Regular Paper
Journal of Computing Science and Engineering,
Vol. 13, No. 3, September 2019, pp. 107-123

An Experimental Investigation into Data Flow Annotated-Activity Diagram-Based Testing

Aman Jaffari and Cheol-Jung Yoo*
Department of Software Engineering, Chonbuk National University, Jeonju, Korea
{aman, cjyoo}@jbnu.ac.kr

Abstract
With the acceptance of Unified Modeling Language (UML) as the de-facto standard for modeling software systems, many research studies have addressed the necessity for utilizing models of systems under testing as inputs for test automation. Recently, activity diagrams have been used as a basis to derive test cases. Current studies have focused on analyzing the control flow of activities. However, examining the control flow among activities is quite simple and such testing on its own is insufficient. This study proposes technique for test case generation that complements an activity diagram with data flow information. To investigate the potential benefits of this technique, we performed an experimental investigation of well-known systems in testing literature. The experimental results were analyzed and compared with a state-of-the-art test suite generation tool as an alternative approach to fault detection effectiveness and efficiency. Overall, the results indicate that the proposed technique outperforms the alternative approach by detecting 27.3% more faults on average. In particular, the proposed technique yielded the best results in detecting faults related to arithmetic operations or parts used for calculation in our context.

Category: Software Engineering

Keywords: Model-based testing; Activity diagram-based testing; Data flow-annotated activity diagram; Data flow information

I. INTRODUCTION

Testing constitutes a significant proportion of software development efforts. In particular, test design is the most challenging and time-consuming part of testing. To test software, testers need to develop cases to demonstrate the presence of defects. Designing appropriate test cases is the key factor revealing the extent of defects. Along with the adaptation of Unified Modeling Language (UML) diagrams as the de-facto standard for modeling software systems, it has become necessary to utilize models of the system under test (SUT) as inputs for test models [1-3]. Accordingly, the research community has shifted its focus on design and development of test cases based on different structural and behavioral models. In particular, researchers have placed more emphasis on investigating the generation 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 types of diagrams [17].

Towards this end, the AD has been regarded as an important design artifact to identify test cases [9]. Currently, the main focus of existing studies is on test automation based on analysis of an AD to gather various

Open Access http://dx.doi.org/10.5626/JCSE.2019.13.3.107 http://jcse.kiise.org

Received 09 July 2019; **Accepted** 16 September 2019
*Corresponding Author

types of control flow information. However, examining the control flow among design elements is quite simple and straightforward [18, 19]. Testing only based on the sequence of activities in an AD is probably not sufficient for fault detection. Thus, finding ways to improve the test quality based on design elements such as an AD is an important ongoing issue. An activity represents system behavior to ensure its accuracy, besides automatically analyzing its control flow. These activities also need further analysis in terms of data flow information.

ADs are used to model system behaviors and the ways in which these behaviors interact with each other by defining the sequence of actions among them. Actions are regarded as the main capabilities of activities and are central to the data flow of activities [2, 17]. Empirical studies in the literature indicate that AD is the most comprehensive [20] and suitable design artifact among the behavioral models, defining the control flow among objects in an object-oriented system [9]. Moreover, an AD is regarded as the best intermediate model between the software specification and the code that provides a rich source of information for data flow analysis. Thus, ADs are investigated further for other purposes such as automatic code generation [21-23].

In this study, we have performed an experimental investigation on an AD-based test case generation technique including data flow information (DFI). This study extends our previous work [24] that was presented at a conference. The AD of a SUT is annotated with DFI to facilitate data flow coverage (DFC) criteria. The inclusion of DFI, instead of the AD of an SUT without DFI, facilitates the analysis of the control flow information, enables detection of definition-use pairs of object variables across activities that support the generation of highly improved test cases. For example, it can be used to detect data flow anomalies that can also validate the model itself before extraction and execution of test cases and faults that require more precise oracles such as calculation. After annotating the AD with DFI, it is mapped to an intermediate model, a so-called data flow graph (DFG), which is comparatively simple and more appropriate for manipulation by automated means. Subsequently, the test paths are generated from DFG in a depth-first search (DFS) manner using specific DFC criteria, and the concrete tests are executed with the oracles and input values provided. Finally, the results are presented and evaluated against the expected outputs.

The experimental investigation was carried out with the commonly used software systems for comparing and assessing both new and existing tools and techniques in testing literature. A comparative analysis of the experimental investigation that employed two techniques, namely data flow annotated activity diagram-based testing (DFAAD), and a so-called state-of-the-art test suites generation tool (EvoSuite) [25], was performed. The impact of the proposed approach on fault detection effectiveness and efficiency was discussed. In the context of this study, effectiveness means revealing the maximum number of faults without considering the number of executed tests, whereas efficiency means revealing the maximum number of faults with the minimum number of executed tests. The experimental investigation was performed to further validate the claims regarding fault detection effectiveness and efficiency. Thus, we aimed to answer the general research question (RQ): How does the proposed DFAAD-based testing technique perform compared to a well-practiced alternative in terms of fault detection effectiveness? In Section 4, our general RQ was further divided into sub-questions.

The rest of the study is organized as follows: Section 2 discusses related work and compares current AD-based test case generation techniques. Section 3 presents the main approach including the basic concepts and definitions, running examples to illustrate the overall concepts, a simple description for identifying and annotating DFI, and also the steps for extracting the DFG from the DFAAD. An experimental study to investigate the potential benefits of this technique against an alternative testing technique is provided in Section 4. Sections 5 and 6 discuss the findings and threats to validity, respectively. Finally, Section 7 provides the conclusion and future study directions.

II. RELATED WORK

This section discusses related work and provides a simple comparison of current studies in the literature involving AD-based test case generation techniques. ADs are used to model the dynamic behavior of the SUT and are widely practiced to support testing. These models are very useful and provide a significant testing opportunity because they precisely describe the functionality of the SUT in a way that can be easily manipulated via automation [26]. ADs can be used to model a system from a high-level business process to each individual unit of the system, as well as the internal logic of a complex unit. In general, there are a number of benefits associated with AD-based testing, such as generating test cases early during software design, better documentation of test cases, early discovery of specification errors, and reduction of the test cost and effort.

There are several studies in the literature with different strategies that have used ADs for test case generation. An AD-based test case generation approach was introduced [14] that uses UML 2.0 with a use case scope based on an AD from a high-level of abstraction. The purpose of the study is to detect greater synchronization and additional loop faults following activity path coverage criteria. An AD-based testing method [11] constructs condition-classification trees by collecting control flow information from decision points and guard conditions. The information is stored in a table and each row of the table represents a test case.

Another technique [10] known as coverage-driven automatic test generation considers both the control flow and data flow by parsing the AD. The study extracts and analyzes the structure of the AD by collecting data members to provide input for a new symbolic model checker (NUSMV). However, it is not clear how the data members are mapped to the NUSMV [27] input and their contribution to test case generation. The test case generation technique using an AD based on a gray box method [5] generates test cases directly from the AD. The study combined the white box and the black box methods. To identify any inconsistency between the implementation and the design, a category-partition method was used to generate the possible values of the input/output parameters. This technique is similar to our study in that it generates test cases that can be used to test the system at code level. However, the technique still focuses on the control flow of operations/method sequences in an AD and applies the basic path coverage criteria. In our study, we explicitly annotated the AD of the SUT with data flow information and used data flow coverage criteria. To improve the test effectiveness, a rule-based approach [4] was presented to derive a combinatorial test design model from ADs. The main idea is to provide rules for identifying the parameters with their corresponding values and constraints by parsing the AD.

For further information on AD-based testing, Table 1 provides a simple comparison of the several perspectives such as the technique used, the existing work. The current studies are compared from research objective, the intermediate model used to generate test cases, and the coverage criteria employed, as well as issues related to these techniques based on our perception. The objective

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

Study	Techniques	Objective	Intermediate model	Coverage criteria	Related issues
[14]	UML 2.0 modeling capabilities with use case scope	Detecting synchronization and loop faults	Activity graph	Activity path	High level of abstraction, missing details of individual activity
[11]	Condition classification tree	Test automation, generate tests early during development	Condition classification tree	Decision point	Difficulties in identifying all feasible paths with complex control flow and their nested combinations (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 states will not access the same object and only execute asynchronously
[6]	XML-based	Test automation, save time and effort	Activity dependency table (ADT)	Branch, predicate, basic path	Lacking validation of fault detection capability with reduced set of generated test paths
[4]	Combinatorial test design model	Test automation, reduce effort, improve effectiveness	CTDM model	Parameter-value	Difficulties on identifying constraints from activity diagram, and constraints 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]	Classification of control constructs	Identification of all possible scenarios	Intermediate 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 test 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 automation, 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] grammar	All paths	Manually generating AC grammar and feasible paths with complex AD including loops, detecting only design errors

of the comparative analysis is to provide a general overview of the current status of AD-based testing. Although we have highlighted several related issues of the current AD-based testing techniques, we have not addressed each and every issue.

