeter and the area of the triangle
The triangle problem's DFAAD is shown in Fig. 3(a) and the corresponding DFAAD information flow diagram is shown in Fig. 3(b) that's wrong. The abstract test cases generated in Fig from the DFG. 3(b) is shown in Table 3 on the left.

5. EXPERIMENTAL DESCRIPTION

The paragraph describes the experiment conducted to test the proposed methodology, following existing guidelines for software engineering empirical research and experimentation [37-39]. Section IV-A provides the concept and scope of the experiment. Section IV-B explains the experimental procedure and design, and the experimental results are described in Section IV-C.

Experimental Definition and Context

The experimental investigation was conducted with an alternative approach to contrast and assess the existing DFAAD-based research methodology. Therefore, the experiment concentrated on testing the possible efficacy of the proposed technique to expose faults with respect to the formulated research questions compared to an established well-practiced test case generation technique. We therefore carried out our experimental research involving three subject systems, namely the Cruise Control System, the Elevator System, and the Coffee Maker.

We were chosen manually based on the following parameters to ensure that the experimental samples were appropriate (e.g. in terms of size and complexity) and suitable for our study. Large and logically complex systems with at least four classes, 60–80 branches, and 150–170 non-comment statements Systems that include all the necessary objects available (e.g. category diagrams and a high-level explanation of the functionalities of the system) to model the behavior of the system. Systems that are not excessively large or complex, or inappropriate for alternative approaches to generate tests, which prevent experimentation within the time limit. The complete description of the three experimental topic structures is listed in Table 4. The Software-Artifact Infrastructure Repository (SIR) for Elevator and Cruise Control Systems (https:/sir.csc.ncsu. edu / content / sir.php)[40] and the coffee maker's NCSU website (https:/www.ncsu.edu/) provide all the necessary objects, including the system source code. The experiments deal with the following RQs and sub-RQs that extend our general RQ provided in Section I. RQ1.1: Compared to an alternative approach, how does the proposed DFAAD-based test case generation technique perform in terms of overall failure detection efficiency?
RQ1.2: What is the difference between the proposed DFAAD-based test case generation technique and the alternative approach to the detected type of faults?
RQ2: What is the relative efficiency, measured by the number of detected faults and the number of generated test cases?

Table. 4 Description of experimental subject systems

System	#LOC#	#Classes	Mutants		
			Min	Mean	Max
Cruise control	358	4	15	27.25	48
Elevator	581	8	2	30.9	111
Coffee maker	393	4	24	39	68

RQ1.1 explores whether the proposed methodology can be comparable to a well-practiced test suite generation method like EvoSuite in terms of identification of faults.
RQ1.2 further explores whether the proposed strategy was more or less successful than the alternative approach in detecting distinct faults. For example, a positive response to this query would imply the detect ability with the DFAAD-based approach of certain types of fault, which is not observable with the alternative approach.

RQ2 investigates whether the test suites produced by one technique may detect additional faults with fewer tests compared to others.

Variable selection

For all RQs, the independent variable is the type of technique used as the basis for the generation of test suite (e.g. DFAAD or alternative technique). The dependent variables are correlated with fault detection efficacy and

efficiencies, such as killed or survived mutants, and different types of fault.

Mutation seeding:

Mutat Fault instrumentation is a common approach used in software testing, to generate mutants for our experimental subjects. We used the PIT mutation testing system, a recently developed automated mutation testing tool that works quickly at the level of byte codes (http:/pitest.org). An empirical study of the efficacy of mutation testing tools [42] reported PITEV's (evaluation version) outperformance compared to other tools.
PIT offers three levels of preferences for mutators (default, stronger, and all mutators). PITEV (evaluation version) documented outperformance compared to other methods in an empirical study of the effectiveness of mutation testing tools [42]. PIT offers three levels of mutator preferences (default, stronger, and all mutators).

