Automatic Test Data Generation Using the Activity Diagram and Search-Based Technique

***Abstract*—In software testing, generating test data is quite expensive and time-consuming. The manual generation of an appropriately large set of test data to satisfy a specified coverage criterion carries a high cost and requires significant human effort. Using system models with search-based techniques to automate test data generation is quite cost-efficient and can reduce the human effort required. In this paper, we introduce a search-based automatic test data generation technique that uses the activity diagram of the system under test (SUT), focusing on its data flow aspect. We use a genetic algorithm as an optimization and heuristic search technique with a fitness function that rewards maximum definition-use pairs coverage. Our experimental investigation used three open-source software systems to assess and compare the proposed technique with two alternative approaches. The experimental results indicate the improved fault-detection performance of the proposed technique, which was 11.1% better than DFAAD and 38.4% better than EvoSuite, although the techniques did not differ significantly in terms of statement and branch coverage. An important variation between the applied techniques regarding the types of faults detected is that the proposed technique detected more computation-related faults than the other techniques. The results also show a relationship between the performance of the adopted techniques and the properties of the subjects such that the proposed technique tends to have better fault detection capability as the system complexity increases.**

Keywords—Automatic test data generation, activity diagram, genetic algorithm, model-based testing, search-based testing.

I. INTRODUCTION

Test case design is an important activity that consumes a large share of the budget and effort required for software testing. In particular, generating test data for executable test cases is a major challenge [1]. Accordingly, building a system to generate test data that maximizes fault detection effectiveness and minimizes the cost and effort is essential. The use of software models to automate the test generation process and reduce the testing cost and effort has been an active area of research for a long time [2-4]. Interest in model-based test automation has expanded along with the acceptance of Unified Modeling Language (UML) diagrams as the de facto standard for modeling software systems. Recent advances in model-based testing (MBT) have increased the feasibility and effectiveness of using MBT to automate or semi-automate the entire test generation process [2, 3].

In automated MBT, a model representing the expected behavior of the system under test (SUT) is developed and automatically analyzed to identify a set of test cases [5]. More recently, MBT practitioners have focused on using behavioral models of the SUT as a test basis to automate the test generation process [6]. Among the behavioral models, the activity diagram (AD), which has a rich symbolic vocabulary, is considered one of the most comprehensive and outstanding design artifacts for modeling all the intended behaviors of the SUT [7, 8]. Existing studies focused mainly on analyzing control flow among activities in an AD from the high-level business process (HLBP) to detect various control flow errors (e.g., loops and synchronization) [7, 9-12].

Generating abstract test cases to examine the control flow among design elements such as the AD on its own is inadequate to test the complete behavior of systems. Most software systems deliver functionality in terms of data; consequently, errors can easily occur if the flow of data is inappropriately modeled, defined, and used. The underlying idea behind data flow testing (DFT) is to detect inappropriate use of data or bad use of calculations. This idea is well-suited to the AD-based approach because ADs represent the sequences of actions that embody lower-level steps in the overall activity and are central to the data flow aspect of activities [13].

The literature shows that automatic test data generation using search-based meta-heuristic optimization algorithms to provide an appropriate set of test data for a specified coverage criterion has a high cost efficiency and can reduce the human effort required compared with manual techniques [14]. Search-based test data generation using meta-heuristic optimization algorithms such as genetic algorithms (GAs) has been widely studied and practiced, mostly in structural testing [14-17]. Our objective in this paper is to use the ADs of the SUT for test data generation by using a genetic algorithm. For this purpose, we present an algorithm for the automatic identification of data flow information and a fitness function to cover the maximum number of definition-use (def-use) pairs. We call the approach AutoTDGen, and we implement it in a simple prototype tool. This paper extends our previous work [18] with the following additional contributions.

- For automated test data generation, we introduce a search-based approach using a GA with a fitness function designed to reward maximum coverage of def-use pairs.
- In addition to automating the test data generation process, this approach, unlike our previous study [19], can directly generate test cases from models of the SUT without transforming them into intermediate test models.
- We have empirically compared and contrasted the effectiveness of AutoTDGen with that of DFAAD and EvoSuite.

To compare the fault detection effectiveness and statement and branch coverage performance of AutoTDGen with the two alternative approaches, we performed an experimental investigation using three software systems chosen from open-source libraries. Through the experiments, we aimed to answer the following research questions (RQs): How do the tests generated by the proposed AutoTDGen perform, compared with alternative approaches, in terms of statement and branch coverage? What is the difference in fault detection effectiveness between the tests generated by the proposed AutoTDGen and alternative approaches? Is there any variation in the types of faults detected by the proposed AutoTDGen and alternative approaches? What is the interaction relationship between the test coverage and fault detection effectiveness of the adopted techniques and subject properties?

The rest of this paper is organized as follows: Section II introduces related work. Section III presents the background of this study, revisiting the data flow representation and concepts in an AD. Section IV introduces the main concepts and activities for automatic test data generation using the AD and search-based technique. An experimental investigation to assess and compare the proposed technique with two alternative approaches in terms of fault detection effectiveness and coverage performance is described in Section V. Section VI discusses the findings based on the reported results. Finally, Section VII discusses possible threats to the validity of this research, with the conclusion and future work presented in Section VIII.

II. RELATED WORK

A. AD-Based Test Data Generation

Currently, an extensive body of literature describes different strategies for using the ADs of the SUT for automated test generation [6,7]. However, most of the studies do not clearly specify their test data generation method [20-23]. According to a systematic mapping study [7] on MBT using UML ADs, some of the commonly adopted methods for test data generation are category-partition, combinatorial logic coverage, data-object trees, and condition-classification trees. For instance, a gray-box method was proposed [24] to use a UML AD to generate test cases directly from ADs without the cost of creating an extra test model. That study adopted the category-partition method to generate a rational combination of input values and output values. Another UML AD–based test-case generation technique [25] transformed an AD into software success and fault trees using combinatorial logic coverage for test data generation. Data-object trees were also proposed for test data generation [26], along with a condition-classification tree method [10].

However, most previous papers are poorly stated and do not precisely specify the test input generation methods. Furthermore, the support for those proposed approaches used very simple examples. Moreover, the results reported by those studies are mainly concerned with control flow–based testing issues from the HLBP. In this study, we examine the ADs from a lower level to analyze the flow of data among activities. We also perform an experimental investigation using relatively extensive open-source systems.

B. *Evolutionary Approaches Using Genetic Algorithms*

The search-based technique using GAs for test data generation is common in the existing literature [14, 27, 28]. A recent study [29] used meta-heuristic algorithms to adapt a model checker for integration testing of systems specified by graph transformation and generate a test suite that satisfies all def-use coverage criteria. Also, several studies suggested model-based test-case generation and prioritization using ADs and GAs. For instance, [30] applied a GA to prioritize test case scenarios generated from UML ADs by identifying the critical path clusters. Other test-case prioritization techniques in the context of model-based testing using ADs and GAs were proposed in [31] and [32]. A UML AD–based approach using an evolutionary algorithm for transition sequence exploration is presented in [33]. It generates test cases that include both test scenarios and test data. Our study differs from the existing approaches because we use a different fitness function that enforces maximum coverage of def-use pair paths.