The results presented in Table 1 lead to the conclusion that existing approaches primarily focused on test automation based on analysis of various items of control information in an AD. However, examining only the control flow among the activities is not sufficient for testing the whole SUT. Although test automation based on analyzing the control flow information is a good idea for improving test efficiency, test quality is more important. In other words, the generated test cases should be capable of detecting faults. Accordingly, improved fault detection capability of the conventional AD-based testing is an ongoing challenge. The experimental investigation in this study improved the coverage criteria for increased fault detection capability by annotating ADs with DFI. Initially, data flow-based testing techniques were introduced to complement conventional control flow-based testing techniques [28, 29].

Some studies also investigated the incorporation of DFI in a model-based testing environment, for example using UML class diagrams [30] and state machine diagrams [19, 31, 32]. However, class diagrams are limited to the static view of SUT and miss dynamic behavior. Also, state machine diagrams are limited to representing the interaction between complex objects and do not represent all of the properties of the SUT [33]. In contrast to existing studies, this study investigates how annotating DFI can improve fault detection capability in the context of an AD.

III. DATA FLOW ANNOTATED ACTIVITY DIAGRAM-BASED TESTING

In this section, we describe the basic concepts and definitions used in this study and provide an overview of DFAAD-based testing. We also describe each of the activities in further detail such as the circumstances for identifying and annotating DFI and the steps for extracting a DFG based on an annotated AD, along with a running example.

A. Basic Concepts and Definitions

Most of the definitions and terminologies used in this study are derived from standard software testing textbooks [18, 34], existing documentation, and research studies with slight modification [2, 9, 10].

Definition 1. Data flow-annotated activity diagram. DFAAD is an extension of the original activity diagram representing the sequence of actions in which the flow of data is explicitly marked across activities. Similar to ADs, a DFAAD can be described formally as a graph, $G = (A, E, C)$ where:

- A is a set of actions/activities including A_0 and A_f, where each A except A_0 and A_f is annotated either with *d, u, cu* or a combination thereof following the name of the data members with stereotype notation in which *d* stands for defined, *u* stands for used, 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 $A_f \neq \emptyset$.
- E denotes a set E of edges, where E is a subset of $A \times A$
- $C = D_n \cup J_n \cup F_n \cup M_n$ is a set of control nodes such that 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 merge nodes. In terms of data flow, D_n is analogous to *p-use* that stands for predicated use.

We restrict both the AD and the data flow graph containing a single initial node.

Definition 2. Data flow graph. A DFG is a simplified representation of an annotated activity diagram (AAD) that can be formally defined as:

- a set N of nodes, where each node is explicitly marked with $DFI_{d,\ u,\ cu,\ \text{and}\ p\text{-use}}$
- 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 $N \times N$

Definition 3. All d-use path coverage (ADUPC). For each def-pair set S = d-use (n_i, n_j, v), TR contains every path d in S [18]. TR stands for test requirements, d-use for definition use, and n_i and n_j represent node i and node j, respectively.

B. Overview of DFAAD-based Testing

This section provides an overview of DFAAD-based testing and describes the major activities in detail. In model-based testing (MBT), a model (e.g., an AD, a state machine diagram) specifies the behaviors of a SUT. According to [1], a typical MBT activity includes three main tasks: design of a functional test model to represent the expected operational behavior of the SUT, determination of test generation criteria to limit the number of generated tests, and generation of tests that can be fully automated.

In this study, we followed a behavioral MBT in which the AD of the SUT was selected on a test basis. Similar to an MBT approach, the DFAAD-based testing includes three major activities, namely behavioral test model design, test generation, and test execution. In order to facilitate DFC criteria, the DFI was identified and annotated within the AD of the SUT. The AAD was mapped into an intermediate model known as DFG for simplification and further automation. The deliverable of this activity was

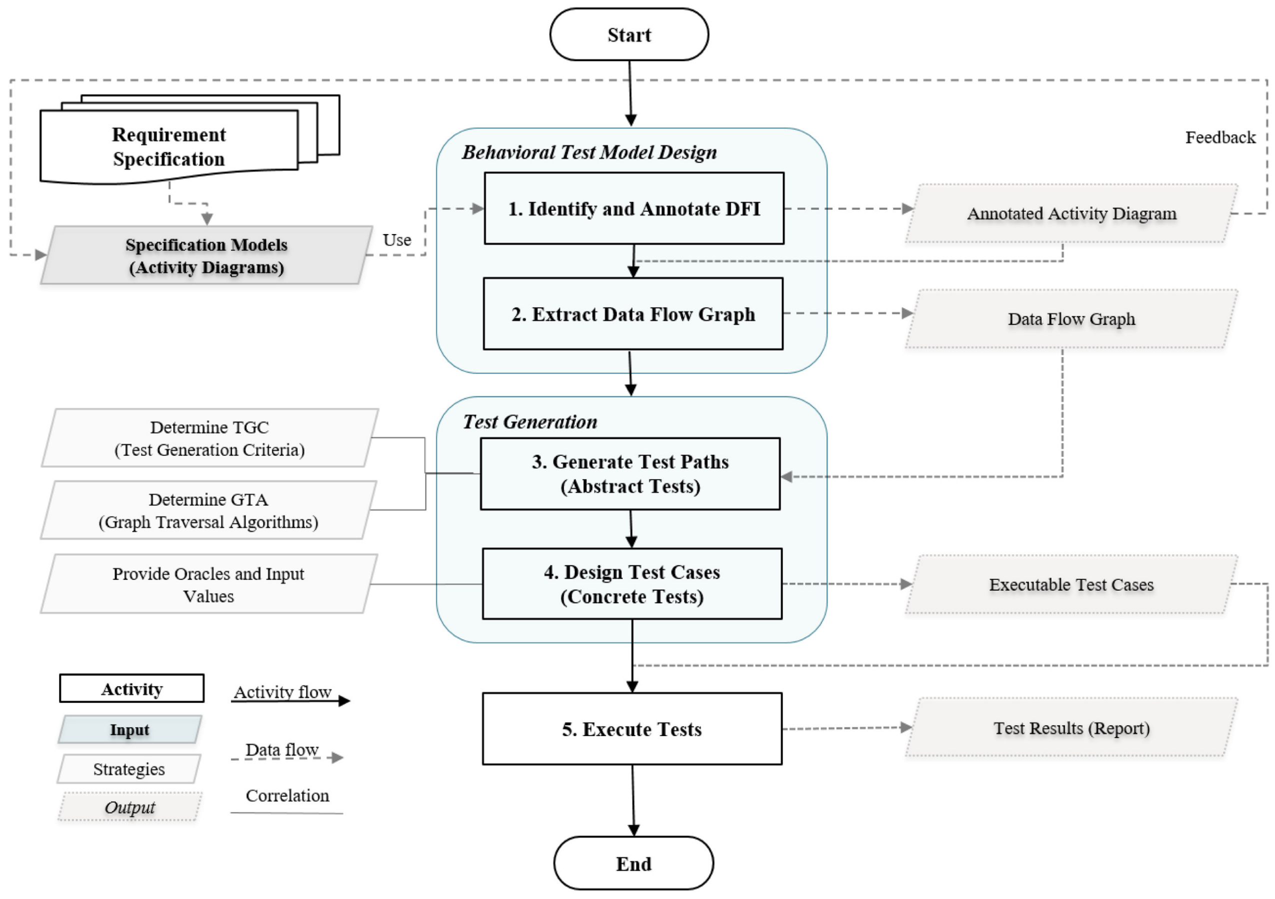
Start
Requirement Specification
Specification Models (Activity Diagrams)
Use
Feedback
Behavioral Test Model Design
1. Identify and Annotate DFI
2. Extract Data Flow Graph
Annotated Activity Diagram
Data Flow Graph
Test Generation
Determine TGC (Test Generation Criteria)
Determine GTA (Graph Traversal Algorithms)
Provide Oracles and Input Values
3. Generate Test Paths (Abstract Tests)
4. Design Test Cases (Concrete Tests)
Executable Test Cases
5. Execute Tests
Test Results (Report)
End
Activity
Input
Strategies
Output
Activity flow
Data flow
Correlation

Fig. 1. Overview of DFAAD-based testing.

an intermediate test model used to generate test paths based on existing graph traversal algorithms such as DFS or BFS. To generate the tests, testers need to determine the test generation criteria (for example, du-path coverage) to limit the number of generated test cases. In this study, we used d-use coverage for data flow testing. Next, in order to adapt the generated test cases, testers provided oracles and input values. Finally, the generated tests were executed and the test results were reported. An overview of DFAAD-based testing is shown in Fig. 1.