Experimental Procedure and Design

This research focuses on comparing and testing the existing method through an alternative approach in terms of effectiveness and efficiency in detecting faults. We therefore defined the terms efficiency and effectiveness in the context of this study as follows:

Effectiveness (E)

The goal of effectiveness is to identify as many faults / killed mutants as possible planted in a system. It is calculated on the basis of the ratio between the number of faults / killed mutants per technique detected and the total number of faults / seed mutants present. In this report, interchangeably use the words ' mutants destroyed ' and ' faults found. '

$$E = \frac{\#\ of\ killed\ mutants}{Total\ numbers\ of\ seeded\ mutants} \times$$

Efficiency (EF)

The performance goal is to find the maximum number of errors with the minimum test case sets. It is calculated based on the ratio of the number of errors detected / mutants killed per procedure and the number of test cases associated with it. The output therefore shows on average the number of observed faults per executed sample.

$$EF = \frac{\#\ of\ detected\ faults}{\#\ of\ generated\ executable\ test\ cases}$$

The experiment was conducted taking into account all the activities outlined in Section III as follows: (1) Because the DFAAD models were not available, the student generated the requisite ADs for each subject system using the Enterprise Architect modeling method, provided the three systems with all the appropriate objects. (2) By applying the three steps, the DFI has been defined and annotated in the SUT ADs. (3) DFAADs were used to derive a data flow chart and to produce abstract test cases. (4) A concrete test with oracles and input values has been designed. The test against the SUT was eventually conducted and the findings were announced. The DFAADs, as defined in Section III, contain the action sequences, guard conditions, forks, joins, data members, and input parameters listed in the Action Pin. The information stream is annotated directly by operation. Figs. Examples of these DFAADs for car simulator and coffee maker behaviors are shown in 4 and 5 respectively. DFAADs can be very complex or very simple, depending on the nature of the SUT. For example, a model of a running car simulation algorithm (Fig. 4) is quite complex compared to a coffee maker adding, removing, or editing recipes. These variations in the nature of experimental samples have a substantial impact on the performance of the techniques or procedure used, and are good examples of their efficacy evaluation and comparison.

Alternative approach

We selected EvoSuite4, a so-called state-of - the-art test suite generation method [25, 42] as an alternative approach to comparison and evaluation. Generating test cases with EvoSuite is a simple task performed by right-clicking and pressing EvoSuite to create results. We selected EvoSuite4, a so-called state-of - the-art test suite generation method [25, 42] as an alternative approach to comparison and evaluation. Generating test cases with EvoSuite is a simple task performed by right-clicking and pressing EvoSuite to create results. We selected EvoSuite because it was widely practiced through different open-source types and sizes as well as industrial software programs, recording many real faults [43]. EvoSuite has achieved the highest overall scores in the device competition for SBST 2016 and 2017 [44, 45]. In addition, EvoSuite provides support for the detection of PIT mutation.

International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

Experimental Results

EvoSuite has achieved the highest overall scores in the device competition for SBST 2016 and 2017 [44, 45]. In addition, EvoSuite provides support for the detection of PIT mutation. Each chapter presents results obtained from the three topics of study.

RQ1.1: Compared to an alternative approach, how does the proposed DFAAD-based test case generation technique perform in terms of overall failure detection efficiency?

RQ1.1 evaluates the potential of the two methods to overall identification of faults.

Fig. 6 The findings of the overall contrast between the two methods are discussed. The figure shows how many faults different techniques have detected or failed to detect and their related effectiveness scores (in percentages).