In conclusion, a large body of literature uses the ADs of the SUT for automatic test-case generation with different strategies, including GAs. The AD models are useful for testing because they have a rich symbolic vocabulary for expressing the system's behavior, and they can be manipulated by automated means. Currently, AD-based testing methods focus on test automation mainly from the HLBP, analyzing control flow information that is missing important aspects (data flow). Previously, we performed an experimental investigation of data flow annotated activity diagram (DFAAD)-based testing [19], in which the flow of data is explicitly annotated in the ADs of the SUT to investigate the feasibility of using ADs to meet the data flow coverage criterion.

III. REVISITING DATA FLOW REPRESENTATION AND CONCEPTS IN ADS

This section presents the background for this study, revisiting the data flow representation and concepts in ADs [18] and providing an example and related definitions.

An *activity* is a behavior specified as a sequence of subordinate units using a control and data flow model [13]. *Actions* are executable nodes required for any significant capability of an activity; they embody lower-level steps in the overall activity. Action notation appears only in the AD and is central to the data flow aspects of activities [13]. Actions can invoke other behaviors and operations, access and modify objects, and perform more advanced coordination of other actions. For example, Fig.1(a) has two actions: a CreateObject action that creates an object of type Activity B, and a CallBehavior action that calls Activity B. Actions also have a specialized form of object nodes (Pins) so that object flows can get and provide data. For example, Fig.1(b) has a SendObject action that sends an object via its output Pin and another action that gets the object via its input Pin and manipulates its contents.

These object flows provide the primary means for exchanging data between activities. Activities also introduce parameters and structural features or variables. As depicted in Fig.1(c), the use of variables in an AD effectively provides indirect data flow paths from the point at which a value is written to the variable to all the points at which the value is read from the variable [13]. By analyzing an action sequence and its behavior and using the information in action Pins, activity parameters, and decision constraints, it is possible to automatically track the flow of data among activities in an AD.

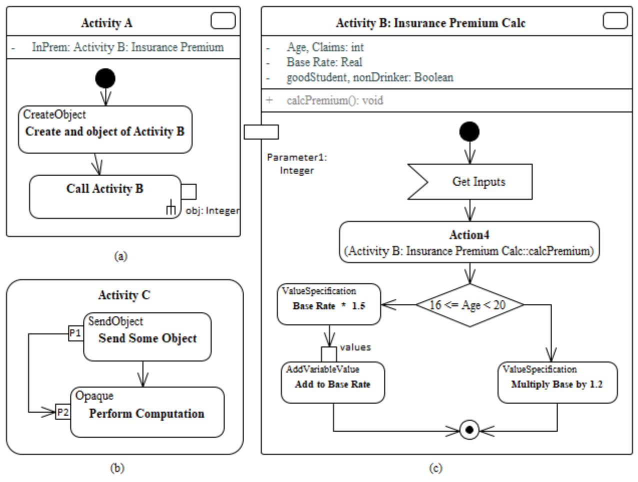


Fig.1. Data flow representation and concepts in ADs.

Definition 1) An AD is a directed graph that can be described formally as an activity model, M = (A, E, C, V, a_I, a_F) where:

- A = {a_1, a_2, …., a_n} is a set of executable nodes (actions are the only kind of executable node, and object nodes are attached to actions as action Pins by executable nodes; we indicate the lower-level steps in the overall activities).
- E denotes a set E of edges, where E ⊆ {$A \times A$}.
- C = {c_1, c_2, …., c_n} = $D_N \cup J_N \cup F_N \cup M_N$ 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 with predicate use.
- a_I is an initial activity node, where $a_I \subseteq$ A and $a_I \neq \emptyset$.
- a_F is a set of final nodes, where $a_F \subseteq$ A and $a_F \neq \emptyset$.
- V = {v_1, v_2, …., v_i} is a set of variables s.t. V includes both object variables and structural attributes (variables).

Definition 2:Variables can appear in different contexts in an AD. Let M be a model that has a set of variables V:

- Variable $v_i \in$ V is defined in node a_n of M, indicated as $def_{a_n}^{v_i}$, meaning that variable v_i is defined in node a_n if v_i is initialized in the corresponding node or its value has been modified by the execution of a certain action.
- Variable $v_i \in$ V is in predicate use (pu) in decision node d_n of M, indicated as $pu_{d_n}^{v_i}$, meaning that variable v_i is in pu in decision node d_n if it appears as a constraint of DN.
- Variable $v_i \in$ V is in computation use (cu) in node a_n of M, indicated as $cu_{a_n}^{v_i}$, meaning that variable v_i is in cu in node a_n if the corresponding node behavior is involved in any sort of computation.

IV. AUTOTDGEN APPROACH

This section introduces the main concepts and activities for automatic test-data generation using the AD and search-based technique. In our approach, tests are generated directly from the ADs of the SUT without converting them into another graphical intermediate test model. However, the ADs are converted into XML, and the XML documents are represented as an ElementTree (ET) using a Python built-in library for the ease of reading and manipulating the automatic data flow analysis.

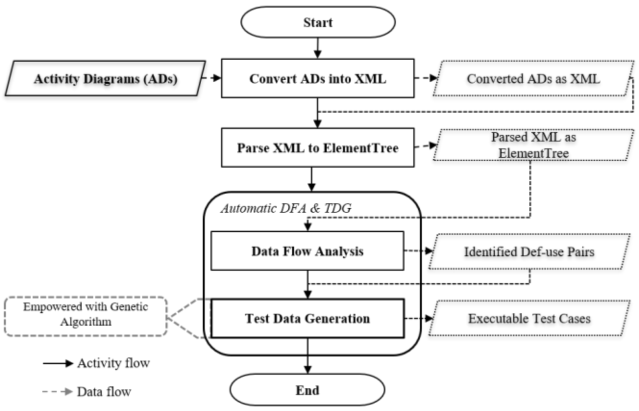


Fig.2. An overview of the major activities in our approach

Fig.2 illustrates the major activities in our approach, including the associated input and output artifacts. Taking the ADs of the SUT as input, the ADs are converted into XML documents, and the XML documents are parsed as an ET. Using the ET, the flow of data among the activities in each AD is automatically analyzed to identify the def-use pairs. Finally, a GA is applied to generate test data and make the test cases executable. Each activity is described in more detail in the following sub-sections.

A. *Convert ADs into XML*

For automatic manipulation of the ADs for data flow analysis, the first step is converting the AD models into XML.

Depending on the UML modeling tools used, converting an AD model into an XML file is as straightforward as a few simple clicks. Most UML modeling tools, including the one we used, support the easy export of models in XML format.

B. *Parse XML to ElementTree*

XML documents originally use a hierarchical data format that can easily be represented as a tree-like structure. Fig.3 shows a high-level schematic diagram of an XML document represented as an ET. As depicted in the figure, the ADs contain four major elements that are useful for data flow analysis: Activity element, Action element, ActionPin element, and Decision element, each of which has properties, links, and tags.