1) Identification and Annotation of DFI

This section presents the circumstances in which the DFI can be identified and annotated in an AD. The control flows among activities are depicted in an AD, and ADs can model a system from a high-level business process to interaction and state changes among activities, returning values, and computations. They represent the sequence of actions among activities. Actions are required for any significant capabilities and are central to the data flow aspect of an AD [2]. The sequence of actions among activities provides useful information about messages, such as the sender and the receiver object, state changes, exchanging parameters, returning values, and guard conditions. These are valuable references to identify DFI of an AD. Also, other supportive references include specification documents, in particular, the use case specification that is the foundation of an AD. The steps and circumstances in which the DFI can be identified and annotated with the AD are as follows:

(1) Identification of participating data members in an AD: Annotating an AD with DFI first requires identification of the participating data members in the AD. There are several possible ways to identify a data member across activities as mentioned above. The easiest way to identify a data member is to use the information in the guard condition. Also, the input or output parameters that are specified in the Action Pin assist the identification of the data members.

(2) 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. An action behavior is depicted by the descriptive name and types of the action. Therefore, one approach is to analyze the encapsulated behavior of each action. It is possible that the different DU pairs of

data members across activities can be detected in the following situations: (a) A define may occur in an A in the following situations: In an ExecutableNode that is the source ActivityNode that gets the variable objects by input (define by input). In an ExecutableNode in which the variable objects are initialized (defined by assignment). (b) A use may occur in an A in the following situations: In an ExecutableNode that is invoked by other behaviors and operations on input data, deliver or return the data to other activities through outgoing edges without modification. The data is received from the sender object and passed or returned to the receiver object without modification. (c) A c-use may occur in an A in the following conditions: In an ExecutableNode that executes a subordinate behavior (e.g. arithmetic computation, manipulation of object contents). (d) A p-use may occur in an A in the following situations: In a C node of an AD with D_n and a guard condition, C nodes without guard conditions being excluded (e.g. J_n, F_n, M_n).

(3) Annotation of the DFI in an AD. After the data members are identified and their DU pairs are detected, this information is annotated with the AD using stereotypes, for example, if a variable is defined in a particular activity it is represented as <<d(variable name)>>. Further information can be found in the running example and the case study sections.

2) Extraction of DFG

With reference to the UML documentation and existing studies, it is possible to easily associate the AD syntax with the DFG syntax. A DFG encapsulates AD constructs systematically for further automation [14]. We can extract test cases either directly from an AD or convert it into a DFG that is an intermediate model and generate test cases by traversing the graph. For the purpose of simplification, we prefer converting the AD into a DFG that simplifies the concepts by encapsulating various syntax of an AD into DFG nodes. The DU pairs that are annotated in an AD are labeled with the corresponding nodes of the DFG. Since, both the AD and DFG are directed graphs, mapping the annotated AD into the DFG is straightforward, and basically involves the following steps:

1. The set A of activities is mapped to the set N of DFG nodes ($A_{activity} \rightarrow N_{node}$), and the occurrences of d-use information are included with the corresponding nodes
2. The A_0 node of an AD is mapped to the N_0 node of a DFG ($A_0 \rightarrow N_0$)
3. The A_f node of an AD is mapped to the N_f of a DFG ($A_f \rightarrow N_f$)
4. The C_n nodes of an AD are mapped to the N of a DFG ($C_n \rightarrow N$)
5. The edge E of an AD is mapped to the edge E of a DFG

The graph theory is a long-standing practice in software testing that provides an important simplification mechanism for testers. An advantage of the AD to DFG conversion is that the DFG has a more simplified structure. Hence, generating test cases based on a DFG is comparatively easy. Also, in the case of a DFG, many algorithms already exist for traversing the graph to generate tests.

3) Generation of Test Paths

The test paths are generated based on the intermediate test model known as DFG. To limit the number of generated tests, it is necessary to determine appropriate test coverage criteria (TCC). The most commonly adopted coverage criteria include branch coverage, decision coverage, simple path coverage, and statement coverage that require the test cases to cover and execute every branch, decision, statement, or path. However, they are not capable of discovering many common faults [28]. Hence, the DFC is introduced to supplement the control flow coverage criteria and the most well-known is the DU path coverage. In our approach, we applied ADUPC for the DFC criteria. The basic concepts and definitions are provided in Section III-1, and Section III-3 provides some examples. After determining the TCC, a graph traversal algorithm (GTA) is needed to extract the abstract tests based on the chosen criteria. The most common graph traversal algorithms include DFS and breadth-first search (BFS). In our case, we have traversed the DFG in a DFS manner. After generating the relevant test paths, these paths are used to design concrete tests by providing additional information such as oracles and input values.

4) 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 test generation activity that needs particular attention. Currently, many guidelines and test oracle strategies exist in the literature. A number of oracle strategies for model-based testing have been reported and their fault detection capability investigated [35]. Though the test oracle strategy is not our main concern, we followed the existing strategies and guidelines to design the concrete tests.

5) Execution of Tests

As the final activity, test cases were executed against the SUT, and the test results were reported. Currently, many test execution frameworks exist that can be used to execute designed test cases. In this study, we have used JUnit as an execution tool that facilitates the comparative analysis against the alternative approach, but any other tools can be used to execute the generated executable tests.

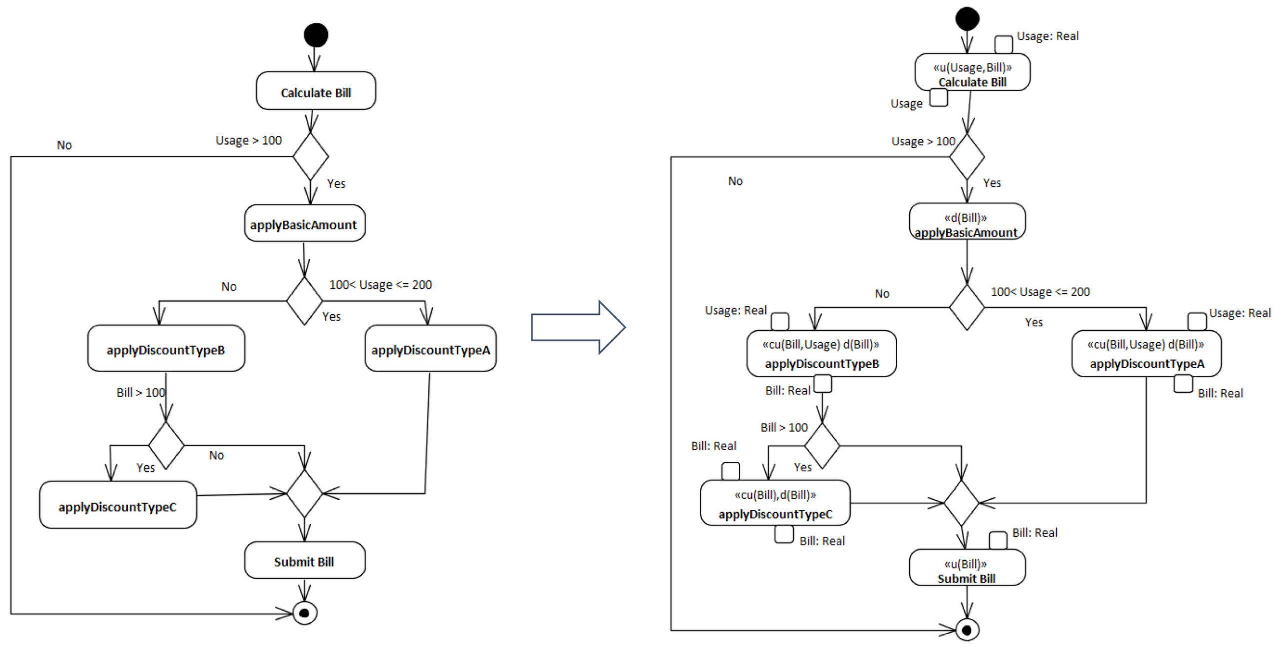

Fig. 2. (a) Customer billing service AD; (b) Customer billing service DFAAD.

