

# ONTOLOGICAL APPROACH FOR OVERCOMING PESTICIDE PARADOX IN INTER-CLASS TESTING

<sup>1</sup>SAYED ABDELGABER, <sup>2</sup>RASHA MANSOUR MOHAMED, <sup>3</sup>LAILA ABDEL HAMID, <sup>4</sup>A.ABDO

<sup>1234</sup>Information Systems Department .Faculty of Computers and AI, Egypt

<sup>4</sup>Faculty of Computing, Arab Open University, Egypt

E-mail: <sup>1</sup>sgaber@fci.helwan.edu.eg, <sup>2</sup>Roshy.mans@yahoo.com, <sup>3</sup>Eng.layla@fci.helwan.edu.eg , <sup>4</sup>Amany Abdo@fci.helwan.edu.eg

## ABSTRACT

Object-Oriented System Testing (OOST) focus on issues emerged with Object-Oriented features e.g. encapsulation, polymorphism, inheritance and dynamic binding. Different faults can detect during the interfacing between classes: interface faults, conflicting functions, and missing functions. With iterative nature of testing process, Traditional (automation) testing techniques become less efficient in forecasting new defects resulting in pesticide paradox. To overcome the limitations of traditional inter-class testing techniques, automation testing techniques need to be powered by artificial intelligence for bring dynamic testing techniques into testing process. This paper presents a new dynamic approach to overcome the pesticide paradox in inter-class testing of object-oriented applications that stores the knowledge into ontologies and providing algorithms, which operate on the knowledge to regenerate testing steps easily with required modification to uncover defects. Hence, ontologies can be modified without changing the algorithms, and vice versa omitting using the same test cases to overcome the pesticide paradox. The proposed approach generates an executable test suite in five phases omitting using the same test cases to overcome the pesticide paradox. To validate the proposed approach, a tool entitled PSCCOTM (Polymorphism State Collaboration Class Ontology Test Model) is developed and a case study is applied using PSCCOTM tool. The results show that, test cases can be easily updated by uploading modified ontology file of a test model into PSCCOTM tool. Also, the execution results show high percentage of faults detection, however new cases studies need to be implemented to confirm the attained results.

**Keywords:** *Object-Oriented System Testing, Pesticide Paradox, Inter-Class Testing, Ontologies, Test Cases*

## 1. INTRODUCTION

Building working object-oriented based applications contains major testing phases, namely, intra-method test, inter-method test, intra-class test, and inter-class test. Among all these forms of testing, inter-class test may be the most costly and the most important [1]. The cost of inter-class test may be 50 –70% of the cost of the entire testing activity [1]. An empirical study stated that 39% of the faults uncovered in the applications examined were interface errors [2]. inter-class test can be defined as " a systematic technique for combining a software system while executing tests to discover errors associated with interfacing". Inter-class test aims to find faults in how one class uses the implemented interface of another class. As the classes' interfaces increase in size, the chains of testing are being grown in number, length, and

complexity [3]. Mainly, the testing phase consists of three steps: (i) Test Case creation, (ii) Test Case Execution, and (iii) Test Case Evaluation [4]. The test case creation step represents a vital step among the three steps to overcome the pesticide paradox. IEEE Standard 829 (1983) defines test case as follows: "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement" [3].

In software testing techniques "pesticide paradox" is a term introduced by the famous author Dr. Boris Beizer in the year 1990 [5]. He has framed the term pesticide paradox in software testing phases as: "a residue of cluster of bugs is left behind by each method that a person uses to prevent against the testing methods that are ineffectual".

Singh (2013) mentioned seven principles of testing which must be considered while testing a system. The last one is the pesticide paradox asserted that; if the same test cases are used repeatedly then the ability of forecasting defects can be decreased. Therefore, testers must update and check their test cases on continual basis [6]. Chaudhary (2015) has stated that developers should be careful about places where testers found more defects. Hence; executing the same test cases will not help find more defects. The test suit needs to be updated to manipulate different areas of the software [7].

Several researchers believed that the techniques for automation of test case generation resolve the pesticide paradox by maintaining test cases for an efficient testing process. By reviewing related work in software testing, most of the well-known automation testing techniques for test case generation encounter pesticide paradox. The testing tools execute the same every time, and for maintaining test cases updated require great intervention of the testers. A real challenge facing automation testing techniques is that the functionality of software alters over time according to customers' requests. Fine-tuning the testing tool is a hard task, as it executes parallel with generation of test model and for maintaining test cases updates to overcome pesticide paradox, test tool needs to be refactored. At last, the inter-class testing techniques companied by Artificial intelligence (AI) and machine learning techniques will bring dynamic testing approach to the business environment enhancing the need for refactoring the testing tool. To overcome such difficulties, new approach and solution need to be proposed. Therefore, the objective of this research is developing a dynamic approach depending on semantic software engineering. Examining how the ontology enabled semantic technologies improve the reusability, sharing and extensibility of software development tasks for overcoming pesticide paradox contradiction. In this work, the proposed approach utilizes knowledge engineering techniques to separate the testing phase into two tasks; (1) the identification of what requirements to be tested, (2) the generation of test paths algorithms to manipulate the ontology file. This separation enables test experts to extend the test model according to system modifications and identify new coverage criteria. The new approach aims to overcome the limitations of the previous attempts regarding overcoming the pesticide paradox by enriching testing process with new testing phase omitting using the same test cases. It would facilitate updating test cases of the system under

test using RDF/XML file of the PSCCOTM ontology with minimum user interaction. PSCCOTM approach aims to : ( 1) improve the quality of test cases by generating test model that reflect complete picture of the system under test. (2) Maximize the automation level through automatically analyzing ontology files of the test model to extract instances of message association class where test cases will be generated.(3) electronically elimination duplication of test cases(4) Define various coverage criteria based on PSCCOTM ontology that have high percentage of faults detection while keeping the testing cost within the project budget. (5) Improve the execution of test cases through identifying new class in the proposed ontology that contains the expected results of test cases. The remainder of this paper is organized as follows: Section 2 presents a brief survey of the related works in the areas of automated test case generation; Section 3 presents the proposed approach to overcome pesticide paradox, including a discussion of the proposed phases of the approach; Section 4 describes the prototype tool to automate proposed approach; Section 5 presents a case study to evaluate the efficiency of the proposed technique; and finally Section 6 concludes the results.