From Fig as can be seen. 6. Test cases developed by DFAAD have been successfully applied to all experimental subjects with an average efficacy of 67.9% for Cruise Control, 69.6% for Elevator and 84.6% for Coffee Maker. In the case of Cruise control, however, test cases resulting from the alternative approach managed to achieve 44 percent efficacy, in the case of Elevator 24.3 percent and in the case of Coffee maker 83.8 percent. The results showed an overall difference in efficiency of 23.9% for the Cruise control system and 45.3% for the Elevator system. By comparison, in the case of Coffee maker, both methods achieved a comparable level of effectiveness. In the case of the Elevator system, the observed difference in effectiveness was important. Analysis also explains the factors leading to these inequalities. The results in Fig. 6 Indicate DFAAD's enhanced ability to detect errors relative to the alternative approach. To assess if the observed difference is statistically significant, we performed a non-parametric rank test signed by Wilcox on [46] with a statistical significance rate below 0.05. The following were the null and alternative hypotheses: H0: no difference between the percentage of DFAAD identified faults and alternative approaches; H1: a disparity between DFAAD's observed proportion of faults and alternative approaches. Consequently, the rank test signed by Wilcox on revealed that the DFAAD (mean rank= 7.36) was considered more effective than the alternative (mean rank= 5.0) with p-value= 0.013 and z-score= -2.488. We therefore dismissed the H0 null hypothesis and concluded that statistically significant was the observed discrepancy.

RQ1.2: What is the difference in efficacy between the proposed DFAAD-based test case generation technique and the alternative approach to the form of observed faults? This query measured the efficacy of each technique in terms of the types of fault found. The findings for each experimental topic and the types of fault found are described in Tables 5–7. That table shows the total number of seeded mutants associated with each mutation operator, the number of mutants killed and survived and the respective effectiveness levels. The most outstanding results are bold and discussed further in the article.

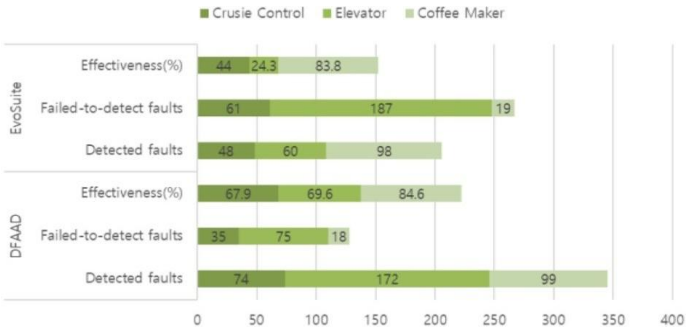


Figure 6. Overall comparison of the two approaches in terms of fault detection effectiveness

Image. Fig. 7 in addition, the types of mutants killed by each strategy are visually outlined through subject systems.

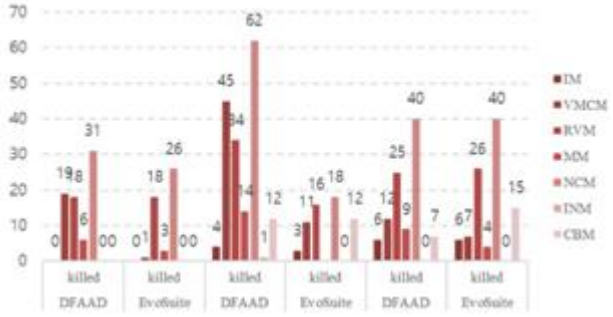


Figure. 7 Comparative analyses of killed mutants across subject systems

As can be easily observed from the tests, in the case of Coffee maker, which is further discussed in the following section, DFAAD succeeded in killing more mutants in nearly all types of mutation operators across subject systems except CBM.

RQ2: What is the relative performance, measured by the number of detected faults and the number of produced test cases?

The response to RQ2 offers insights into the relative efficiency of the techniques in terms of the number of mutations killed per test performed. We found that with less functional sample sets, DFAAD destroyed more mutants. The average number of killed mutants performed was: 2.6 mutants versus 0.6 in the case of the Elevator, 2 mutants versus 0.7 in the case of the Cruise

Control, 3.7 mutants versus 1.1 in the case of the Coffee Maker, and finally 2.6 mutants versus 0.8 in the overall subject systems. Image. Fig. 8 summarizes a cross-comparison of two test case generation techniques ' output.