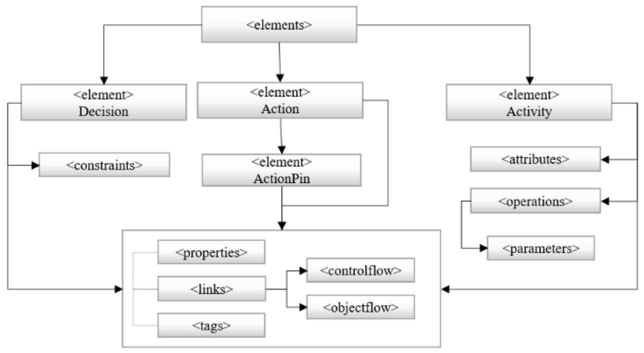


Fig.3. A high-level XML ElementTree schematic diagram

C. *Automatic Data Flow Analysis & Test Data Generation*

This section presents the main idea of our study: automatic data flow analysis and search-based test data generation using a GA. The test data are generated in terms of the data flow coverage criterion, specifically all definition use coverage. The flow of data among activities is directly identified from the ADs of the SUT without transformation into an intermediate test model. Sub-section 1 describes the automatic data flow analysis, specifically the algorithm for generating def-use pairs. The GA and its fitness function for test data generation are provided in sub-section 2.

1. Data Flow Analysis: This sub-section presents how the flow of data among activities in an AD is analyzed automatically. Algorithm 1 describes the generation of def-use pairs from an AD represented as an ET.

The algorithm takes a model *M* of an AD in the form of an ET; a given node *c*, which is the current node; an initial node a_i; an object variable *v*; and a path *P*. Starting from the given node *c* in *M* (and ensuring that *c* is not in *P*), the algorithm checks the properties for the corresponding *c*. If *v* is not in *def*, the algorithm appends *c* to *P*, sets the incoming node as the current node, and makes a recursive call. The iteration continues until it finds a node in which *v* is in *def* and returns the path *P*. The algorithm also checks whether *P* is subsumed by another path. If the path is subsumed, it sets the prefix to yes (Y) and appends *P* to the list *L*. Otherwise, it sets the prefix to no (N) and appends the discovered *P* to *L*. Simply put, the algorithm finds the *def* of v_i used in a given node anywhere in the model by searching backward or forward based on the tester's preference.

Algorithm 1. *ExtractDuPairs* $(\mathcal{M}, c, a_i, v, \mathcal{P}[]$: *The algorithm for* automatically generating the def − use pair paths.

```
Input: An AD model $\mathcal{M}$, a current given node $c$, an initial node $a_i$,
an object variable $v$ and a list of path $\mathcal{P}$:
Outputs: $\mathcal{L}$: A list of all def − use pair paths
1: Begin
2: $\mathcal{L} \leftarrow \emptyset$
3: $\mathcal{P} \leftarrow \mathcal{P} + [c]$ #add current node to a path $\mathcal{P}$
4:   if $c == a_i$ return $\mathcal{P}$
5:     for node $c$ incoming edges, $e = (m, n)$ s.t $n \in \mathcal{E}$ do
6:       if $c$ in $\mathcal{M}$ and node c has not been visited then
7:         record edge $e$ as the discovery edge for node $c$
8:         #get previous node $p$
9:         for $v_i$ in $c$ do:   #Check the occurance of $v$ in $c$
10:          check the type of node $c$:
11:          append $c$ to $\mathcal{P}$ as $def_{a_n}^{v_i} \vee pu_{d_n}^{v_i} \vee cu_{a_n}^{v_i}$
12:          if $\mathcal{P}$ is subsumed by another $\mathcal{P}$ then
13:            $\mathcal{L} \leftarrow \mathcal{P} \wedge \mathcal{PF} \leftarrow \mathcal{Y}$ #Set prefix to yes
14:          else
15:            $\mathcal{L} \leftarrow \mathcal{P} \wedge \mathcal{PF} \leftarrow \mathcal{N}$ #Set prefix to no
16:        endfor
17:        #set the previous node $n$ as start node $s$
18:        $s \leftarrow n$
19:        Recursively call ExtractDuPairs $(\mathcal{M}, s, a_i, v, \mathcal{P})$
20:      endif
21:    endfor
22:    return $\mathcal{L}$
23:  endif
24: End
```

An example of generated def-use pair paths is depicted in Table I. To save space, the textual element names are denoted as numerical values except for the decision element that includes branches used to guide the search. The prefix "Y" signifies that the def-use pair is subsumed by another path, and the prefix "N" means that the def-use pair is not subsumed by any other path or paths.

TABLE I. AN EXAMPLE OF GENERATED DEF-USE PAIR PATHS

Variable	*Def-use path set*	*Def-use pair paths*	*Prefix*
Locks	(1, locks)	[1, locks ≥ 1]	Y
		[1, locks ≥ 1, 3, 4]	N
	(5, locks)	[5, 6, locks ≥ 4]	Y
		[5, 6, locks ≥ 4, 3, 4]	N
Stocks	(3, stocks)	[3, stocks ≥ 1)	N
Barrels	(3, barrels)	[3, barrels >= 1)	N
Commissions	(14, commissions)	[14, 31≥commissions && commissions >15, 21]	N
Sales	(8, sales)	[8, sales ≥ 1800]	Y
		[8, sales ≥ 1800, 10]	Y
		[8, sales ≥ 1800, 10, 11, 12, 13, 14]	N
		[8, 10, sales ≥ 1000]	Y
		[8, 10, sales ≥ 1000, 16, 17, 18]	N
		[8, 10, sales ≤ 990, 19, 20]	N

2. Test Data Generation: To generate test data, we use the search-based technique with a GA that is applied to a set of def-use pairs. The def-use pairs are automatically generated from the ADs of the SUT. GAs are optimization and heuristic search techniques that can find optimal solutions to a wide range of software engineering problems with many potential solutions [34,14]. A key aspect of generating test data using a GA is designing a fitness function to determine how the test data generated by the GA fit the desired solution (the target def-use pairs). Thus, designing an appropriate fitness function to guide the search is the key to the entire search-based test data optimization technique.

Fitness Function: In this paper, we focus on data flow coverage as the test coverage criterion, but our proposed technique is not limited to a certain test criterion. It can be generalized to any test criterion. In data flow–based testing, an individual test datum can satisfy more than one def-use pair (Table I). Thus, to guide parent selection and evaluate how the test data generated by the GA cover the target def-use pairs, we designed a fitness function that rewards maximum coverage of def-use pairs. For example, if more than one def-use pair is reachable and subsumed by another def-use pair, the prefix guides the algorithm to reward the one that is not subsumed or dominated.

To guide the search toward the target def-use pairs, we use both the approach level (AL) and the branch distance (BD) for the fitness function [35]. The AL is calculated based on the number of def-use pairs covered by an individual test datum, and the BD is calculated using the predicates. The BD heuristic is commonly used for test data generation and evaluates how close a branch is to being estimated as covered or not covered [36]. For instance, if we have a variable x in a predicate x > 6 with the value x = 2, the branch distance will be calculated as 6 – 2 + k, with the number k > 0. For the BD heuristic, we used a set of rules defined in [37]. The def-use pairs coverage fitness function to minimize the AL and BD for an individual v_i and a target def-use pair (dup) is defined as:

$$f(v_i, dup) = n(\mathcal{AL}) + \mathcal{BD} \qquad (1)$$

To optimize the def-use pairs coverage it is important to make sure that all the def-use pairs are covered. Therefore, the $f(v_i, dup)$ function is defined as:

$$f(v_i, dup) = \begin{cases} 0, & \textit{if branch has been covered, and the appraoch level has reached zero} \\ n(a_{norm}(AL)), & \textit{if branch has been covered, and the approach level has not reached zero} \\ n(a_{norm}(AL)) + BC, & \textit{otherwise} \end{cases}$$