## 2. BACKGROUND AND RELATED WORK

Several researchers asserted that the automation techniques of test case generation reduce the pesticide paradox contradiction and detect any other faults easily. Also, it helps to maintain test cases and enhances the accuracy of software testing techniques [8], [9]. Is the automation of test case generation sufficient to eliminate the pesticide paradox in inter-class testing of the object-oriented applications [10]. This is the question that should be answered through this section.

In this section, automated test case generation methods for object oriented testing will be discussed in details, classifying them on their background techniques. An enormous amount of works target, test inputs generation, test scenarios selections, and test oracles generation from formal models and specification. Qiu Zhipeng et al (2021) proposed a test case generation method for embedded software controlling. The proposed method generates formal requirements model to analysis the error type of the test case by defining constraint path from the input variable to the output variable of the defined model [11]. Another contextual demand-based test case generation (TCG) approach for object oriented (OO) systems

is proposed by Rajvir Singh et al (2019) to optimize selection of test cases by applying optimization algorithms [12].

UML activity diagrams are used for testing information and generating test cases. RANbunathan, and ABasu (2019) proposed using pairwise testing and genetic algorithm to derive a reduced number of test cases in activity diagram with concurrent activities [13]. Swadhin Kumar Barisal et al (2019) proposed generating java code from XSD (“XML Schema Definition”) of activity diagram to generate test cases based on concolic testing. Then, the generated test cases and derived Java source code were inserted into COPECA tool (COverage PErcentage CALculator) to calculate MC/DC (Modified Condition/Decision Coverage) score [14]. Another automatic-based testing technique (ATCG) is proposed by Arvinder Kaur et al (2018) that utilizes UML collaboration diagram to generate test cases. An algorithm has been introduced for generation of graphs from collaboration diagrams ensuring full path coverage. By traversal the graph, test cases are generated, restrict the path selection to minimal and avoid duplicate or unbounded path selection [15]. Using hybrid solutions, Shah et al (2019) have proposed a methodology to generate test cases from class and sequence diagrams. A survey has been conducted in this paper to evaluate the proposed framework [16]. Another approach for integration testing is proposed by Yi Sun et al. (2019) based on collaboration diagram and logic contracts, an intermediate model called execution tree of components built as component specification, then test cases are automatically generated through contract solving technology [17]. The literature shows that (1) A model-based testing approach mainly concentrate on a small boundary of the system, a model of the complete system behavior is not often exist and likewise, an overall evaluation of the system using model-based testing is missing. (2) Only few researchers execute an abstract test case generation step that can be used for generating generic and reusable test cases, which are unable to overcome the pesticide paradox.

Another testing approach depends on generating random sampling from the input space of the program under test [18]. Adaptive random testing (ART) has been proposed by Chen et al. (2017) to improve fault-detection effectiveness of random testing by evenly spreading random test inputs across the input domain. ART makes use of distance measurements between consecutive inputs

[19]. To overcome the problems of previous tools which are not dealing with objects and methods of multiple classes. Jinfu Chen et al (2017) proposed a more generic distance metric known as, the object and method invocation sequence similarity (OMISS) metric, which facilitates integration testing of OOS [19]. The overhead caused by the computation of the distance metric makes ART less effective than pure-random approaches, questioning its practical effectiveness [20]. Marko Dim a sevi et al. (2018) proposed a hybrid approach that integrates dynamic symbolic execution and feedback-directed random testing into an algorithm for automatic testing of object-oriented software. The main limitation of the proposed approach is the non-applicability for integration testing [21]. Hanyu Pei et al (2019) have compared the performance of DRT through a more comprehensive study compared with previous works, in which more metrics are adopted in the experiments [22]. In the final, an absolute disadvantage of random testing is that randomly generated test cases are in general difficult to interpret; consequently, a considerable effort is required to understand them and to write meaningful oracles [23]. For deriving testing execution to specific code blocks, a symbolic execution analysis approach is proposed [24]. The main limitation of traditional symbolic execution often leads to an exponential number of paths those result in constraint solver termination [25]. Another approach is proposed for overcoming the limitations of symbolic execution and test automation is Dynamic Symbolic Execution (DSE) [26]. DSE techniques for object-oriented systems have been implemented in tools like jCUTE (Java) [27] and Pex (.NET) [28]. Also to overcome the limitations of symbolic execution is combined with fuzz testing [29]. Symbolic fuzzing framework using S2E symbolic execution engine to quickly reach more code areas without getting lost in a large execution tree proposed by Chao-Chun Yeh et al (2015) [29].

To limit path explosion in hybrid testing, Bin Zhang et al (2018) proposed a novel Lazy concretization of the symbolic pointer (LCSP) to operate states forked from symbolic pointers [30]. Search heuristic techniques combined in dynamic symbolic execution to reduce path explosion [31]. Sooyoung Cha et al (2019) proposed a new approach for dynamic symbolic execution. It combines a parametric search heuristic and a learning algorithm for finding good parameter values [31]. Sooyoung Cha et al (2021) in [32] presented a technique to generate an algorithm that

efficiently finds an optimal heuristic to overcome the limitations of manually generating search heuristics. The main problem of Symbolic execution, is the shortage of processing real-world code, especially, assessing path feasibility and explosion (path constraint cannot be solved). Researchers attempt to mitigate these problems by leveraging dynamic symbolic execution with other techniques, such as search heuristic, machine learning. Hybrid testing can be seen as an instance of the general framework of search-based software testing/engineering [33]. Search-based testing approaches examine testing process as a searching problem by implementing meta-heuristic search algorithms for test cases generation [33]. Sina Shamshiri et al. (2017) compared in their study between the efficiency of evolutionary algorithms (including a genetic algorithm and chemical reaction optimization) and random search techniques in unit test suites generation. The study asserted that the difference between the two techniques is not large [34]. Snehlata Sheoran et al (2019) proposed an artificial bee colony algorithm to discover and prioritize the definition-use paths in code-based testing which are not definition-clear paths [35]. Madhumita Panda et al. (2020) proposed a hybrid FA-DE framework complete transition path coverage; using UML behavioural state chart model along with the hybrid Firefly algorithm (FA) and Differential Algorithm (DE) [36]. Implementing search-based testing techniques in integration tests needs great efforts to solve several issues, such as the combinatorial explosion of conditions or pre-condition failures [37]. An automated test case generation presents the main pesticide paradox; it performs the same every time. Given the iterative nature of system development, the growth of systems leads to test case generation to take place multiple times during a system development project. Automated test case generation can be improving the confidence hazards and repeat tests many times. But, to perform automated test case generation precisely; needs assuring that a sufficient combination of human testing is involved in testing process. The tester faces difficult to maintain brittle scripts, test data and test frameworks that requires updates frequently when the software under test alters.