C. Running Examples

For the ease of understanding, in this section, we provide two running examples using a customer billing service application and a triangle problem. The first example illustrates the annotation of data flow in an AD and the latter further exemplifies the generation of DFG and test cases.

1) Example 1

As a first running example, we have used a modified version of a cellular service customer billing application [36]. The application calculates billing based on the customer's usage amount and offers three different types of discount. The following is a simple description for "Calculate Billing Service" use case specification:

Normal flows:

1. The application obtains the usage amount from the actor of the system
2. If the usage is greater than zero, it calculates the bill based on the type of discount
3. If the usage is between 100 and 200, a discount type A is applied (50 cents for every additional minute)
4. If the usage is greater than 200, a discount type B is applied (10 cents for every additional minute)

Alternate/exceptional flows:

1.1 If the usage is less than or equal to zero, the bill is zero

4.1 If the bill is greater than 100, a discount type C is applied (10% discount from the total amount)

The use case specification is the foundation for generating an AD. It is also counted as a rich source of information for identifying DFI in an AD. The original AD for billing service is depicted in Fig. 2(a), and the corresponding AAD is shown in Fig. 2(b). The AD represents the sequence of actions that are required to calculate the billing amount. Actions are regarded as the main capabilities of activities and are central to the data flow aspects [2]. We can use Action Pins to represent data values passed out of and into an Action. For annotating the AD with DFI we apply the three steps defined previously. The first step is to identify the participating data members across activities using the information in the guard conditions and the in/out put parameters specified in the Action Pins. In Fig. 2(b), a total of two data members (Usage, Bill) were identified by analyzing the action pins and guard conditions. Subsequently, we detected the DU pairs of each identified data member across activities by analyzing the encapsulated behavior of Actions. In the final step, the AD was annotated with the identified DU pairs.

For better understanding, Table 2 illustrates the DU pairs associated with the data members across different activities in a simple format. Table 2 also shows abstract and concrete tests that include input values and expected outputs. In this example, the input and expected values in Table 2 represent only dummy values used for demonstration purposes. In this context, in this running example, we did not perform any mapping between the AD and DFG. The abstract test cases were directly extracted by manually traversing the DFAAD. Identifying the DU pairs across activities also facilitates the detection of data

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

Variable	Activity nodes									
	Calculate Bill	Decision (node1)	Basic Amount	Decision (node2)	Discount TypeA	Discount TypeB	Decision (node3)	Discount TypeC	Merge node	Submit Bill
Usage	d	pu		pu	cu	cu				
Bill	d		d		cu, d	cu, d	pu	cu, d		u

	Abstract tests	Concrete tests	
		Input value	Expected output
Usage du-pairs	CalculateBill → Decision node1 → Decision node2 → DiscountTypeA	199	89.5
	CalculateBill → Decision node1 → Decision node2 → DiscountTypeB	900	99.0
Bill du-pairs	applyBasicAmount → DiscountTypeA	150	140
	applyBasicAmount → DiscountTypeB	230	110
	DiscountTypeA → SubmitBill	120	120
	DiscountTypeB → Decision node3 → SubmitBill	600	400
	DiscountTypeB → Decision → DiscountTypeC → SubmitBill	200	120
	DiscountTypeC → SubmitBill	300	150

flow anomalies. Based on Table 2, we can see that the Bill variable appears as define-define, which is a double definition that is considered as a potential bug. Detection of such anomalies in the model helps prevention of similar anomalies in the code.

2) Example 2

This example further illustrates the data flow annotation, the conversion of DFAAD to DFG using triangle problem.

Triangle problem specification: the triangle problem takes three positive whole-numbers as input. The program responds with a description of the triangle as follows:

- If all three sides have equal length, classify it as an equilateral triangle;
- If two sides have equal length, classify it as an isosceles triangle;
- If one angle is a right angle, classify as a right-angled triangle;
- If all sides have different lengths, and no right angles, classify it as a scalene triangle;

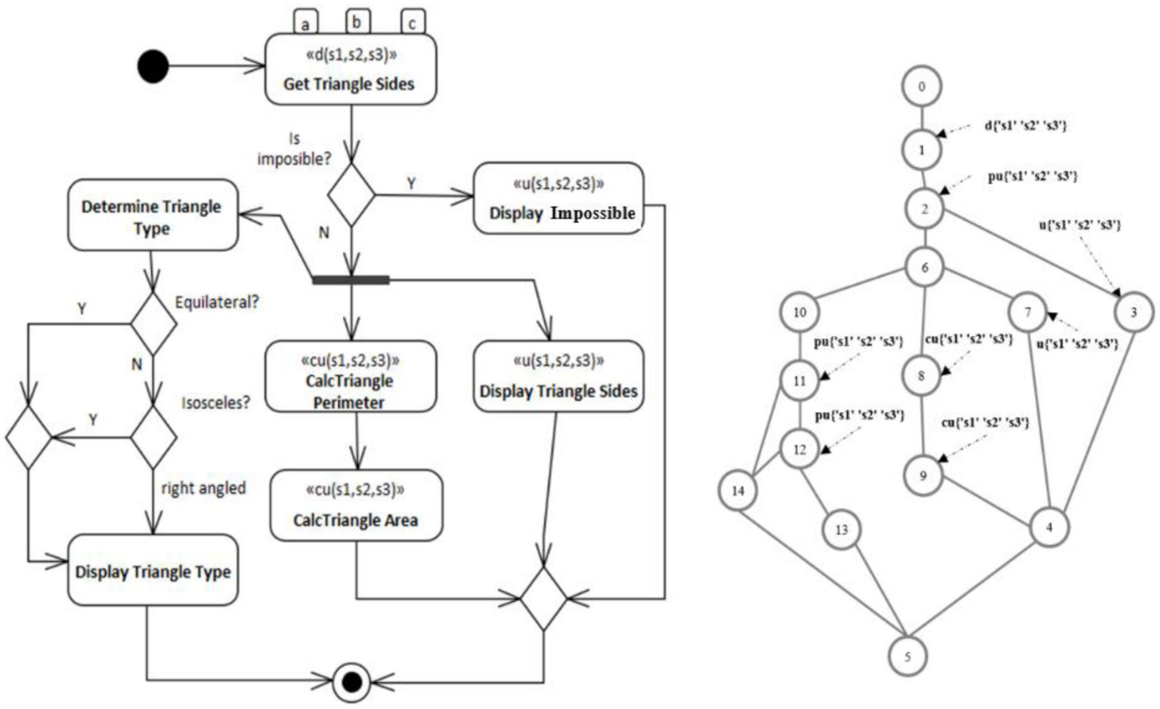

Fig. 3. a. Triangle problem DFAAD, b. Triangle problem DFG.

Table 3. Concrete tests and execution results for the triangle problem

Coverage	TC ID	Test paths	Input values			Expected outputs	Results
			s1	s2	s3		
ADUPC	TC1	1-2-3 - pu (s1,s2,s3)	20	0	10	Impossible	P
	TC2	1-2-6-7 u (s1,s2,s3)	40	50	20	40,50,20	P
	TC3	1-2-6-8 cu (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	113137.085	F
	TC5	1-2-6-10-11-12 pu (s1,s2,s3)	200	200	200	Equilateral	P
	TC6	1-2-6-10-11-14-5 pu (s1,s2,s3)	13	55	55	Isosceles	F
	TC7	1-2-6-10-11-12-13-5 pu (s1,s2,s3)	12	21	33	Impossible	F

- If the given sides do not form a triangle, classify it as impossible;
- Calculate the triangle perimeter and area.