Where:

$$n(a_{norn}(\mathcal{AL})) = \frac{\mathcal{T}_{tar} - \mathcal{C}_{cov}}{\mathcal{T}_{ttcov} + len(\mathcal{L})} \qquad (a)$$

$$n(a_{norm}(\mathcal{AL})) + \mathcal{BD} = \frac{\mathcal{T}_{tar} - \mathcal{C}_{cov}}{\mathcal{T}_{ttcov} + len(\mathcal{L})} + \mathcal{BD} \qquad (b)$$

Here, T_{tar} is the total number of target def-use pairs to be covered related to a specific object variable, C_{cov} refers to the targets covered by an individual v_i, T_{ttcov} is the total number of remaining target def-use pairs after the C_{cov} subtraction, and len(L) is a constant for normalizing the AL [0,1], where L is the entire set of target def-use pairs. The $n(a_{norm}(AL))$ normalized function ensures that all the def-use pairs are covered. It enables the search algorithm to yield an optimal solution wherever a branch is covered for a specific target def-use pair and keep searching to find a solution for the remaining targets. Hence, an individual is considered to be optimal when it covers at least one target def-use pair and its fitness value is $f(v_i, dup) < 1$.

In the following example, consider the dup set (8, sales) in Table I, which has a total of six def-use pairs associated with it, when len(L) = 5. The variable sales is in the form of a function composition such that the range of three variables (locks, stocks, and barrels) is its domain. That is, three individuals are involved in this case, one for each variable, and they must be adjusted to calculate the fitness (e.g., locks * 45 + stocks * 30 + barrels * 25 = sales).

As shown below in a), the predicate involved in this case is sales ≥ 1800; the individuals searched by the algorithm to satisfy the BD are 24, 14, and 12 for locks, stocks, and barrels, respectively; the C_{cov} of these individuals is recorded as 3; and the fitness is calculated $f(\text{sales}, (8, \text{sales})) = 0.27$. In this example, the BD has been covered, and the fitness, $f(v_i, dup)\,! = 0$, indicating that targets remain to be covered. As shown below in b), because $T_{ttcov} = T_{tar} - C_{cov} = 3$, the predicate involved in this case is sales ≥ 1,000, and the solutions (individuals) found by the algorithm are 4, 4, and 28. The C_{cov} of these individuals is 2, and the fitness is calculated as $f(v_i, dup) = 0.12$, indicating that still more targets need to be covered. That evaluation process is repeated until $f(v_i, dup) = 0$. As shown below in c), when the fitness reaches zero, the test data covering all def-use pairs associated with the variable sales have been optimized. The example in d) shows a situation in which neither the branch is covered nor the AL has reached zero. Therefore, $f(v_i, dup) > 1$, indicating that the individuals are not optimal test data.

a) $f(\text{sales}, (8, \text{sales})) = \frac{(6-3)}{6+5} + 0 = 0.27$
$BD = ((24 * 45) + (14 * 30) + (12 * 25)) - 1800 = 0$

b) $f(\text{sales}, (8, \text{sales})) = \frac{(6-3-2)}{3+5} + 0 = 0.12$
$BD = ((4 * 45) + (4 * 30) + (28 * 25)) - 1000 = 0$

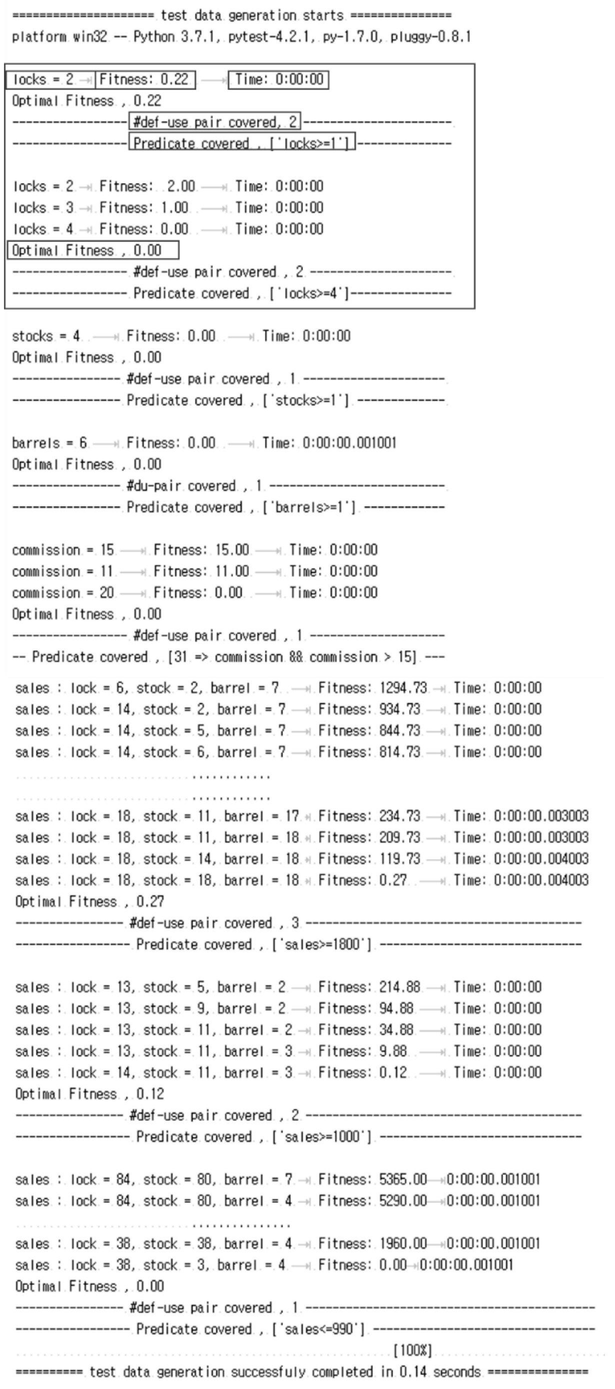
c) $f(\text{sales}, (8, \text{sales})) = \frac{(6-3-2-1))}{1+5} + 0 = 0.00$
$BD = 990 - ((8 * 45) + (16 * 30) + (6 * 25)) = 0.00$

d) $f(\text{sales}, (8, \text{sales})) = \frac{(6-0)}{6+5} + 85 = 85.54$
$BD = ((23 * 45) + (10 * 30) + (22 * 25)) - 1800 = 85$

The overall fitness function to minimize a set of test data V on a set of def-use pairs L is thus as follows:

$$fitness(\mathcal{V}, \mathcal{L}) = \sum_{dup \in \mathcal{L}} f(v_i, dup) \qquad (2)$$