Few researches have studied in the direction of developing ontologies to improve software testing phase. Josip Bozic et al (2021) proposed ontologies based web testing approach that combines knowledge about common attacks and the system under test. The proposed approach depends on transforms ontologies into input models to generate abstract test cases that can be converted to concrete test cases [38]. Franz Wotawa et al (2020) proposed environment ontology based testing by converting ontologies into input models and using a combinatorial testing algorithm for deducing the test cases [39].

So, there has been no specific study that focused on developing an automated dynamic test case generation approach for overcoming pesticide paradox in inter-class testing based on ontology building. Verma et al (2010) describe how a collection of semantic models may help to automate steps in the development process. By defining semantic representation of knowledge in ontology, tools used in different phases can communicate knowledge across phases [40]. This Research proposes using ontology to annotate the Test Model with semantic information. In addition, the ontology of application domain and the system behavior can support a smarter retrieval of test cases based on this semantic information.

### 3. THE PSCCOTM APPROACH

The testing phase is an iterative process of tasks, debugging, modifying program code and, testing again. The laborious process of testing object-oriented applications was motivated to develop a new approach to improve pesticide paradox in testing applications. In this work, a new set of testing phases are developed that help the user to regenerate the steps easily with the required modification to uncover a defect. It generates an executable test suite in five phases. Figure 1 illustrates the phases of the proposed approach, and their inputs and outputs.

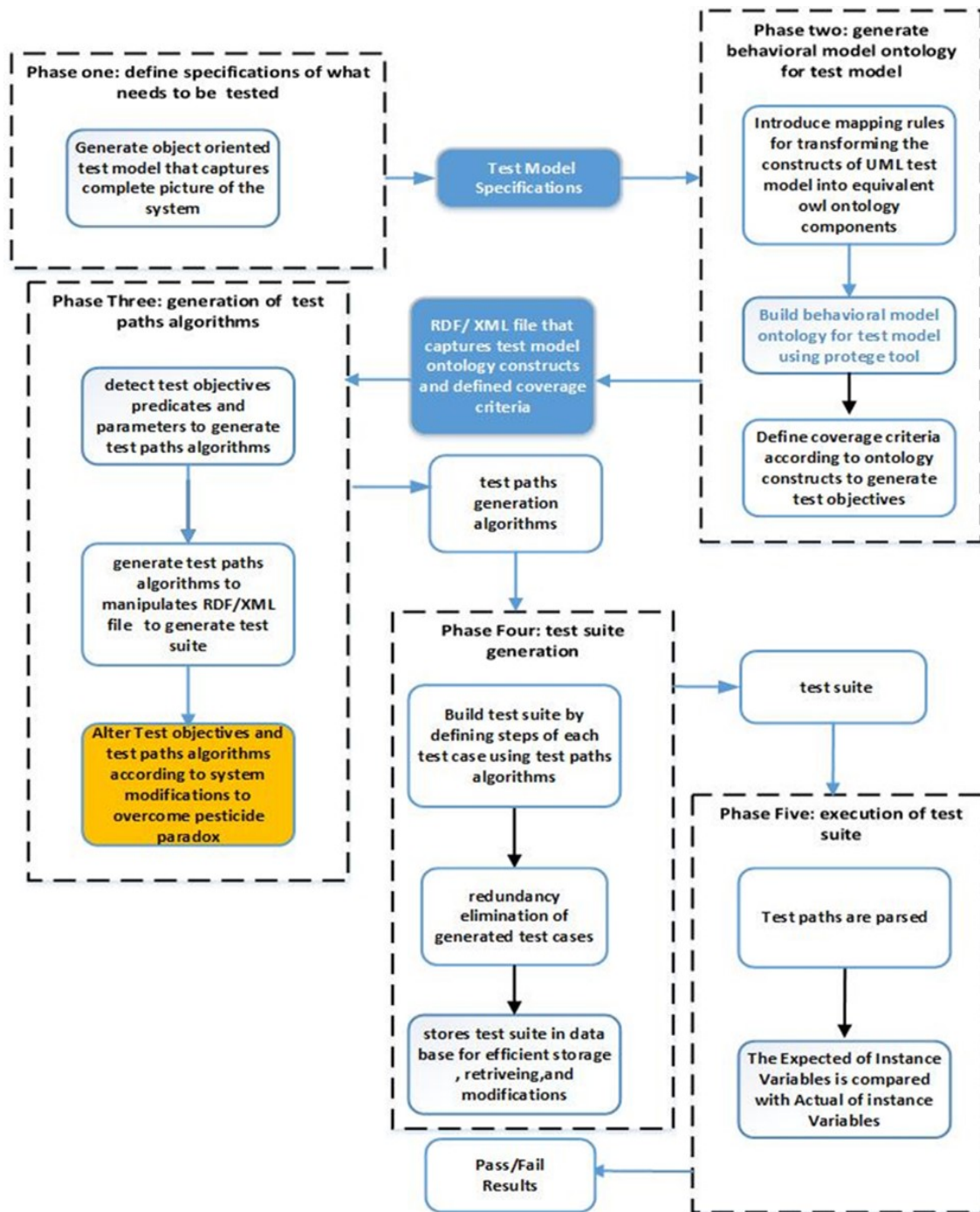
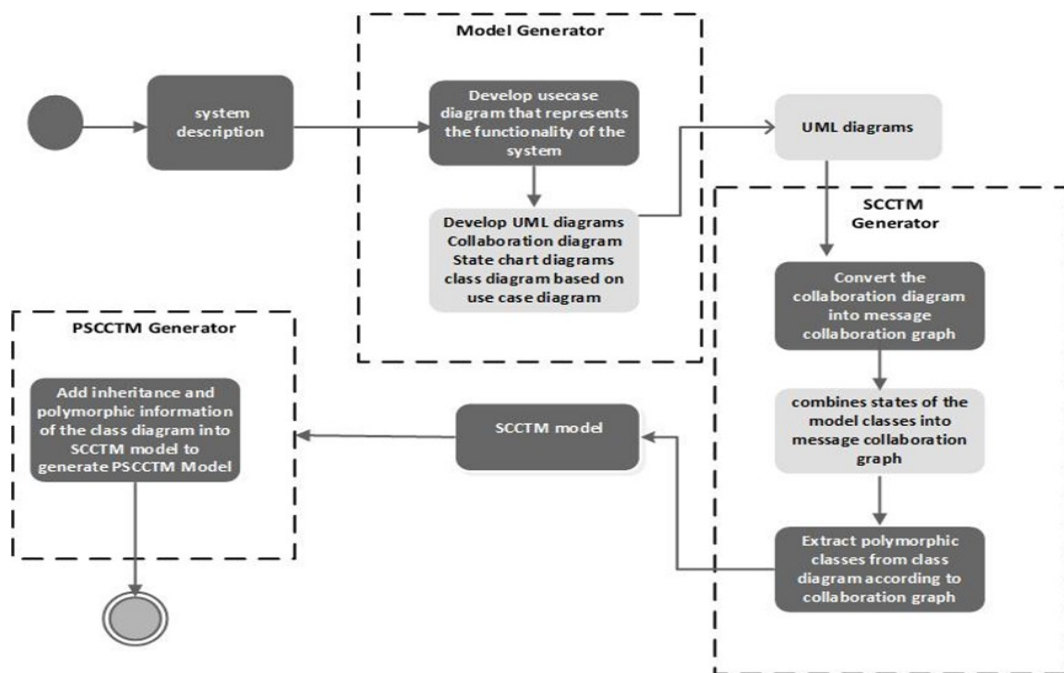