The DFAAD of the triangle problem is depicted in Fig. 3(a), and the corresponding data flow graph extracted from the DFAAD is shown in Fig. 3(b). The abstract test cases generated from the DFG in Fig. 3(b) are presented on the left in Table 3.

IV. EXPERIMENTAL DESCRIPTION

This section describes the experiment carried out to evaluate the proposed technique, following existing guidelines for empirical research and experimentation in software engineering [37-39]. Section IV- A presents the experimental definition and context. Section IV- B describes the experimental procedure and design, and Section IV- C presents the experimental results.

A. Experimental Definition and Context

The experimental investigation was performed to compare and assess the proposed DFAAD-based testing technique with an alternative approach. Therefore, the experiment was focused on examining the potential effectiveness of the proposed technique to reveal faults compared with an existing well-practiced test case generation technique regarding the formulated research questions. Hence, we performed our experimental investigation involving three subject systems, namely the Cruise control system, Elevator system, and Coffee maker. To ensure that the experimental subjects were realistic (e.g., in terms of size and complexity) and suitable for our experiment, they were selected manually based on the following criteria.

- Large and logically complex systems containing a minimum of four classes, 60–80 branches, and 150–170 non-commenting statements.
- Systems comprising all the required artifacts available (e.g., class diagrams and a high-level description of the system's functionalities) for modeling the system behavior.
- Systems not excessively large or complex that prevent experimentation within the time constraint or inappropriate for alternative approaches to generate tests.

Table 4 summarizes the full description of the three experimental subject systems. All the required artifacts, including the source code of the systems, are available in the Software-Artifact Infrastructure Repository (SIR) for Elevator and Cruise control systems (https://sir.csc.ncsu.edu/content/sir.php) [40] and NCSU website (https://www.ncsu.edu/) for the Coffee
maker.

The experiments address the following RQ and sub-RQs that expand our general RQ given in Section I.

RQ1.1: How does the proposed DFAAD-based test case generation technique perform compared with an alternative approach in terms of overall fault detection effectiveness?

RQ1.2: What is the difference in effectiveness between the proposed DFAAD-based test case generation technique and the alternative approach regarding the type of faults detected?

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

Table 4. Description of experimental subject systems

Systems	#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 investigates whether the proposed technique can reach a certain level of effectiveness in terms of fault detection that is comparable to a well-practiced test suite generation approach like EvoSuite.

RQ1.2 further examines whether the proposed technique was more or less effective in detecting distinct faults compared with the alternative approach. For example, a positive response to this question will indicate the detectability of certain types of fault using the DFAAD-based approach, which is not detectable with the alternative one. RQ2 investigates if the test suites generated by one technique were capable of detecting additional faults with fewer tests than others.

Variable selection: The independent variable for all RQs is the type of technique used as a basis for test suite generation (e.g., DFAAD or the alternative technique). The dependent variables are related to effectiveness and efficiencies of fault detection, such as killed or survived mutants, and various fault types.

Mutation seeding: 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 automatic mutation testing tool, which works fast at the bytecode level (http://pitest.org). A potential benefit of automatic mutation testing is generation of a large number of mutants, which can increase the statistical significance of results obtained [41]. An empirical investigation of the effectiveness of mutation testing tools [42], revealed the outperformance of PIT_{EV} (evaluation version) compared with other tools. PIT provides three levels of mutator preferences (default, stronger, and all mutators). In our experiment we used the default mutators: increments mutator (IM), void method call mutator (VMCM), return value mutator (RVM), math mutator (MM), negate conditional mutator (NCM), invert negative mutator (INM), and conditional boundary mutator (CBM).

B. Experimental Procedure and Design

The focus of this experiment is to compare and assess the proposed technique via an alternative approach, in terms of fault detection effectiveness and efficiency. Accordingly, in the context of this study, we defined the terms efficiency and effectiveness as follows:

Effectiveness (*E*): The goal of effectiveness is to detect the maximum number of faults/killed mutants seeded in a program. It is measured based on the ratio between the number of detected faults/killed mutants per technique and the total number of existing faults/seeded mutants. In this study, the terms ‘mutants killed’ and ‘faults detected’ are used interchangeably.

$$\text{E} = \frac{\#\ \textit{of killed mutants}}{\textit{Total numbers of seeded mutants}} \times 100 \qquad (1)$$

Efficiency (*EF*): The goal of efficiency is to detect the maximum number of faults with the minimum sets of test cases. It is measured based on the ratio of the number of faults detected/mutants killed per technique and the number of associated executable test cases. Hence, the efficiency indicates the number of detected faults per executed test on average.

$$\text{EF} = \frac{\#\ \textit{of detected faults}}{\#\ \textit{of generated executable test cases}} \qquad (2)$$

The experiment was carried out considering all the activities presented in Section III as follows: (1) Because the DFAAD models were not available, given the three systems with all the required artifacts, the student created the required ADs for each subject system using the Enterprise Architect modeling tool. (2) The DFI was identified and annotated within the ADs of the SUT by applying the three steps. (3) Based on the DFAADs, a data flow graph was extracted and abstract test cases were generated. (4) A concrete test was designed by providing oracles and input values. Finally, the test was executed against the SUT and the results were reported.

As described in Section III, the DFAADs contain the sequences of actions, guard conditions, forks, joins, data members, and input parameters specified in the Action Pin. The data flow is explicitly annotated across activities. Figs. 4 and 5 demonstrate examples of such DFAADs for the car simulator and coffee maker activities, respectively. Depending on the nature of the SUT, DFAADs can be very complex or very simple. For example, a model of a running car simulation algorithm (Fig. 4) is quite complex compared to adding, deleting, or editing recipes in a coffee maker. Such differences in the complexity of experimental subjects significantly affect the performance of the tools or technique employed, and represent good examples for assessing and comparing their effectiveness.

Alternative approach: As an alternative approach for comparison and assessment, we selected EvoSuite[4] a so-called state-of-the-art test suite generation system [25, 42]. Generating test cases with EvoSuite is a simple task, which is carried out by right-clicking and pressing generate tests with EvoSuite. We selected EvoSuite because it has been practiced widely across different types and sizes of open-source as well as industrial software programs, reporting several real faults [43]. Also, EvoSuite attained the highest overall scores in the SBST 2016 and 2017 tool competition [44, 45]. Moreover, EvoSuite provides support for PIT mutation coverage.

C. Experimental Results

This section presents results obtained with the three experimental subjects.

RQ1.1: How does the proposed DFAAD-based test case generation technique perform compared with an