Fig.4 shows a snapshot of the test data generated for all the def-use pairs in Table I. The figure displays the object variables and their associated test data (individuals) generated by the algorithm, the calculated fitness value, the number of def-use pairs covered by each individual, and the related branches, as well as the time spent. Some of the generated data with higher fitness values are deliberately omitted to save space, and they are emphasized with dots showing the continuation of the optimization process.



```
==================== test data generation starts ===============
platform win32 -- Python 3.7.1, pytest-4.2.1, py-1.7.0, pluggy-0.8.1

locks = 2   Fitness: 0.22     Time: 0:00:00
Optimal Fitness , 0.22
-----------------#def-use pair covered, 2-----------------------
-----------------Predicate covered , ['locks>=1']--------------

locks = 2   Fitness:  2.00    Time: 0:00:00
locks = 3   Fitness: 1.00     Time: 0:00:00
locks = 4   Fitness: 0.00     Time: 0:00:00
Optimal Fitness , 0.00
----------------- #def-use pair covered , 2 --------------------
----------------- Predicate covered , ['locks>=4']--------------

stocks = 4     Fitness: 0.00     Time: 0:00:00
Optimal Fitness , 0.00
---------------- #def-use pair covered , 1 --------------------
---------------- Predicate covered , ['stocks>=1'] ------------

barrels = 6    Fitness: 0.00     Time: 0:00:00.001001
Optimal Fitness , 0.00
---------------- #du-pair covered , 1 -------------------------
---------------- Predicate covered , ['barrels>=1'] -----------

commission = 15     Fitness: 15.00    Time: 0:00:00
commission = 11     Fitness: 11.00    Time: 0:00:00
commission = 20     Fitness: 0.00     Time: 0:00:00
Optimal Fitness , 0.00
----------------- #def-use pair covered , 1 -------------------
-- Predicate covered , [31 => commission && commission > 15] ---
sales : lock = 6, stock = 2, barrel = 7    Fitness: 1294.73   Time: 0:00:00
sales : lock = 14, stock = 2, barrel = 7   Fitness: 934.73    Time: 0:00:00
sales : lock = 14, stock = 5, barrel = 7   Fitness: 844.73    Time: 0:00:00
sales : lock = 14, stock = 6, barrel = 7   Fitness: 814.73    Time: 0:00:00
                              ...........
                              ...........
sales : lock = 18, stock = 11, barrel = 17   Fitness: 234.73   Time: 0:00:00.003003
sales : lock = 18, stock = 11, barrel = 18   Fitness: 209.73   Time: 0:00:00.003003
sales : lock = 18, stock = 14, barrel = 18   Fitness: 119.73   Time: 0:00:00.004003
sales : lock = 18, stock = 18, barrel = 18   Fitness: 0.27     Time: 0:00:00.004003
Optimal Fitness , 0.27
---------------- #def-use pair covered , 3 ----------------------------------------
----------------- Predicate covered , ['sales>=1800'] -----------------------------

sales : lock = 13, stock = 5, barrel = 2    Fitness: 214.88   Time: 0:00:00
sales : lock = 13, stock = 9, barrel = 2    Fitness: 94.88    Time: 0:00:00
sales : lock = 13, stock = 11, barrel = 2   Fitness: 34.88    Time: 0:00:00
sales : lock = 13, stock = 11, barrel = 3   Fitness: 9.88     Time: 0:00:00
sales : lock = 14, stock = 11, barrel = 3   Fitness: 0.12     Time: 0:00:00
Optimal Fitness , 0.12
---------------- #def-use pair covered , 2 ----------------------------------------
----------------- Predicate covered , ['sales>=1000'] -----------------------------

sales : lock = 84, stock = 80, barrel = 7   Fitness: 5365.00  0:00:00.001001
sales : lock = 84, stock = 80, barrel = 4   Fitness: 5290.00  0:00:00.001001
                              ..............
sales : lock = 38, stock = 38, barrel = 4   Fitness: 1960.00  0:00:00.001001
sales : lock = 38, stock = 3, barrel = 4    Fitness: 0.00  0:00:00.001001
Optimal Fitness , 0.00
---------------- #def-use pair covered , 1 ------------------------------------------
----------------- Predicate covered , ['sales<=990'] --------------------------------
                                                      [100%]
========== test data generation successfuly completed in 0.14 seconds ===============
```

Fig.4. Snapshot of the test data generated for Table I.

As can be seen in the figure, the algorithm's run time for generating the test data is very short. Although the algorithm runs several times, it does not affect the execution time. The efficiency of this automated test data generation is thus not comparable with the manual generation of test data by humans, and we will not further consider its cost-effectiveness or efficiency in contrast to manual test-data generation.

V. EXPERIMENT DESCRIPTION

Our goal for this experiment was to assess and compare AutoTDGen with two alternative approaches, DFAAD and EvoSuite. Following existing guidelines for empirical studies [38, 39] and experimental investigations [40] in software engineering, we here detail the experiments we carried out to assess the proposed AutoTDGen and answer our four RQs.

A. *Experimental Subjects*

The context of our study is a manual selection of the three open-source software systems that we used in our previous experiment on DFAAD [19]. The Elevator and Cruise Control systems were chosen from the Software-artifact Infrastructure Repository[1]. The Coffee Maker system[2] is used as an example of black-box testing by the NCSU computer science department. Table II summarizes the three experimental subjects in more detail.

Admittedly, the size of our experimental subjects is not very large. In fact, finding large open-source software systems with all the requirements (e.g., use cases) and design artifacts (e.g., class diagrams) is quite difficult. Understanding and modeling system behavior without the availability of such documentation is very difficult. However, as also stated in [41], it is important to remember that in model-based development, ADs are used to model the intended behavior of the SUT in individual use cases rather than modeling the whole system in a single AD. Therefore, even when the subject systems are small, the ADs can be quite large and complex in terms of the flow of control among activities, decision constraints, activity parameters, etc.

TABLE II. EXPERIMENTAL SUBJECTS

		Subjects		
		Cruise Control	***Elevator***	***Coffee Maker***
#LOC		358	581	393
#Classes		4	8	4
Branches	Min	10	0	16
	Mean	16.5	17.5	26.7
	Max	28	72	48
Statements	Min	31	8	42
	Mean	41.5	45.75	52.7
	Max	62	152	72
Mutants	Min	15	2	24
	Mean	27.25	30.9	39
	Max	48	111	68

Because the task of generating the ADs of the SUT manually is quite time-consuming, as detailed in a systematic mapping study for MBT using ADs [7], most of the existing literature used small examples for validation. In our previous experiment, we also considered those systems to be adequate in terms of their size and complexity [19, 41]. Because the three systems used in [41] are very similar in terms of the system behavior, we used the Coffee Maker instead of OrdSet to ensure the diversity of the subjects for better assessment, especially when comparing AutoTDGen with EvoSuite.

[1] https://sir.csc.ncsu.edu/portal/index.php

[2] https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/

B. *Experiment Planning*

a) Research Questions: In this experimental investigation, we addressed the following four RQs:

- *RQ1*: How do the tests generated by the proposed AutoTDGen perform, compared with alternative approaches, in terms of statement and branch coverage?
- *RQ2*: What is the difference in fault detection effectiveness between the tests generated by the proposed AutoTDGen and alternative approaches?
- *RQ3*: Is there any variation in the types of faults detected by the proposed AutoTDGen and alternative approaches?
- *RQ4*: What is the interaction relationship between the test coverage and fault detection effectiveness of the adopted techniques and subject properties?