Figure 1: Phases of the Proposed Approach

### 3.1 Generation of Behavioral PSCCTM test model for Object –Oriented Software

The first phase in the approach is an attempt to develop a standard modeling technique to generate the test model for any software implementing Object-Oriented characteristics. The interactions between collaborating objects need to be tested to ensure the correct functionality of the system. The proposed test model includes information about interactions among objects, state transitions within objects, inheritance relationships, and polymorphic methods under system testing. Therefore, the proposed test model will combine four diagrams in order to reveal different kinds of information, which is provided by each diagram. It uses a use case diagram, a collaboration diagram, a class diagram, and statechart diagrams. In previous work, the augmentation of collaboration diagram with objects states was already done [41]. This research proposes the calling of a class diagram of a system to capture inheritance relations and polymorphic methods for a complete and coherent description of the system. In the first step for the construction test model, the collaboration diagram will be transformed into a message collaboration graph according to the sequence of messages.

represents an object and each arrow represents a message. Each object in the test model can be represented as a modal vertex or non-modal vertex according to the states of the object. If the object has only one state, the object will be represented as the vertex of the non-modal class in the test model. Vice versa, if the object has many states in the Statechart diagram, the object, in this case, will be represented as the vertex of modal class in the test model. In the second step, the Statechart diagrams will be added to the test model according to its corresponding objects in the collaboration graph. In the third step, abstract classes are extracted from the class diagram according to involved classes in the collaboration graph. The vertex in the test model will be extended according to the inheritance relationship described in the class diagram. Figure 2 shows the framework of the general procedure to create the test model. Before the test model construction, it is assumed that all UML diagrams are consistent.



The graph begins with a null vertex that models an external message. The vertex in the test model

Figure 2: Framework of Proposed Test Model for Object Oriented Software Testing.

### 3.2 Building PSCCOTM Behavioral Model Ontology of Test Model

In this phase, behavioral model ontology is built for the ontological representation of the PSCCOTM test model of the system under test and saved in RDF/XML file format. Ontology absolutely defines the concept in a field, the characteristics of properties, attributes, as well as specific constraints related to the described concept [42]. Based on the domain knowledge defined by the ontology, coverage criteria are formalized to generate appropriate test objectives for test cases generation.

The notion for mapping between UML and Owl must be formalized. The proposed notation is described as follows:

- In the UML, the class is represented as U(C), the attribute is represented as U (A), and the relation is represented as U(R).
- In OWL the class is represented as O(C), the data type property is represented as O (A), and object property between classes is represented as O (E1OE2).

Based on the Ontology Definition Metamodel (ODM) [41], which is a specification adopted by the OMG, the test model elements are mapped to the OWL elements. Table 1 summarizes an overview of transformations rules for mappings the UML test model representation into the OWL representation.

Table 1: Mapping of the Test Model Elements into OWL Ontology Elements

The Test Model structure Element	Representation in OWL
Class in test model	transformed into OWL class respectively
The Abstract Class component represents the class that has the inheritance relationship and polymorphic methods.	Add class O(C) and class O(Ci) which is a subclass of class O(C).
Attribute of the class	Transformed into Data Type Property of the Owl class
Message Details as simple attributes	The message details are transformed in OWL Ontology as the attributes of the message Association class using data type property.
states and state transitions of each model class	Treat states and state transitions of each model class as separate classes and connect them with the base model class through

The Test Model structure Element	Representation in OWL
	the "Object Property" axiom.
Message links in PSCCTM test model	Considering the Message link as association class with attributes:- <ul style="list-style-type: none"> <li>• An OWL class (named message association class) with instances of the class for each message.</li> <li>• Object property chains between the different classes connected to the association class</li> </ul>