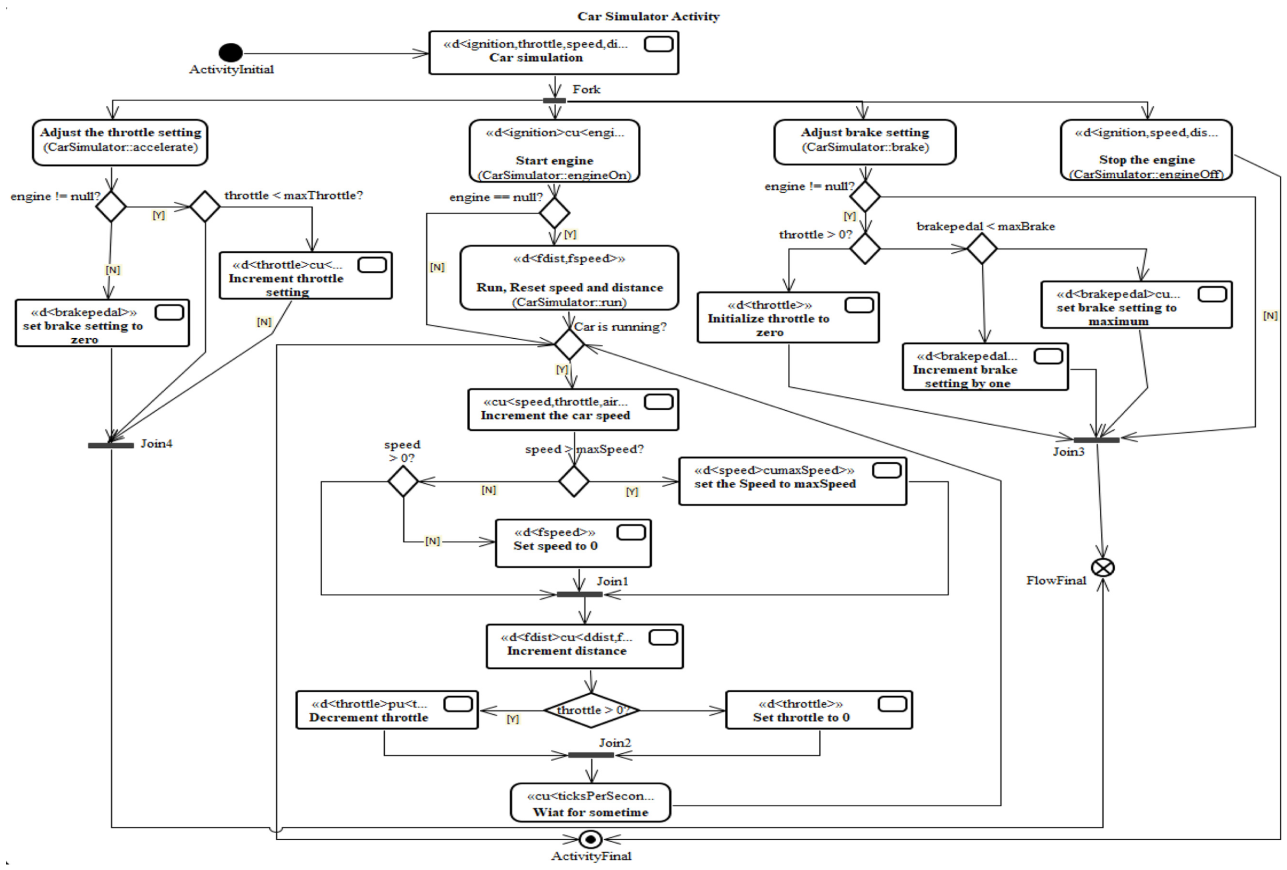

Fig. 4. An example of car simulator DFAAD for the Cruise control system.

alternative approach in terms of overall fault detection effectiveness?

RQ1.1 investigates the capability of the two approaches with regard to overall fault detection.

Fig. 6 presents our results of overall comparison of the two approaches. The figure depicts the number of faults detected or failed-to-detect by different techniques and their corresponding effectiveness scores (in percentages). As can be seen from Fig. 6, test cases generated from DFAAD was successfully applied to all experimental subjects reaching an overall effectiveness of 67.9% in the case of Cruise control, 69.6% in the case of Elevator, and 84.6% in the case of Coffee maker. However, test cases derived from the alternative approach managed to reach 44% effectiveness in the case of Cruise control, 24.3% in the case of Elevator, and 83.8% in the case of Coffee maker. The results indicated an overall difference in effectiveness of 23.9% in the case of Cruise control system and 45.3% in the case of Elevator. By contrast, both approaches attained a similar level of effectiveness in the case of Coffee maker. The observed difference in effectiveness was significantly large in the case of Elevator system. The factors contributing to these differences are further explained in Discussion.

The results in Fig. 6 indicate the improved fault detection capability of DFAAD compared with the alternative approach. To determine whether or not the observed difference was statistically significant, we performed a non-parametric Wilcoxon signed rank test [46] with a statistical significance level less than 0.05. The null and alternative hypotheses were as follows:

H_0: no difference between the proportion of faults detected by DFAAD and alternative approaches;

H_1: a difference between the proportion of faults detected by DFAAD and alternative approaches. As a result, the Wilcoxon signed rank test indicated that the DFAAD (mean rank = 7.36) was rated more effective than the alternative (mean rank = 5.0) with p-value = 0.013 and z-score = -2.488. Thus, we rejected the H_0 null hypothesis and concluded that the observed difference was statistically significant.

RQ1.2: What is the difference in effectiveness between the proposed technique for DFAAD-based test cases generation and the alternative approach regarding the type of faults detected?

This question evaluated the effectiveness in terms of the fault types detected by each technique. Tables 5–7 present the respective results for each experimental subject and the types of fault detected. Each table presents the

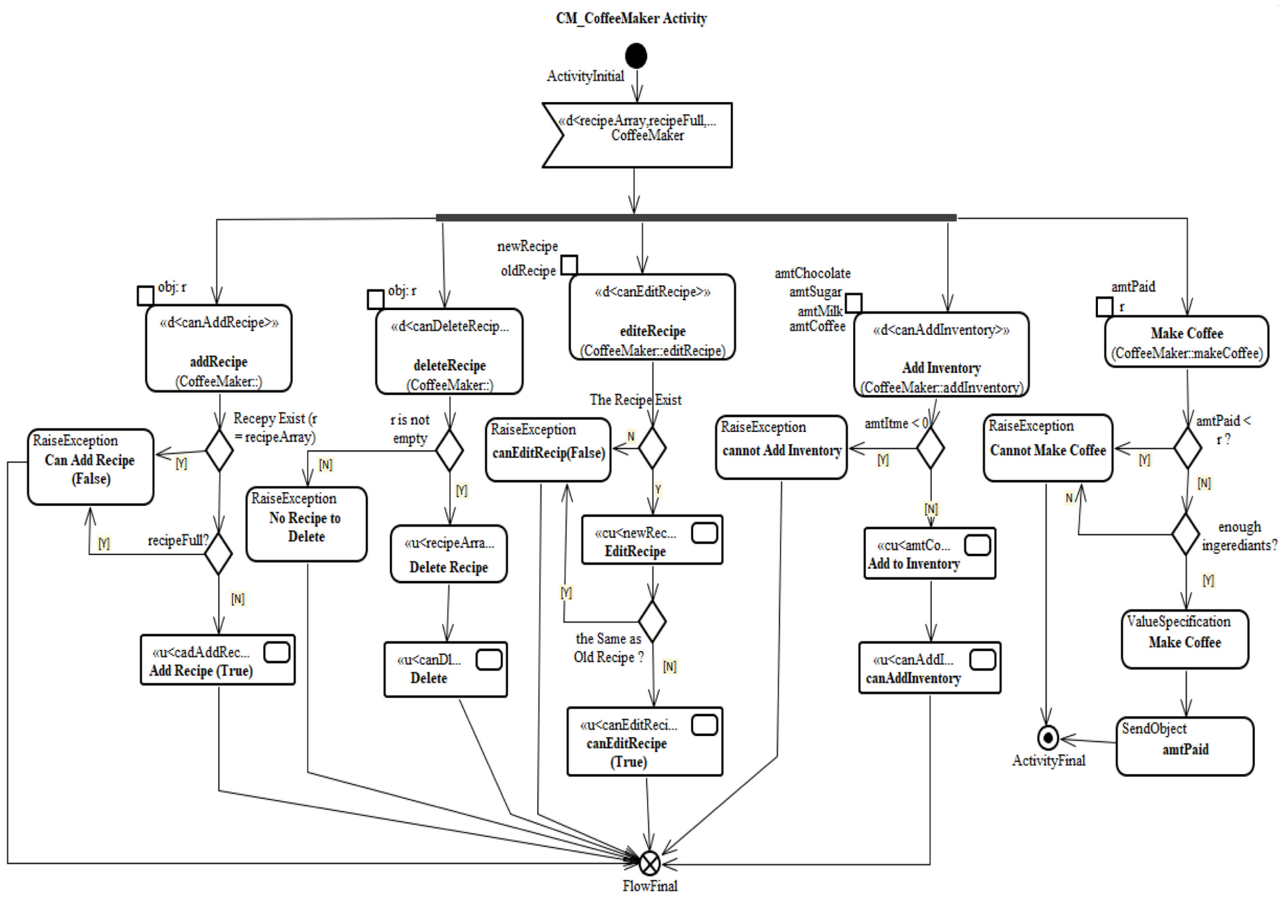
CM_CoffeeMaker Activity
ActivityInitial
«d<recipeArray,recipeFull,... CoffeeMaker
obj: r
«d<canAddRecipe>» addRecipe (CoffeeMaker::)
«d<canDeleteRecip... deleteRecipe (CoffeeMaker::)
newRecipe
oldRecipe
«d<canEditRecipe>» editeRecipe (CoffeeMaker::editRecipe)
amtChocolate
amtSugar
amtMilk
amtCoffee
«d<canAddInventory>» Add Inventory (CoffeeMaker::addInventory)
amtPaid
Make Coffee (CoffeeMaker::makeCoffee)
RaiseException Can Add Recipe (False)
Recepy Exist (r = recipeArray)
r is not empty
RaiseException canEditRecip(False)
The Recipe Exist
RaiseException cannot Add Inventory
amtItme < 0
RaiseException Cannot Make Coffee
amtPaid < r?
RaiseException No Recipe to Delete
«u<recipeArra... Delete Recipe
«cu<newRec... EditRecipe
«cu<amtCo... Add to Inventory
enough ingerediants?
recipeFull?
the Same as Old Recipe ?
ValueSpecification Make Coffee
«u<cadAddRec... Add Recipe (True)
«u<canDl... Delete
«u<canEditReci... canEditRecipe (True)
«u<canAddI... canAddInventory
ActivityFinal
SendObject amtPaid
FlowFinal

Fig. 5. An example of making coffee DFAAD for the Coffee maker system.