RQ1, *RQ2*, and *RQ3* compare the proposed AutoTDGen with alternative approaches in terms of the percentage of coverage attained and the test effectiveness, indicated by the proportion of artificially seeded faults (mutants) detected and variations in the types of faults detected. *RQ4* investigates whether the performance of the applied testing techniques is affected by the properties (e.g., complexity, concurrency, or real-time behavior) of the subject systems.

b) Baseline selection: To answer our RQs, we selected the following two testing techniques for comparison:

DFAAD, an experimental investigation of data flow annotated AD-based testing from our previous study [19]. In DFAAD, the flow of data is explicitly marked manually across the ADs of the SUT. We used DFAAD to investigate the feasibility and benefits of using ADs to derive test cases for data flow testing. The previous study converted the ADs into an intermediate test model, a data flow graph, and provided the test inputs manually. Therefore, we used DFAAD as a baseline for our RQs. *EvoSuite*[3], an open-source test suite generator for Java classes that uses an evolutionary approach based on a GA [15, 42]. EvoSuite has been evaluated in various open-source and industrial software systems. In academia, particularly in search-based testing, EvoSuite is considered to be a state-of-the-art testing tool.

c) Fault Seeding: Finding subject systems of suitable size and properties with real faults to assess test techniques is difficult. Even if subjects with real faults exist, their faults are not diverse enough to be valuable for an experimental investigation. One solution to this problem is seeding a large number of distinct artificial faults either manually or automatically [43]. Because manually seeding a large number of faults is difficult, we applied PIT, a mutation testing tool, to introduce faults into the SUT automatically. PIT4 is a state-of-the-art mutation testing tool that is fast and scalable. In our experiment, we used the default mutator group: Conditional Boundary Mutator, Increments Mutator, Invert Negatives Mutator, Math Mutator, Negate Conditionals Mutator, Return Values Mutator, and Void Method Call Mutator.

[3] http://www.evosuite.org/

d) Variable Selection: In this experiment, the independent variable is the approach applied as a basis for test generation (AutoTDGen, DFAAD, or EvoSuite). Our dependent variables are the percentage of statements and branches covered, fault-detection effectiveness (the number of faults detected/failed to detect), and the type of faults detected (faults related to different mutation operators).

e) Experimental Protocol: Our experiments compared the proposed AutoTDGen with the two baselines as alternative approaches. Traditionally, the effectiveness of a test is measured based on the proportion of statements, lines, or branches covered. However, that coverage measures only which part of the code is executed. Therefore, we performed mutation testing to inject artificial faults into the SUT and ensure that all the statements or branches have truly been tested. To answer RQ1 and RQ2, the statement coverage (SC), branch coverage (BC), and fault detection (FD) ratio were measured as follows:

$$\mathcal{SC} = \frac{\#\,\mathcal{CS}_{covered\_statements}}{\mathcal{TS}_{total\_statements}} \times 100 \qquad (3)$$

$$\mathcal{BC} = \frac{\#\,\mathcal{CB}_{covered\_branches}}{\mathcal{TB}_{total\_branches}} \times 100 \qquad (4)$$

$$\mathcal{FD} = \frac{\#\,\mathcal{DF}_{\text{detected\_faults}}}{\mathcal{TF}_{\text{total\_faults\_injected}}} \times 100 \qquad (5)$$

C. *Experimental Results*

This section presents the results of the experiments by answering the RQs.

RQ1: *How do the tests generated by the proposed AutoTDGen perform, compared with alternative approaches, in terms of statement and branch coverage*? The results presented in Table III for SC indicate that AutoTDGen performed slightly better than DFAAD and EvoSuite on two subjects, although EvoSuite had the best SC coverage in one subject (Coffee Maker). In terms of BC, AutoTDGen achieved greater BC than EvoSuite in only a single subject (Elevator), but it outperformed DFAAD in all subjects. On the other hand, except for the Elevator system, EvoSuite performed very well in terms of BC. The average coverage achieved by EvoSuite was higher than that of DFAAD, but AutoTDGen performed better on average, with an improvement in covered branches of 3.67 percent. Thus on average, AutoTDGen achieved higher or nearly equal branch and statement coverage compared with both DFAAD and EvoSuite. However, as reported in Table III, the magnitude of the differences is small.

RQ2: *What is the difference in fault detection effectiveness between the tests generated by the proposed AutoTDGen and alternative approaches*? The overall results for fault detection effectiveness are also reported in Table III. AutoTDGen outperformed both EvoSuite and DFAAD in all

[4] https://pitest.org/

experimental subjects, with a mean effectiveness difference of 31.53 percent and 8.23 percent, respectively.

TABLE III. THE OVERALL SC, BC, AND FAULT DETECTION RATIO ACHIEVED ACROSS SUBJECTS AND APPLIED TECHNIQUES

Subject Systems	**DFAAD**			**EvoSuite**			**AutoTDGen**		
	SC	***BC***	***FD***	***SC***	***BC***	***FD***	***SC***	***BC***	***FD***
Cruise Control	97.6	77.3	67.9	97	**95.5**	44	**99**	83.3	**73**
Elevator	96.2	86.4	69.6	77	63	24.3	**96.5**	87.4	**85.82**
Coffee Maker	98	90	84.6	**100**	**100**	83.8	99.4	98.7	**88**
Mean	97.3	84.6	74	91.25	**86.13**	50.7	**98.3**	89.8	**82.33**

To provide a better view of fault detection effectiveness, Fig.6 highlights the number of detected and failed to detect faults, as well as the fault detection ratio across subject systems and applied techniques. As shown in Fig.5, the best fault-detection result achieved by AutoTDGen was for Elevator, which is large and complex with 247 seeded mutants, of which AutoTDGen detected 212 mutants, compared with 60 and 172 mutants detected by EvoSuite and DFAAD, respectively.

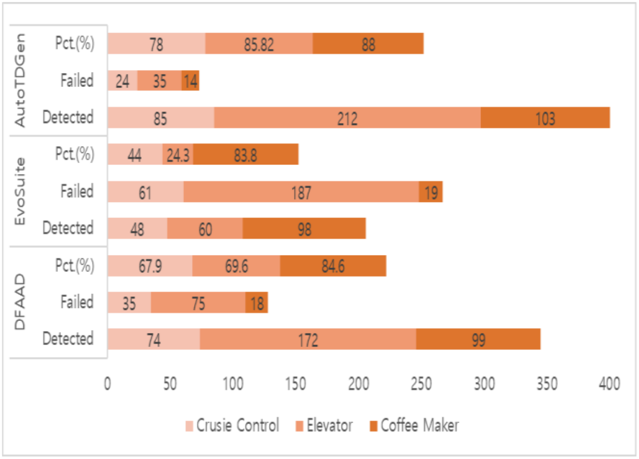


Fig.5. The percentage and number of mutants detected and failed to detect.

***RQ3**: Is there any variation in the types of faults detected by the proposed AutoTDGen and alternative approaches*? The types of fault detected across the experimental subjects and techniques are reported in Table IV, Table V, and Table VI. The tables list the total number of faults (mutants) seeded per mutation operator, the number of faults detected, the number of failed to detect faults, and the percentage of faults detected per mutation operator by each technique. The best results for each condition are bolded.

RQ3 investigates whether AutoTDGen is more likely to detect certain types of faults compared with the alternative approaches. The results indicate a great variation in the number and types of faults detected by the applied techniques. For instance, out of the 30 MM seeded faults for the Elevator system, 26 faults were detected by AutoTDGen, and 14 faults were detected by DFAAD, but none of the MM faults seeded in the Elevator system were detected by EvoSuite.