The most remarkable results are bolded

Table 5. Cross-comparison between observed forms of faults and the cruise control system

Mutation operators	Total mutants	Cruise control					
		DFAAD			EvoSuite		
		killed	Survived	Effectiveness (%)	Killed	Survived	Effectiveness (%)
IM	0	0	0	0	0	0	0
VMCM	26	19	7	73	1	25	3.8
RVM	20	18	2	90	18	2	90
MM	20	6	14	30	3	17	15
NCM	33	31	2	94	26	7	78.8
INM	0	0	0	0	0	0	0
CBM	10	0	10	0	0	10	0

Table 6. Cross-comparison with the elevator network of observed fault forms

Mutation operators	Total mutants	Elevator					
		DFAAD			EvoSuite		
		killed	Survived	Effectiveness (%)	Killed	Survived	Effectiveness (%)
IM	4	4	0	100	3	1	75
VMCM	76	45	31	59	11	65	14
RVM	45	34	11	76	16	29	35
MM	30	14	16	47	0	30	0
NCM	70	62	8	89	18	43	26
INM	1	1	0	100	0	1	0
CBM	21	12	9	57	12	9	57

Table 7. Cross-comparison of detected fault types with Coffee maker system

Mutation operators	Total mutants	Coffee maker					
		DFAAD			EvoSuite		
		killed	Survived	Effectiveness (%)	Killed	Survived	Effectiveness (%)
IM	6	6	0	100	6	0	100
VMCM	12	12	0	100	7	5	58
RVM	26	25	1	96	26	0	100
MM	9	9	0	100	4	5	44
NCM	40	40	0	100	40	0	100
INM	0	0	0	0	0	0	0
CBM	24	7	17	29	15	9	62

International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

6. DISCUSSION

This chapter addresses the efficacy and reliability of DFAAD-based research on fault detection. To compare and test the suggested method, the experimental results are used. We have formulated a variety of RQs to better understand and summarize the experimental results and updated our discussion as follows: Discussion of RQ1.1: We performed a statistical analysis of the derived data to improve the reliability of the results on this issue.

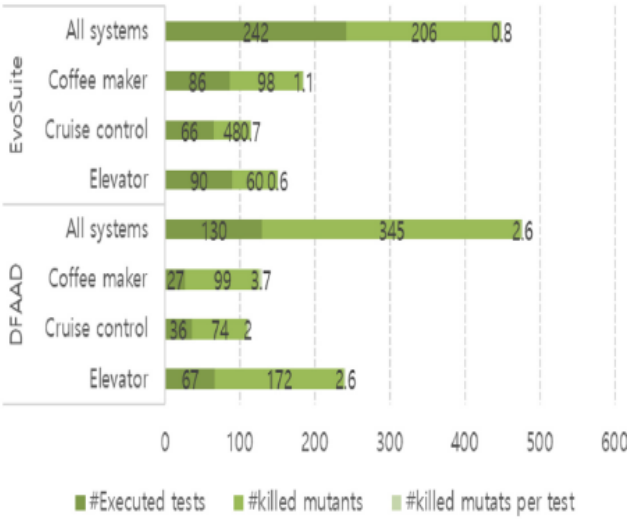


Figure 8. Summary of efficiency results based on cross-comparison

We used the Wilcox on signed-rank test, a non-parametric statistical hypothesis test with a statistical significance < 0.05, due to the small sample size and non-normal distribution of the results. Ultimately, the proposed null hypothesis (p- value = 0.013) was dismissed in order to conclude that the DFAAD was able to achieve greater efficiency. Fig. 9 The average distribution ratio of observed and undetected faults across DFAAD and EvoSuite for all subject systems is shown. Like EvoSuite, in the case of observed errors, the DFAAD yielded a higher mean and was lower in the case of failure to detect events. It was found that the DFAAD is relatively effective in detecting errors across all study subjects. However, the effectiveness of the alternative approach varied among different systems. For example, in the case of Coffee maker, the alternative approach resulted in better efficiency and performed fairly well in the case of Cruise control but in the case of Elevator system it was very weak. Perhaps this result was due to the system's more complex and dynamic run time behavior. EvoSuite, for example, is excellent at optimizing protected branches and claims, but is not adequately capable of handling the SUT's real-time properties. In the case of DFAAD, however, the system's real-time actions are more accurately recorded to strengthen test cases.