#### 3.2.1. The defined coverage criteria for test model ontology

Based on PSCCOTM ontology construction various coverage criteria are proposed for test paths generation:-

- **Message association coverage criterion:** This criterion ensures that each instance of message association class in RDF/XML file is tested once. It can be used only to check if the interactions between classes are taking place correctly.
- **All-state transition class coverage:** This criterion will be implemented by traversing all instances for each state transition class in PSCCOTM ontology revealing invalid transitions within state transitions classes. The number of test paths in this criterion is determined by the product of the instances of all state transitions classes. The classes that have an inheritance relationship will be seen as one vertex in the PSCCTM model, as child classes of abstract class will have the same transitions.
- **All transitions and generalization coverage criterion:** This criterion will be implemented by traversing the all instances for each state transition class in child classes of PSCCOTM ontology thus ensuring that all child classes and all state transitions are tested at least once. This criterion is used to reveal invalid transitions within inheritance relations. The number of test paths in this criterion is decided by the product of maximum number of child classes for abstract classes by the number of test paths for All-State Transition Instances Coverage criterion.

### 3.3 Generation of Test Paths Algorithms

In this phase, the proposed approach aims at defining a correspondence between test model constructs and the defined coverage criteria. This phase consists of three steps; the first step detects test objectives predicates, parameters based on the constructs of PSCCOTM test model ontology, and defined coverage criteria. The second step generates test paths algorithms for the manipulation of RDF/XML file to generate the test paths. In the second step, the detected test objectives are used as input of the test paths algorithms. In the third step, Test objectives and test paths algorithms can be altered (predicates, parameters, or algorithms) according to system modifications to overcome pesticide paradox without the need to modify all the steps to generate the test suite.

### 3.4 Generation of Test Suite

In this phase, test case is generated for each test objective and added to the partially generated abstract test suite. To decide whether test case is already satisfied by test case generator or not is checked by redundancy elimination operation to avoid duplication of test cases. After that the partially generated abstract test case will be saved in database for efficient storage, retrieving, and modifications. Abstract means that it is implementation-neutral and programming-language-neutral, depending merely on the PSCCOTM ontology constructs. A test case is specified by a list of steps. A Step is described by the attributes of message association class that change the state of the receiver class, and its corresponding state-transition of the Statechart diagram, as illustrated in figure 3.

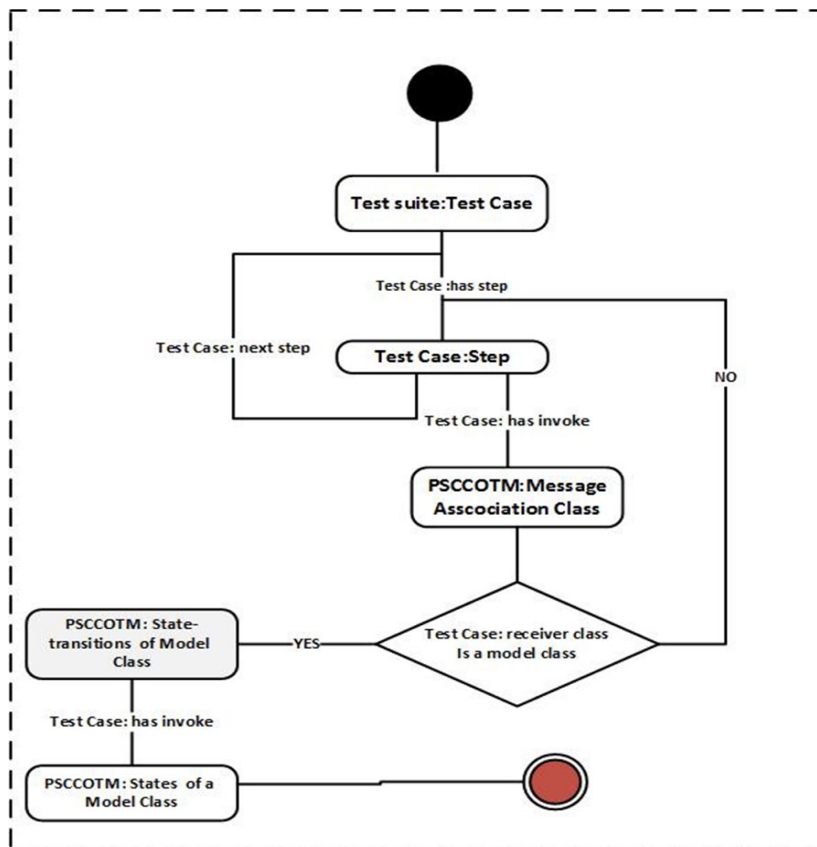


Figure 3: Activity diagram representing steps of test case generation.