TABLE IV. COMPARISON RESULTS FOR THE TYPES OF MUTATIONS COVERED IN CRUISE CONTROL

Mutation operators	*Total mutants*	**Cruise Control**								
		AutoTDGen			***DFAAD***			***EvoSuite***		
		Detected	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*
IM	0	0	0	0	0	0	0%	0	0	0
VMCM	26	23	3	**88**	19	7	73	1	25	3.8
RVM	20	20	0	**100**	18	2	**90**	18	2	**90**
MM	20	10	10	**50**	6	14	30	3	17	15
NCM	33	32	1	**97**	31	2	94	26	7	78.8
INM	0	0	0	0	0	0	0	0	0	0
CBM	10	0	10	0	0	10	0	0	10	0

TABLE V. COMPARISON RESULTS FOR THE TYPES OF MUTATIONS COVERED IN ELEVATOR

Mutation operators	*Total mutants*	**Elevator**								
		AutoTDGen			***DFAAD***			***EvoSuite***		
		Detected	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*
IM	4	4	0	**100**	4	0	**100**	3	1	75
VMCM	76	54	22	**71**	45	31	59	11	65	14
RVM	45	39	6	**87**	34	11	76	16	29	35
MM	30	26	4	**87**	14	16	47	0	30	0
NCM	70	69	1	**99**	62	8	**89**	18	43	26
INM	1	1	0	**100**	1	0	**100**	0	1	0
CBM	21	19	2	**90**	12	9	57	12	9	57

TABLE VI. COMPARISON RESULTS FOR THE TYPES OF MUTATIONS COVERED IN COFFEE MAKER

Mutation operators	*Total mutants*	**Coffee Maker**								
		AutoTDGen			***DFAAD***			***EvoSuite***		
		Detected	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*	*Detected*	*Failed*	*Pct. (%)*
IM	6	6	0	100	6	0	100	6	0	100
VMCM	12	12	0	**100**	12	0	**100**	7	5	58
RVM	26	25	1	96	25	1	96	26	0	**100**
MM	9	9	0	**100**	9	0	**100**	4	5	44
NCM	40	40	0	100	40	0	100	40	0	100
INM	0	0	0	0	0	0	0	0	0	0
CBM	24	11	13	46	7	17	29	15	9	**62**

RQ4: *What is the interaction relationship between the test coverage and fault detection effectiveness of the adopted techniques and subject properties*? In terms of fault detection effectiveness, the results indicate a remarkable relationship between the subject properties and adopted techniques. Among the experimental subjects, the Elevator system is peculiar and difficult to test due to its concurrency and complex behavior, and AutoTDGen managed to detect more of its faults than the other techniques. On the other hand, in terms of statement and branch coverage, the results showed no notable relationship between the performance of a technique and the properties of the subjects, though the small improvement attained by AutoTDGen in terms of the statement and branch coverage was in the Elevator system. Furthermore, the DFAAD and AutoTDGen performance on SC and BC was comparatively consistent across the subjects, whereas the EvoSuite performance varied from subject to subject depending on the subject properties. Specifically, EvoSuite achieved poor SC for Elevator and the best coverage for Coffee Maker. In *summary*, the reported results indicate that the concurrency, dynamicity, and complexity of the subjects are a good indicator of the adopted technique's capability for fault detection.

VI. DISCUSSION

Our experimental results showed slight differences among the three test generation techniques in terms of the statement and branch coverage. EvoSuite is a powerful tool for maximizing statement and branch coverage, in some cases achieving even greater coverage than AutoTDGen and DFAAD. Although the notable improvement accomplished by AutoTDGen was for Elevator, overall, the reported results indicate that the properties of the experimental subjects do not have much effect on the statement and branch coverage performance of the applied test generation techniques. For example, despite the real-time behavior and complexity of the Elevator system over Cruise Control and that of Cruise Control over Coffee Maker, which provides comparatively simple functionality, all three applied test techniques achieved entirely satisfactory coverage of all three subjects. However, it is important to remember that the statement and branch coverage measures only which statements or branches are executed by the test; it does not fully reflect the completeness of the test.

The experimental results indicate that AutoTDGen outperformed both EvoSuite and DFAAD for all experimental subjects in fault detection. Unlike the statement and branch coverage, this result shows that the properties and functionality of the experimental subjects affect the fault detection effectiveness of the applied testing techniques in important ways, as shown by Fig.5.

The three applied test techniques showed an important variation in the types of faults detected. Because the experimental subjects with complex properties involve more computation, they have more computational-related operators (MM). AutoTDGen and DFAAD outperformed EvoSuite in detecting computational-related faults, possibly because of the modeling capability of ADs in capturing the concurrency and real-time properties of the system, which could allow the AD-based techniques to generate more suitable tests than EvoSuite when the criteria are well-defined.

Summary: Our main observation is that when the complexity of the experimental subjects increases, EvoSuite tends to have lower fault detection capability than AutoTDGen and DFAAD, as shown by EvoSuite's large number of undetected MM faults across experimental subjects, especially in the case of Elevator, where it had zero coverage. On the other hand, AutoTDGen, and DFAAD are good at handling the real-time behaviors and properties of the SUT. Therefore, EvoSuite needs further improvement to handle concurrency and real-time behaviors. Overall, our results also show that the difference between DFAAD and AutoTDGen is relatively small. However, the first one is manually generated and the latter is automated, so even a small difference can lead to a much greater efficiency.

VII. THREATS TO VALIDITY

External Validity. A common threat to external validity in all experimental studies is the size and types of SUT used. A lack of available open-source systems with sufficient specifications and design artifacts limited our study to small experimental subjects. Despite our best effort in manually selecting experimental subjects that varied by size, behavior, and complexity, our study did not escape this threat to validity. However, in model-based practice using ADs, software systems are not modeled in a single AD, and even small systems can have very complex ADs.

Construct Validity. A threat to construct validity could be that we applied PIT, an automatic mutation testing tool, to generate synthetic faults. Using systems with real faults or seeding faults manually, or even using different mutation testing tools might yield different results. However, in software testing experiments, mutation testing (injecting artificial faults into the SUT) has become a common practice because finding systems with real faults is difficult. Previous studies have claimed that artificially injected faults are adequately representative of real faults [43].

Internal Validity. Keeping in mind that in MBT, the quality of the test is always directly related to the quality of the test model, a threat to internal validity could be related to the individuals modeling the SUT, which could influence the extracted results. A poor modeling exercise in practice could lead to different results. *Conclusion Validity*. A major threat to conclusion validity in our study is a lack of statistical analysis. To better specify our experimental goals, we could have performed hypothesis testing. However, because our results are quite intuitive and the differences are notable, we believe such statistical testing is unnecessary.

VIII.CONCLUSION AND FUTURE WORK

Search-based testing using GAs and ADs has been widely adopted to automate the test generation process. In this paper, we have presented a search-based technique to automatically generate test data for data flow testing using ADs and a GA. Our experimental investigation used three open-source systems to compare and assess the applicability of the proposed technique against two alternative approaches. The experimental results show the superiority of the proposed technique in terms of fault detection effectiveness, as indicated by an improved fault detection ratio. We also noticed a remarkable variation in the effectiveness of the applied techniques in terms of the types of faults detected. Specifically, AutoTDGen detected more MM-related faults than the other techniques. However, in terms of statement and branch coverage, the results indicated no significant differences among the applied techniques.