Discussion of RQ1.2: This problem investigated whether, compared to the alternative, the proposed approach was more or less successful in detecting different types of faults. Fig. 10 Displays the distribution of the identification of faults between the mutation operators and the techniques employed. As you can see from Fig. 10 Unlike the alternative approach, the DFAAD has, on average, been able to detect a higher number of defects with respect to all mutation operators except CBM. In particular, the average detection of NCM mutants (mean difference + 16.33), MM mutants (mean difference + 7.33), RVM mutants (mean difference + 6.66), and VMCM mutants (mean difference + 19) were improved by the DFAAD. Nonetheless, in identifying CBM mutants, the alternative approach demonstrated greater effectiveness (mean difference + 2.66). In the case of Coffee maker, the disparity is significant and may be due to the lack of usability when designing the AD. It is likely that even if the system is less complex, less effort or attention may have been exerted by the modeler to properly model the system. No notable difference was found in the identification of IM-related faults between the two approaches. In the case of Elevator, which is protected by DFAAD, only one INM mutant was created by the mutation method.

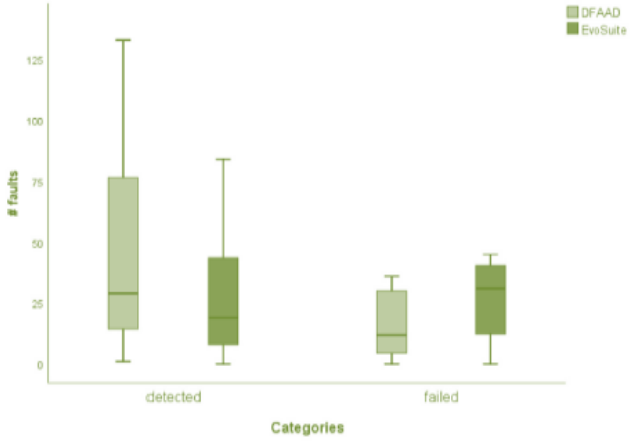


Figure 9. Average ratio of observed and undetected error distribution through techniques

International Innovative Research Journal of Engineering and Technology
ISSN: 2456-1983 Vol: 5 Issue: 2 December 2019

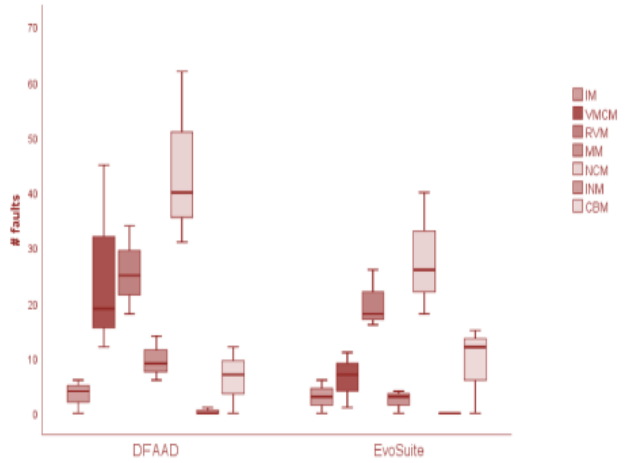


Figure 10. Distribution of faults detection ratios across mutation operators and techniques