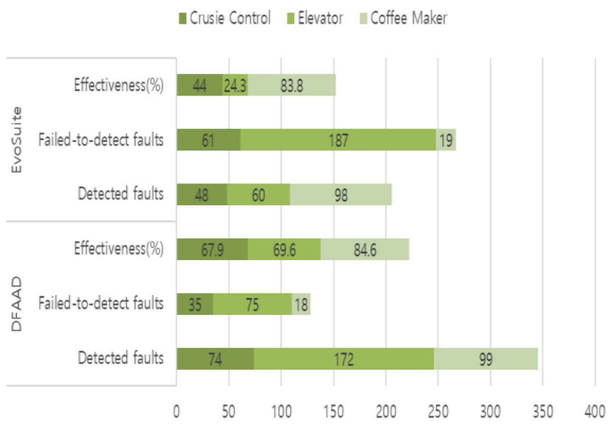

Fig. 6. Overall comparison of the two approaches in terms of fault detection effectiveness.

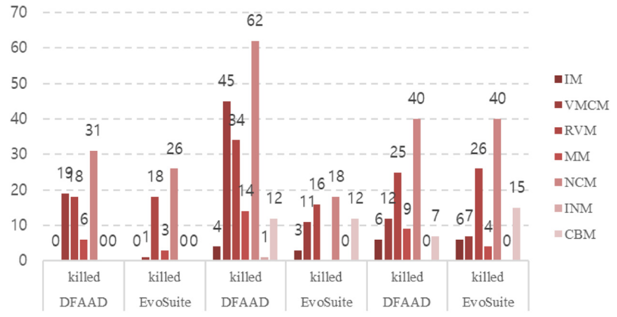

Fig. 7. Comparative analysis of killed mutants across subject systems.

total number of seeded mutants related to each mutation operator, the number of mutants killed and survived, and the corresponding rates of effectiveness. The most remarkable results are bolded and are further discussed in Discussion.

Fig. 7 further visually summarizes the types of mutants killed by each technique across subject systems. As can be easily observed from the results, DFAAD managed to kill more mutants in almost all types of mutation operators across subject systems except CBM in case of Coffee maker, which is further discussed in the following section.

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

The answer to RQ2 provides insights into the relative

Table 5. Cross-comparison of detected fault types with 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

The most remarkable results are bolded.

Table 6. Cross-comparison of detected fault types with Elevator system

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

The most remarkable results are bolded.

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**

The most remarkable results are bolded.

efficiency of the techniques in terms of the numbers of killed mutations per executed tests. We found that DFAAD killed more mutants with fewer executable sets of tests. The average numbers of killed mutants per executed test was: 2.6 mutants against 0.6 in the case of Elevator, 2 mutants against 0.7 in the case of Cruise control, 3.7 mutants against 1.1 in the case of Coffee maker, and finally 2.6 mutants against 0.8 in the overall subject systems. Fig. 8 summarizes the cross-comparison of the efficiency of two test case generation techniques.

V. DISCUSSION

This section discusses the impact of DFAAD-based testing on fault detection effectiveness and efficiency. The experimental results are used to compare and assess the proposed technique. In order to better understand and summarize the experimental results, we have formulated a number of RQs and modified our discussion accordingly, as follows:

Discussion of RQ1.1: To improve the validity of the

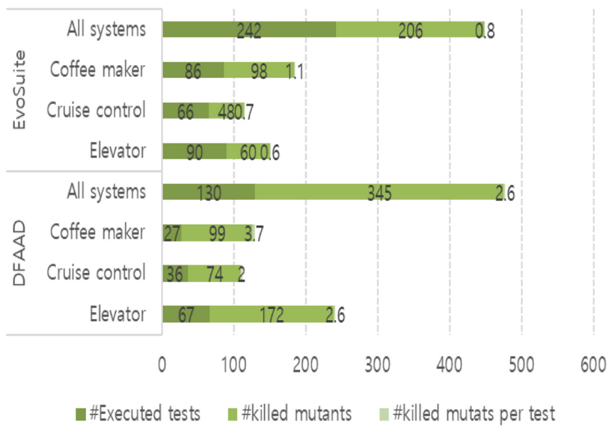

Fig. 8. Summary of efficiency results based on cross-comparison.

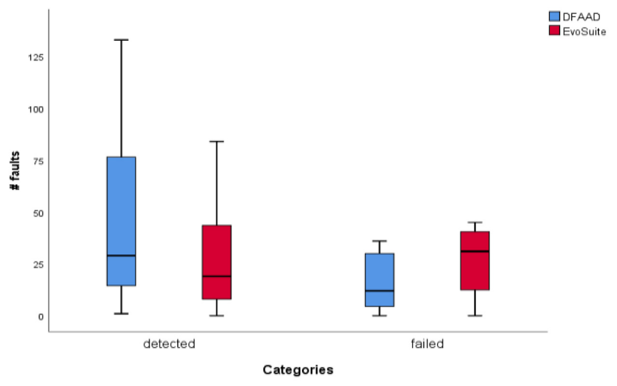

Fig. 9. Overall ratio of the distribution of detected and undetected faults across techniques.

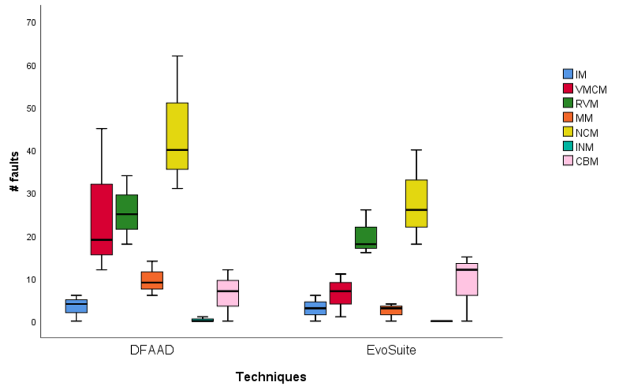

Fig. 10. Distribution of faults detection ratios across mutation operators and techniques.

conclusions for this question, we performed a statistical analysis of the extracted data. Due to the small sample size and non-normal distribution of the data, we used the Wilcoxon signed-rank test, a non-parametric test of statistical hypothesis with a statistical significance < 0.05. Overall, the formulated null hypothesis was rejected (*p*-value = 0.013) to conclude that the DFAAD managed to reach greater effectiveness.

Fig. 9 depicts the overall ratio of distribution of detected and undetected faults across DFAAD and EvoSuite regarding all subject systems. In contrast to EvoSuite, the DFAAD yielded a higher mean in the case of detected faults and was lower in the case of failed-to-detect cases. The DFAAD was found to be relatively effective in detecting faults across all the experimental subjects. However, the effectiveness of the alternative approach varied across different systems. For instance, the alternative approach resulted in better effectiveness in the case of Coffee maker, and performed relatively well in the case of Cruise control but very poor in the case of Elevator system. This result was probably attributed to the more complex and dynamic run time behavior of the system. For instance, EvoSuite is good at maximizing the branches and statements covered but is not adequately capable of handling the real-time properties of the SUT. However, in the case of DFAAD, the real-time behaviors of the systems are captured more appropriately to improve test cases.