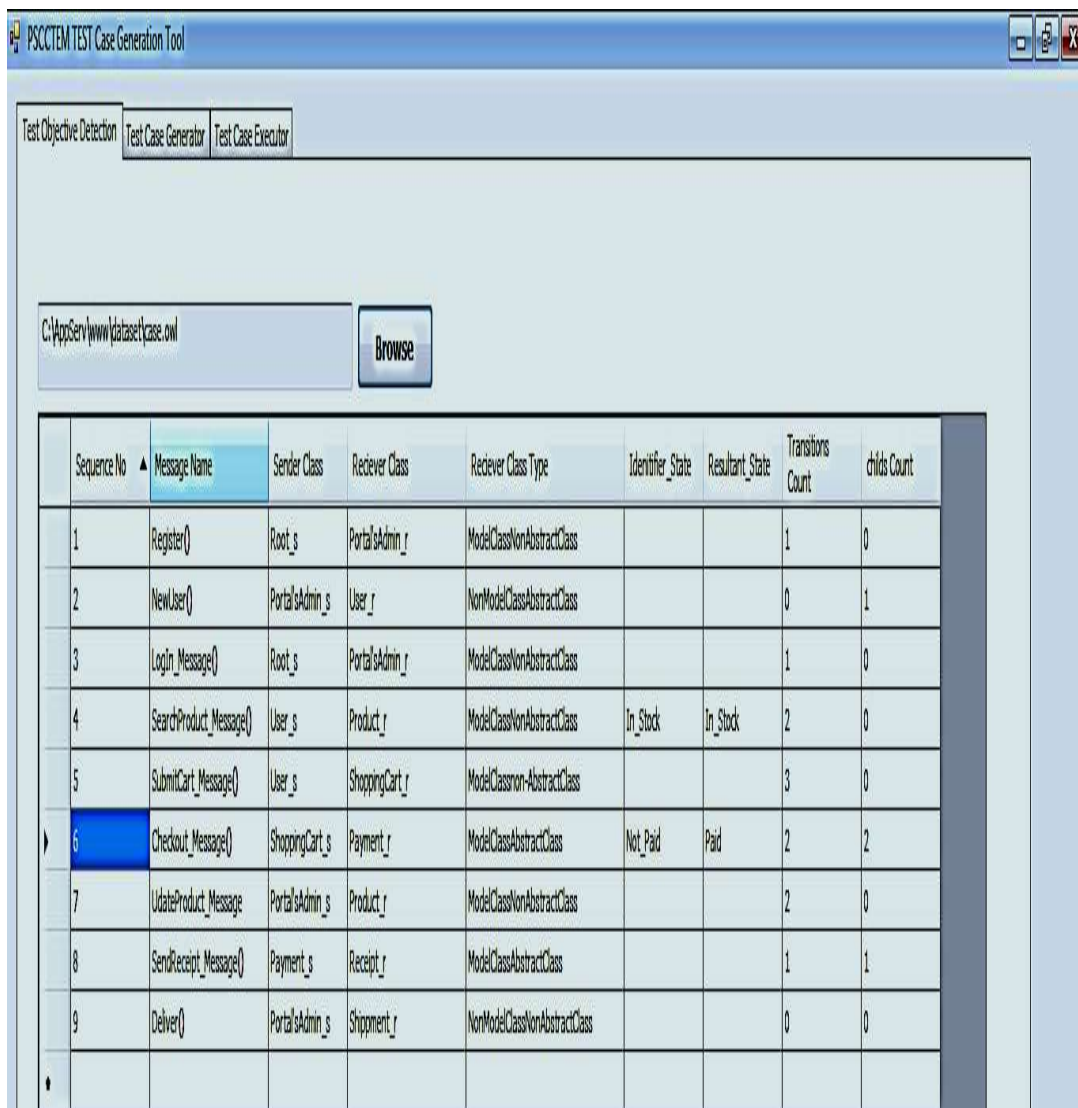


### 3.5 Execution of Test Suite

Test path are parsed by extracting instances of expected outcome class in PSCCOTM ontology to identify the state transitions of model classes and their Expected of Instance Variables. The actual data of execution are extracted from the log file of the system under test. The Expected of Instance Variables is compared with Actual of instance Variables to identify the result of the test case. If any instance variable of any receiver class of model type is not in the required resultant value after execution, the corresponding test case is considered to have failed.

### 4. PSCCOTM IMPLEMENTATION

The PSCCOTM prototype tool was developed based on the ontology-based approach described in the previous section for manipulation of RDF/XML file. The PSCCOTM tool consists of three major modules: (1) Test objective detection, (2) Test case generation, and (3) Test suite Execution. The screenshot in Figure 4 shows the interface of the PSCCOTM tool. The following subsections describe the functionality of this tool.



The screenshot shows the PSCCOTM Case Generation Tool interface. It has three tabs: 'Test Objective Detection', 'Test Case Generator', and 'Test Case Executor'. The 'Test Objective Detection' tab is active. Below the tabs, there is a text input field containing 'C:\AppServ\www\dataset\case.owl' and a 'Browse' button. Below this is a table with the following columns: Sequence No, Message Name, Sender Class, Receiver Class, Receiver Class Type, Identifier\_State, Resultant\_State, Transitions Count, and credits Count. The table contains 9 rows of test cases, with the 6th row highlighted in blue.

Sequence No	Message Name	Sender Class	Receiver Class	Receiver Class Type	Identifier_State	Resultant_State	Transitions Count	credits Count
1	Register()	Root_s	PortalAdmin_r	ModelClassNonAbstractClass			1	0
2	NewUser()	PortalAdmin_s	User_r	NonModelClassAbstractClass			0	1
3	Login_Message()	Root_s	PortalAdmin_r	ModelClassNonAbstractClass			1	0
4	SearchProduct_Message()	User_s	Product_r	ModelClassNonAbstractClass	In_Stock	In_Stock	2	0
5	SubmitCart_Message()	User_s	ShoppingCart_r	ModelClassnon-AbstractClass			3	0
6	Checkout_Message()	ShoppingCart_s	Payment_r	ModelClassAbstractClass	Not_Paid	Paid	2	2
7	UpdateProduct_Message	PortalAdmin_s	Product_r	ModelClassNonAbstractClass			2	0
8	SendReceipt_Message()	Payment_s	Receipt_r	ModelClassAbstractClass			1	1
9	Deliver()	PortalAdmin_s	Shipment_r	NonModelClassNonAbstractClass			0	0

Figure 4: Screenshot of PSCCOTM tool.

#### 4.1 Test Objective Detection Module

Since instances of message association class are the main ontology constructs in the test

case generation process. The PSCCOTM tool gave them great attention to decide the sequence of extracting them from the RDF/XML file. The PSCCOTM tool relied on the system.XML library in C sharp language to detect test objectives. The process begin with creating an object of XML document class to store the nodes of the ontology file, after that, the RDF/XML of the PSCCOTM ontology is loaded using the load method in the XML library. Using the name of the node method, the nodes of the document are searched and extracted which are stored in the following properties:-

Sequence\_Number, Message\_Name, Sender\_Class, Receiver\_ClassName, Receiver\_ClassType, Transition\_Name, Transition\_State From, Transition\_state\_to. B.

Test case Generation Module

#### 4.2.1. Message association coverage algorithm

The routine takes RDF/XML file of PSCCOTM ontology and detected test objective as an input and returns single test path that ensures that each instance message of message association class in PSCCOTM ontology extracted once. In the algorithm below, Lines 5 to 23 generate a test path by extracting all instances of message association class using two methods:-

- Child node. Name method to fetch the name of node.
- Child node.InnerText to fetch the value of the node.