RQ2 discussion: It is possible to measure the performance or cost-effectiveness of test case generation techniques from different perspectives such as test execution and sample generation. In published studies, cost-effectiveness is calculated in different ways, for example, based on the size of test drivers in terms of LOC, the execution time of the CPU, the number of calls of methods [33], and the proportion of faults identified per separate assertions created[35]. We assume that the test case generation output is proportional to the size produced by the executable tests. Therefore, the performance is determined by detecting as many faults as possible with minimal checks. Our results showed that test cases produced using DFAAD were able to detect an average of 2.6 faults, while test cases based on the alternative approach detected an average of only 0.8 faults. To sum up this discussion, EvoSuite failure detection performance differed from system to system, while system-wide DFAAD performance was nearly constant. In the case of the elevator system, EvoSuite effectiveness was quite weak and slightly better in the case of the coffee maker, indicating that EvoSuite effectiveness depends heavily on the existence of SUT. For example, EvoSuite is not a full cable of test systems with simultaneous and complex run-time behaviors. DFAAD, on the other hand, has proved to be best suited for such systems forms. In addition, these results suggest that DFAAD has allowed the identification of additional arithmetic operation-related faults that are considered more critical and difficult to detect. While EvoSuite is good at optimizing branches and covering statements, our results indicate that it does not yet represent the completeness of the experiment. DFAAD provides the full benefits of model-based analysis in addition to providing greater efficacy in detecting faults. Therefore, the pros and cons of model-based analysis or its comparison with other methodologies will not be discussed further.

7. THREATS TO VALIDITY

We discuss various possible threats to the credibility of the experimental research carried out and how they can be mitigated in this paragraph. External validity risks apply to problems that have an effect on the conclusions reached, such as using the original source code without errors. Different conclusions could be drawn if real fault structures were used. Nonetheless, it is not easy to identify appropriate devices with real experimental faults, so fault instrumentation is a common practice in research studies. For test suite creation and PIT mutation testing software for fault seeding we used EvoSuite as an alternative approach. Therefore, the efficacy depends heavily on the characteristics of the devices used. Therefore, using different test suite creation techniques or methods (e.g. manually produced, Randoop [47]) or using actual faults or different mutation tools (e.g., MuJava) will produce different results. We also failed to consider the impact of human subjects in the generation of test cases in our research. For example, in the DFAAD-based approach, the outcomes can be significantly influenced by individuals modeling the structures and their role in modeling. External validity challenges contribute to our study's Generalizability concerning the research topics and the forms of fault used. Despite our efforts to produce valid results, we cannot be entirely certain about the Generalizability of the chosen topics, as the findings are always linked to the SUT. Through choosing different systems like our experimental samples, for example, with a different domain, scale, form, and complexity, we may have obtained different results. We considered limited requirements concerning the viability of the systems in order to minimize this risk when selecting the subject systems. Similarly, it is not possible to generalize in all situations the method used for mutant seeding and the types of mutants given by such a tool. Nevertheless, we have ensured that well-established and common resources are chosen, which are actively supported. Threats to construct credibility are compatible with the Generalizability and suitability of the measures used in our experiments. To equate our technique's fault detection capacity with the alternative approach, we used the fault detection ratio, widely used in studies with reliable results to determine test techniques [41]. In addition, the ratio between the number of detected faults and the number of usable test cases produced in our experiment may not

reflect the actual efficiency of the test. For example, if we used alternative metrics such as the ratio of detected faults to produced distinct assumptions or the number of call methods, the results could have been different.

8. CONCLUSION AND FUTURE WORK

In this study, we have suggested a technique for the generation of cases based on DFAAD. The SUT AD is annotated with DFI to promote DFC parameters and define DU pairs of object variables through activities to model more appropriate test cases. We conducted an observational study using three widely used methods in software testing literature to examine the possible efficacy of the proposed technique. Compared to an alternative state-of - the-art test suite generation tool, the efficacy and efficiency of the proposed technique was evaluated. The statistical significance of the experimental results suggests that in terms of effectiveness, the new method outperforms EvoSuite. The findings also showed that the technique proposed was comparatively more effective in detecting critical faults (e.g. faults related to arithmetic operations). It was quite surprising that the results indicate that, given the full advantages of MBT, the proposed technique allowed the identification of a wide range of faults, some of which can not be identified using an alternative approach. In the future, we have a project to perform detailed experimental studies involving additional variables such as time and cost efficiency for different applications. We will also build supporting tools for our test case generation technique to enable automatic mapping of the AD with the related DFG and to allow automatic detection of entity variable DU pairs through activities.