Discussion of RQ1.2: This question investigated if the proposed technique was more or less effective in detecting different types of faults in contrast to the alternative. Fig. 10 shows the distribution of fault detection ratio across mutation operators and employed techniques. As can be seen from Fig. 10, in contrast to the alternative approach, on average, the DFAAD managed to detect a higher number of faults regarding all mutation operators except CBM. Particularly, the DFAAD achieved better coverage, on average, in detecting NCM mutants (mean difference +16.33), MM mutants (mean difference +7.33), RVM mutants (mean difference +6.66), and VMCM mutants (mean difference +19). However, the alternative approach showed greater effectiveness in detecting CBM mutants (mean difference +2.66). The difference is notable in the case of Coffee maker and is possibly due to the missing functionality while modeling the AD. It is possible that even if the system is less complex, the modeler may have applied less effort or attention, to model the system adequately. No notable difference was observed between the two approaches in detecting IM-related faults. Only a single INM mutant was generated by the mutation tool, in the case of Elevator, which is covered by DFAAD.

Discussion of RQ2: The efficiency or cost-effectiveness of test case generation techniques can be measured from different perspectives such as test execution and test generation. In published studies, the cost-effectiveness is measured according to various aspects, for example, based on the size of test drivers in terms of LOC, the CPU execution time, the number of methods calls [33], and the proportion of faults detected per distinct assertions created [35]. We assume that the efficiency of test case generation is proportional to the size of executable tests generated. Hence, the efficiency is estimated by detecting maximum number of faults with minimum tests. Our results indicated that test cases generated using DFAAD managed to detect 2.6 faults on average, whereas, test cases based on the alternative approach detected only 0.8 faults on average.

To summarize this discussion, the fault detection performance of EvoSuite varied from system to system, while DFAAD performance was almost constant across systems. The effectiveness of EvoSuite was quite poor in the case of Elevator system and relatively better in the case of Coffee maker, suggesting that the effectiveness of EvoSuite depends strongly on the nature of SUT. For instance, EvoSuite is not fully cable of testing systems with concurrent and dynamic run-time behaviors. By contrast, DFAAD proved to be best suited for such types of systems. Moreover, these results suggest that DFAAD facilitated the detection of additional faults related to arithmetic operations, which are regarded as more critical and difficult to detect. Despite the fact that EvoSuite is good at maximizing branches and coverage of statements, our results indicate that it still does not reflect test completeness. In addition to providing greater fault detection effectiveness, DFAAD offers the full advantages of model-based testing. However, we will not further discuss the pros and cons of model-based testing, or its comparison with other methodologies.

VI. THREATS TO VALIDITY

In this section, we discuss various possible threats to the validity of the experimental investigation performed and how they can be mitigated. Threats to *internal validity* relate to issues that impact the conclusions arrived at, for example, the use of the original source code without faults. Different conclusions were possible if systems with real faults were used. However, identifying appropriate systems with real experimental faults is not easy, and fault instrumentation is a common practice in testing studies. We used EvoSuite as an alternative approach for test suite generation and PIT mutation testing tool for fault seeding. Therefore, the effectiveness strongly depends on the characteristics of the employed tools. Hence, using different techniques for test suite generation or tools (e.g., manually generated, Randoop [47]), or using real faults or different mutation tools (e.g., MuJava) may yield different results. Also, in our experiment, we did not consider the impact of human subjects in generating test cases. For example, in DFAAD-based approach, individuals modeling the systems, and their role in modeling can significantly influence the results.

Threats to *external validity* relate to the generalizability of our study involving the experimental subjects and the types of fault used. Despite our efforts to provide valid results, we cannot be absolutely sure about the generalizability of the chosen subjects, because the results are always related to the SUT. We may have achieved different results by selecting different systems as our experimental subjects, for example, with a different domain, size, type, and complexity. To mitigate this threat, while choosing the subject systems, we considered minimal criteria involving the feasibility of the systems. Similarly, the tool used for mutant seeding and the types of mutants provided by such a tool may not be generalized to all cases. However, we ensured selection of well-established and popular tools, which are actively supported.

Threats to *construct validity* correspond to the generalizability and the appropriateness of the measures used in our experiments. To compare the fault detection capability of our technique with the alternative approach, we used the fault detection ratio, which is commonly used for the assessment of test techniques in studies with reliable outcomes [41]. Moreover, in our experiment, the ratio between the numbers of detected faults and the numbers of generated executable test cases might not reflect the actual test efficiency. For example, the results might have been different if we used alternative measurements such as the ratio of detected faults to distinct assertions created or the number of call methods.

VII. CONCLUSION AND FUTURE WORK

In this study, we have proposed a DFAAD-based test case generation technique. The AD of the SUT is annotated with DFI to facilitate DFC criteria and identification of the DU pairs of object variables across activities for the design of more appropriate test cases. To investigate the potential effectiveness of the proposed technique, we performed an experimental investigation using three commonly used systems in software testing literature. The effectiveness and efficiency of the proposed technique was evaluated by comparison with an alternative state-of-the-art test suite generation tool.

The statistical significance of the experimental results indicates that the proposed technique outperforms EvoSuite in terms of effectiveness. The results also showed that the proposed technique was comparatively more efficient in detecting critical faults (e.g., arithmetic operations related faults). It was quite remarkable that the results indicate that the proposed technique facilitated the detection of a wide range of faults, some of which are not detectable using an alternative approach, despite the full advantages of MBT.

In future, we have a plan to conduct comprehensive experimental studies with different applications involving additional variables such as time and cost efficiency. Also, we will develop supporting tools for our test case generation technique to facilitate automatic mapping of the AD with the associated DFG, and to enable automatic detection of DU pairs of object variables across 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, 2017.
20. 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.
21. D. Gessenharter and M. Rauscher, “Code generation for UML 2 activity diagrams,” in *Modelling Foundations and Applications*. Heidelberg: Springer, 2011, pp. 205-220.
22. 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.
23. S. Schupp, “Code generation for UML activity diagrams in real-time systems,” Ph.D. dissertation, Technische Universität Hamburg, Germany, 2016.
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.
25. 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.
26. J. Offutt and A. Abdurazik, “Generating tests from UML specifications,” in *UML’99 – The Unified Modeling Language*. Heidelberg: Springer, 1999, pp. 416-429.
27. 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.
28. 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.
29. 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.
30. 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.

31. 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.
32. 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.
33. 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.
34. P. C. Jorgensen, *Software Testing: A Craftsman's Approach*. Boca Raton, FL: CRC Press, 2014
35. 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.
36. J. Badlaney, R. Ghatol, and R. Jadhwani, "An introduction to data-flow testing," North Carolina State University, *Technical Report No. TR-2006-22*, 2006.
37. R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. Boca Raton, FL: CRC Press, 2015.
38. C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. New York, NY: Springer, 2012.
39. 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.
40. Software-artifact Infrastructure Repository, https://sir.csc.ncsu.edu/portal/index.php.
41. 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.
42. 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.
43. 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.
44. 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.
45. 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.
46. R. Lowry, "Concepts and Applications of Inferential Statistics," Vassar College, Poughkeepsie, NY, 2011.
47. 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.

Aman Jaffari

Aman Jaffari received the M.Sc. degree in Software Engineering from Chonbuk National University in 2015. He is currently a Ph.D. candidate in the Department of Software Engineering and a research fellow at Chonbuk National University. His research interests are software quality assurance, model-based testing, search-based software testing, etc.

Cheol-Jung Yoo

Cheol-Jung Yoo received B.S. degree in Computer Science and Statistics from Chonbuk National University, Jeonju, Korea in 1982, then, M.S. degree in Computer Science and Statistics from Chonnam National University, Gwangju, Korea in 1985, and then, Ph.D. degree in Computer Science and Statistics from Chonbuk National University, Jeonju, Korea in 1994. He is currently a professor in the Department of Software Engineering, Chonbuk National University, Jeonju, Korea. He is also the associate chair of the Korea Information Processing Society. He has been a visiting professor in EECS at the University of California Irvine (UCI) from January 2012 to July 2013. He has also been the chair of the Software Engineering Research Society of the Korea Information Processing Society from 2014 to 2015. His current research interests include software quality and testing, interoperability testing of embedded system and software, software complexity, big data analysis, etc. He is a life member of the Korean Institute of Information Scientists and Engineers and the Korea Information Processing Society.