## 4.2 Test Case Generation Module

This module mainly is concerned with building message expression for each test case step. Test\_Case\_Generator module has three main classes to cover detected test objectives: message association class, state transition class, Generalization transition class. Based on the sequence of interactions in the PSCCOTM test model, instances of message association class will be extracted including state transitions for each receiver class of a type model class.

A test path generation algorithms for detected test objectives are discussed in the following subsections:-

Transition states of modal classes are picked up by calling two functions:-

- **GetTransitionStateFrom(transition\_name):**To capture the source state of state transition.
- **GetTransitionStateTo(transition\_name):**To capture the destination state of state transition.

The pseudo code below shows the algorithm for Message-Association Coverage.

---

#### Algorithm 1: test path generation for Message Association Coverage criterion

---

```

1   Input R: RDF/XML file
2   Output MASEq: Single Test Path(MASEq)

3   Declare MPSeq: A sequence (OCL 1.5) of message properties
   in R(Sequence_number, Message_Name, sender_class, receiver_class_name,
   receiver_class_type, transition_name, transition_value)
   Xml Document doc = new Xml Document;
4   doc.load(RDF/XML file)
5   foreach node in doc
6       If (node. Name=="owl:NamedIndividual(MessageAssociationClass)")
7           foreach (child node in node.Child Nodes)
8               If (child node. Name == "Receiver_Class")
9                   Receiver_Class_Name = child node.InnerText;
10          foreach (receiver node in doc.Child Nodes)
11              if (receiver node.Name == Receiver_Class_Name)
12                  Receiver_class_type = receiver node.ChildNodes[1].InnerText;
13                  else if (child node.Name == "Sequence_number")
14                      Sequence_number = child node.InnerText;
```

---

```

15         else if (child node.Name == "Message_Name")
16             Message_Name = child node.InnerText;
17         else if (child node.Name == "Sender_class")
18             Sender_class = child node.InnerText;
19     If (! (Reciever_class_type. Contains("NonModelClass")))
20         transtion_name = GetTransitions(Reciever_Class_Name)
21         transtion_value = GetTransitionValue(transtion_name)
22         transtion_state_from = GetTransitionStateFrom(transtion_name)
23         transtion_state_to = GetTransitionStateTo(transtion_name)
24         inserAt(test_cases_list,MPSeq)

```

#### 4.2.2. State transition class coverage algorithm

The routine takes RDF/XML file of PSCCOTM ontology as an input and returns a set of test paths as an output. The loop at line 9 executes for number of transition classes in a PSCCTEM ontology that has maximum number of transition instances. The loop in line 11 used to achieve combinations between state transitions instances of model classes. Lines 18 to 26 returns all instances of message association class according to message sequence number.

Lines 27 to 31 complete steps of each test path by fetching state transitions data. The line number 28 retains the data of state transitions of test case in case\_details variable to be used in redundancy elimination operation. The last line in the algorithm stores the generated test paths in database table. The processed algorithm is shown in the following.

---

#### Algorithm 2: test paths generation for State Transition Property Coverage criterion

---

```

1   Input  melst: A messages list
2   Output MPSeq: A sequence of test paths
3   Declare MPSeq: A sequence of message properties(Sequence_number, Message_Name,
sender_class, receiver_class_name, receiver_class_type, case_transition, transtion_value,
transtion_state_from, transtion_state_to )
4   Xml Document doc = new Xml Document;
5   Doc.load(RDF/XML file)
6   for (i=1 to MessagesList.Count)
7       if (MessagesList [i].Value> largest_transition_count)
8           largest_transition_count = MessagesList [i].Value
9   for (no = 1 < largest_transition_count + 1)
10      //get state transitions for all receiver classes in test case
11      For (int index = 0 < MessagesList. Count - 1)
12          main_receiver_name = MessagesList [2, index].Value
13          main_receiver_type = MessagesList [3, index].Value
14          main_sequence_number = messageslist [0, index].Value
15          if (!(main_receiver_type.Contains("NonModelClass")))
16              foreach (t in get_transitions(main_receiver_name)
17  // get steps of each test case
18  for (int step = 0 < messageslist.Count)
19      Sequence_number = messageslist[0, step].Value
Message_Name = messageslist [1, step].Value
sender_class = messageslist [2, step].Value
receiver_class_name = messageslist [3, step].Value

```

---

```

receiver_class_type = messageslist [3, step].Value
20   If (!(receiver_class_type.Contains("NonModelClass")))
21     if (receiver_class_name == main_receiver_name
22     && main_sequence_number == Sequence_number)
23       Case_class = main_receiver_name
24       case_trans = t
25       Else
26         case_class = receiver_class_name
27         case_trans=Get_Class_transition (receiver_class_name, no)
28       Else
29         case_class = receiver_class_name
30         case_transition=""
31         transtion_state_from = ""
32         transtion_state_to = ""
33         transtion_value = ""
34     if (!(case_transition == ""))
35       case_details = string.Concat(case_details,":", case_transition)
36       transtion_value = GetTransitionValue(case_transition)
37       transtion_state_from= GetTransitionStateFrom(case_transition)
38       transtion_state_to= GetTransitionStateTo(case_transition)
39     insert At(test_case_table, MPSeq)

```

#### 4.2.3. All transitions and generalization class coverage algorithm

The routine takes RDF/XML file of PSCCOTM ontology as an input and returns a set of test paths as an output. The condition at line 11 gets largest number of child classes for abstract classes in PSCCOTM ontology, according to that number, the number of test paths identified. Lines 35 to 40 fetch child class of receiver classes to be calling in test case. The line of number 44 retains the data of test case transitions in case\_details variable to be used in redundancy elimination operation.

The last line in the algorithm stores the generated test paths in database table. The processed algorithm is shown in the following.