REFERENCES

[1] I. Schieferdecker, “Model-based testing,” IEEE Software, vol. 29, no. 1, pp. 14-18, 2012

[2] Object Management Group, "Unified Modeling Language Specification Version 2.5.1," 2017; https://www.omg.org/ spec/UML/About-UML/.

[3] M. Shirole and R. Kumar, “UML behavioral model based test case generation: a survey,” ACM SIGSOFT Software Engineering Notes, vol. 38, no. 4, pp. 1-13, 2013.

[4] P. Satish, K. Sheeba, and K. Rangarajan, “Deriving combinatorial test design model from UML activity diagram,” in Proceedings of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013, pp. 331-337

[5] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng, “Generating test cases from UML activity diagram based on gray-box method,” in Proceedings of 11th Asia-Pacific Software Engineering Conference, Busan, Korea, 2004, pp. 284-291.

[6] P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba, “An enhanced test case generation technique based on activity diagrams,” in Proceedings of 2011 International Conference on Computer Engineering & Systems, Cairo, Egypt, 2011, pp. 289-294.

[7] H. Kim, S. Kang, J. Baik, and I. Ko, “Test cases generation from UML activity diagrams,” in Proceedings of 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), Qingdao, China, 2007, pp. 556-561.

[8] P. Mahali, S. Arabinda, A. A. Acharya, and D. P. Mohapatra, “Test case generation for concurrent systems using UML activity diagram,” in Proceedings of 2016 IEEE Region 10 Conference (TENCON), Singapore, 2016, pp. 428-435

[9] A. Nayak and D. Samanta, “Synthesis of test scenarios using UML activity diagrams,” Software & Systems Modeling, vol. 10, no. 1, pp. 63-89, 2011.

[10] M. Chen, P. Mishra, and D. Kalita, “Coverage-driven automatic test generation for UML activity diagrams,” in Proceedings of the 18th ACM Great Lakes Symposium on VLSI, Orlando, FL, 2008, pp. 139-142.

[11] S. Kansomkeat, P. Thiket, and J. Offutt, “Generating test cases from UML activity diagrams using the Condition- Classification Tree Method,” in Proceedings of 2010 2nd International Conference on Software Technology and Engineering, San Juan, PR, 2010.

[12] C. Mingsong, Q. Xiaokang, and L. Xuandong, “Automatic test case generation for UML activity diagrams,” in Proceedings of the 2006 International Workshop on Automation of Software Test, Shanghai, China, 2006, pp. 2-8.

[13] K. Pechtanun and S. Kansomkeat, "Generation test case from UML activity diagram based on AC grammar," in Proceedings of 2012 International Conference on Computer & Information Science (ICCIS), Kuala Lumpur, Malaysia, 2012, pp. 895-899.

[14] D. Kundu and D. Samanta, "A novel approach to generate test cases from UML activity diagrams," Journal of Object Technology, vol. 8, no. 3, pp. 65-83, 2009.

[15] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," International Journal of Engineering & Technology IJET-IJENS, vol. 11, no. 3, pp. 1-21, 2011.

[16] C. H. Chang, C. W. Lu, W. C. Chu, X. H. Huang, D. Xu, T. C. Hsu, and Y. B. Lai, "An UML behavior diagram based automatic testing approach," in Proceedings of 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops, 2013, pp. 511-516.

[17] S. Dahiya, R. K. Bhatia, and D. Rattan, "Regression test selection using class, sequence and activity diagrams," IET Software, vol. 10, no. 3, pp. 72-80, 2016.

[18] P. Ammann and J. Offutt, Introduction to Software Testing. New York, NY: Cambridge University Press, 2008.

[19] A. Rauf, "Data flow testing of UML state machine using ant colony algorithm (ACO)," International Journal of Computer Science and Network Security, vol. 17, no. 10, pp. 40-44,
[20] 2017.