Even though the experimental data indicate that this approach has better fault detection capability, our technique could be further investigated and improved. For instance, further investigation could be conducted using different software systems with varying complexity and size. Moreover, our prototype tool can be extended and improved to incorporate other kinds of search heuristics algorithms. In the future, we intend to extend our approach by incorporating different kinds of search heuristics. We also want to empirically evaluate our approach using different cases with different domains, sizes, and complexities.

REFERENCES

[1] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, “A search-based OCL constraint solver for model-based test data generation,” in Proc. QSIC, Madrid, Spin, 2011, pp. 41-50.

[2] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, "Recent advances in model-based testing," Elsevier, Advances in Computers, vol. 101, pp. 53-120, 2016.

[3] M. Utting, and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Elsevier Inc, San Francisco, 2007.

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

[5] P. C. Jorgensen, *The Craft of Model-Based Testing*, CRC Press, Boca Raton, FL, USA, 2017, pp. 3-13.

[6] 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, July. 2013.

[7] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, “Model-based testing using UML activity diagrams: a systematic mapping study,” Computer Science Review, vol. 33, pp. 98-112, Aug. 2019.

[8] M. Felderer, and A. Herrmann, “Comprehensibility of system models during test design: a controlled experiment comparing UML activity diagrams and state machines,” SQJ, vol. 27, pp. 125-147, Apr. 2019.

[9] P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba, “An enhanced test case generation technique based on activity diagrams,” in Proc. ICCES, Cairo, Egypt, 2011, pp. 289-294.

[10] S. Kansomkeat, P. Thiket, and J. Offutt, “Generating test cases from UML activity diagrams using the Condition-Classification Tree Method,” in Proc. ICSTE, PR, USA, 2010, pp. V1-62-V1-66.

[11] D. Kundu, and D. Samanta, “A novel approach to generate test cases from UML activity diagrams,” JOT, vol. 8, pp. 65-83, May-June 2009.

[12] A. Nayak, and D. Samanta, “Synthesis of test scenarios using UML activity diagrams,” SoSyM, vol. 10, no. 1, pp. 63-89, Feb. 2011.

[13] Unified Modeling Language (OMG UML®) Version 2.5.1, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/About-UML/

[14] D. S. Rodrigues, M. E. Delamaro, C. G. Correa, and F. L. S. Nunes, “Using genetic algorithms in test data generation: A critical systematic mapping,” ACM Comput. Surv., vol. 51, pp. 1-23, 2018.

[15] G. Fraser, and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in Proc. 19th ACM SIGSOFT symposium and 13th ECFSE, Szeged, Hungary, 2011, pp. 416-419.

[16] G. Fraser, A. Arcuri, and P. McMinn, “A memetic algorithm for whole test suite generation,” JSS, vol. 103, pp. 311-327, May 2015.

[17] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” IEEE TS ENG, vol. 44, pp. 122-158, 2017.

[18] A. Jaffari, C.-J. Yoo, and J. Lee, “automatic data flow analysis to generate test cases from activity diagrams,” KCSE, 2019, pp. 136-139.

[19] A. Jaffari, and C.-J. Yoo, “An experimental investigation into data flow annotated-activity diagram-based testing,” JCSE, vol. 13, no. 3, pp. 107-123, Sept. 2019.

[20] A. Heinecke, T. Brückmann, T. Griebe, and V. Gruhn, “Generating test plans for acceptance tests from UML activity diagrams,” in Proc. 17th IEEE ICWECBS, Oxford, UK, 2010, pp. 57-66.

[21] R. Chandler, C. P. Lam, and H. Li, “AD2US: an automated approach to generating usage scenarios from UML activity diagrams,” in Proc. 12th APSEC, Taipei, Taiwan, 2005, p. 8.

[22] X. Dong, H. Li, and C. P. Lam, “Using adaptive agents to automatically generate test scenarios from the UML activity diagrams,” in Proc. 12th APSEC, Taipei, Taiwan, p. 8.

[23] A. Hettab, E. Kerkouche, and A. Chaoui, “A graph transformation approach for automatic test cases generation from UML activity diagrams,” in Proc. C3S2E, Yokohama, Japan, 2008, pp. 88-97.

[24] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, “Generating test cases from UML activity diagram based on gray-box method,” APSEC, South Korea, 2005, pp. 284-291.

[25] S. Tiwari, and A. Gupta, “An approach to generate safety validation test cases from uml activity diagram,” in Proc. 20th APSEC, Bangkok, Thailand, 2014, pp. 189-198.

[26] X. Bai, C. P. Lam, and H. Li, “An approach to generate the thin-threads from the UML diagrams,” COMPSAC, 2004, pp. 546-552.

[27] G. Fraser, and A. Arcuri, “Whole test suite generation,” IEEE TS ENG, vol. 39, no. 2, pp. 276-291, Feb. 2013.

[28] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, “Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques,” ESE, vol. 12, no. 2, pp. 183-239, Nov. 2006.

[29] A. Kalaee, and V. Rafe, “Model-based test suite generation for graph transformation system using model simulation and search-based techniques,” IST, vol. 108, pp. 1-29, Apr. 2019.

[30] C. Sharma, S. Sabharwal, and R. Sibal, “Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams,” IJCSI, vol. 8 no. 2, p. 12, May 2011.

[31] P. Mahali, and A. A. Acharya, “Model based test case prioritization using UML activity diagram and evolutionary algorithm,” IJCSI, vol. 3, no. 2, pp. 42-47, 2013.

[32] F. M. Nejad, R. Akbari, and M. M. Dejam, “Using memetic algorithms for test case prioritization in model based software testing” in Proc. 1st CSIEC, Bam, Iran, 2016, pp. 142-147.

[33] M. Shirole, M. Kommuri, and R. Kumar, “Transition sequence exploration of UML activity diagram using evolutionary algorithm,” in Proc. 5th ISEC, Kanpur, India, 2012, pp. 97-100.

[34] O. Kramer, “Introduction, Genetic Algorithms,” in Genetic algorithm essentials, Springer, Cham, Switzerland, 2017, pp. 3-19.

[35] P. McMinn, “Search-based software test data generation: a survey,” Softw. Test. Verif. Reliab., vol. 14, no. 2, pp. 105-156, May 2004.

[36] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, “Search-based data-flow test generation,” in Proc. IEEE 24th ISSRE, USA, 2013, pp. 370-379.

[37] A. Arcuri, “It Does Matter How You Normalise the Branch Distance in Search Based Software Testing,” ICST, France, 2010, pp. 205-214.

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

[39] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, “Preliminary guidelines for empirical research in software engineering,” IEEE Trans. Softw. Eng., vol. 28, no. 8, pp. 721-734, Aug. 2002.

[40] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*, Springer-Verlag, Berlin Heidelberg, 2012, pp. 83-157.

[41] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. D. Penta, “Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments,” IEEE Trans. Softw. Eng., vol. 37, no. 2, pp. 161-187, Mar.-Apr. 2011.

[42] G. Fraser, and A. Arcuri, “Evolutionary Generation of Whole Test Suites,”, in Proc. 11th QSIC, 2011, pp. 31-40.

[43] J.H.Andrews, L.C.Briand, and Y.Labiche, “Is mutation an appropriate tool for testing experiments?,” ICSE, USA, 2005, pp. 402-411.