[21] M. Felderer and A. Herrmann, "Comprehensibility of system models during test design: a controlled experiment comparing UML activity diagrams and state machines," Software Quality Journal, vol. 27, no. 1, pp. 125-147, 2019.

[22] D. Gessenharter and M. Rauscher, "Code generation for UML 2 activity diagrams," in Modelling Foundations and Applications. Heidelberg: Springer, 2011, pp. 205-220.

[23] M. Hossein, A. Hemmat, O. A. Mohamed, and M. Boukadoum, "Towards code generation for ARM Cortex-M MCUs from SysML activity diagrams," in Proceedings of 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, Canada, 2016, pp. 970-973.

[24] S. Schupp, "Code generation for UML activity diagrams in real-time systems," Ph.D. dissertation, Technische Universität Hamburg, Germany, 2016

[25] 21. A. Jaffari, J. Lee, C. J. Yoo, and J. H. Jo, "Test case generation technique for IoT mobile application," in Proceedings of 2017 Spring KIPS Conference, Jeju, Korea, 2017, pp. 618-620.

[26] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 2011, pp. 416-419.

[27] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in UML'99 – The Unified Modeling Language. Heidelberg: Springer, 1999, pp. 416-429.

[28] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," International Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 410-425, 2000.

[29] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," IEEE Transactions on Software Engineering, vol. 14, no. 10, pp. 1483-1498, 1988.

[30] G. Denaro, M. Pezze, and M. Vivanti, "On the right objectives of data flow testing," in Proceedings of 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, OH, 2014, pp. 71-80.

[31] R. Anbunathan and A. Basu, "Dataflow test case generation from UML Class diagrams," in Proceedings of 2013 IEEE International Conference on Computational Intelligence and Computing Research, Enathi, India, 2013, pp. 1-9.

[32] L. Briand, Y. Labiche, and Q. Lin, "Improving the coverage criteria of UML state machines using data flow analysis," Software Testing, Verification and Reliability, vol. 20, no. 3, pp. 177-207, 2010.

[33] T. Waheed, M. Z. Z. Iqbal, and Z. I. Malik, "Data flow analysis of UML action semantics for executable models," in Model Driven Architecture-Foundations and Applications. Heidelberg: Springer, 2008, pp. 79-93.

[34] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta, "Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments," IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 161-187, 2010.

[35] P. C. Jorgensen, Software Testing: A Craftsman's Approach. Boca Raton, FL: CRC Press, 2014

[36] N. Li and J. Offutt, "Test oracle strategies for model-based testing," IEEE Transactions on Software Engineering, vol. 43, no. 4, pp. 372-395, 2016.

[37] J. Badlaney, R. Ghatol, and R. Jadhwani, "An introduction to data-flow testing," North Carolina State University, Technical Report No. TR-2006-22, 2006.

[38] R. Malhotra, Empirical Research in Software Engineering: Concepts, Analysis, and Applications. Boca Raton, FL: CRC Press, 2015.

[39] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in Software Engineering. New York, NY: Springer, 2012.

[40] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 721-734, 2002.

[41] Software-artifact Infrastructure Repository, https://sir.csc. ncsu.edu/portal/index.php.

[42] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, 2005, pp. 402-411.

[43] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults," Empirical Software Engineering, vol. 23, no. 4, pp. 2426-2463, 2018.

[44] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: finding real faults in a financial application," in Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, Buenos Aires, Argentina, 2017, pp. 263-272.

[45] G. Fraser and A. Arcuri, "EvoSuite at the SBST 2016 tool competition," in Proceedings of 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST), Austin, TX, 2016, pp. 33-36. IEEE.

[46] G. Fraser, J. M. Rojas, J. Campos, and A. Arcuri, "EvoSuite at the SBST 2017 tool competition," in Proceedings of 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), Buenos Aires, Argentina, 2017, pp. 39-42.

[47] R. Lowry, "Concepts and Applications of Inferential Statistics," Vassar College, Poughkeepsie, NY, 2011

[48] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback directed random test generation," in Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, 2007, pp. 75-84.