---

#### Algorithm 3: test paths generation for All Transitions and generalization Property Coverage criterion

---

- 1 **Input** melst: A messages list
  - 2 **Output** MPSeq: A sequence of test paths
  - 3 **Declare** MPSeq: A sequence of message properties(Sequence\_number, Message\_Name, sender\_class, receiver\_class\_name, receiver\_class\_type, case\_transition, transtion\_value, transtion\_state\_from, transtion\_state\_to )
  - 4 **Xml Document** doc = new Xml Document;
  - 5 **Doc.load**(RDF/XML file)
  - 6 **for** (i=1 to MessagesList.Count)
-

```

7   if (MessagesList [i].Value > largest_transition_count)
8       largest_transition_count = MessagesList [i].Value
9       for (i=1 to MessagesList.Count)
10          if (MessagesList [i].Value > largest_childs_count)
11              largest_childs_count = MessagesList [i].Value
12              for (int child_index = 0 < largest_childs_count)
13                  for (no = 1 < largest_transition_count + 1)
14                      //get state transitions for all receiver classes in test case
15                      for (int index = 0 < MessagesList.Count - 1)
16                          main_receiver_name = MessagesList [2, index].Value
17                          main_receiver_type = MessagesList [3, index].Value
18                          main_sequence_number = messageslist [0, index].Value
19                          if (!(main_receiver_type.Contains("NonModelClass")))
20                              foreach (t in get_transitions(main_receiver_name))
21                                  // get steps of each test case
22                                  for (int step = 0 < messageslist.Count)
23                                      Sequence_number = messageslist[0, step].Value
24                                      Message_Name = messageslist [1, step].Value
25                                      sender_class = messageslist [2, step].Value
26                                      receiver_class_name = messageslist [3, step].Value
27                                      receiver_class_type = messageslist [3, step].Value
28                                      If (!(receiver_class_type.Contains("NonModelClass")))
29                                          if (receiver_class_name == main_receiver_name
30                                              && main_sequence_number == Sequence_number)
31                                              Case_class = main_receiver_name
32                                              case_trans = t
33                                      Else
34                                          case_class = receiver_class_name
35                                          case_trans = Get_Class_transition (receiver_class_name, no)
36                                      If (!(receiver_class_type.Contains("NonAbstractClass")))
37                                          if (get_child(case_class).Count <= child_index)
38                                              case_class = get_child(case_class)[child_index - 1]
39                                          Else
40                                              case_class = get_child(case_class)[child_index]
41                                      Else
42                                          case_class = receiver_class_name
43                                          case_transition = ""
44                                          transtion_state_from = ""
45                                          transtion_state_to = ""
46                                          transtion_value = ""
47                                      if (!(case_transition == ""))
48                                          case_details = string.Concat(case_details, ":", case_class + case_transition)
49                                          transtion_value = GetTransitionValue(case_transition)
50                                          transtion_state_from = GetTransitionStateFrom(case_transition)
51                                          transtion_state_to = GetTransitionStateTo(case_transition)
52                                      insert At(test_case_table, MPSeq)

```

#### 4.2.4. Redundancy elimination operation

A redundancy elimination operation uses clear duplicate () function to check whether test cases are already satisfied by test case generator or not. A test case is preserved by the test suite if a test case that satisfies the test objective already holds in test cases table. The parameters of generated test cases are hold in test cases table as case\_details. The case\_details of the test case are defined by the attributes of case\_class to retain the name of child class, and case\_transition to retain the name of state transitions. Using

check\_case () method to check whether if the case\_details already exist in test cases table or not. After the test case is generated the check\_case method is called by passing case\_details parameter. The Boolean flag case\_exist will return true, if case\_details exists otherwise will return false. The figure 5 illustrates the flowchart of redundancy elimination operation.

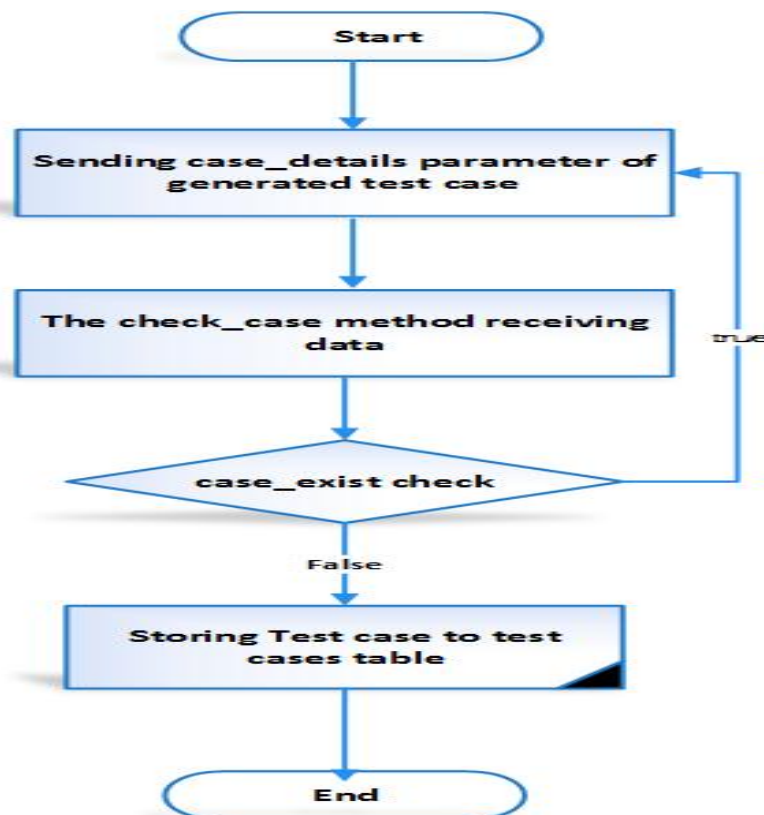


Figure 5: Flowchart of Redundancy Elimination Function

#### 4.3 Test Suite Execution Module

In test executor tab, list of test cases IDS appeared based on selected coverage criterion, as illustrated in figure (6). According to selected test case, a concert test case steps will be loaded with expected values of instance variables loaded from Expected\_Outputs class in PSCCOTM ontology.

The Test Executor executes concrete test cases by filling the test data in the function calls of test paths. Each test case is then executed on the implementation and the execution results are logged in the file to read by PSCCOTM tool for comparing the results of a test run with the expected results.

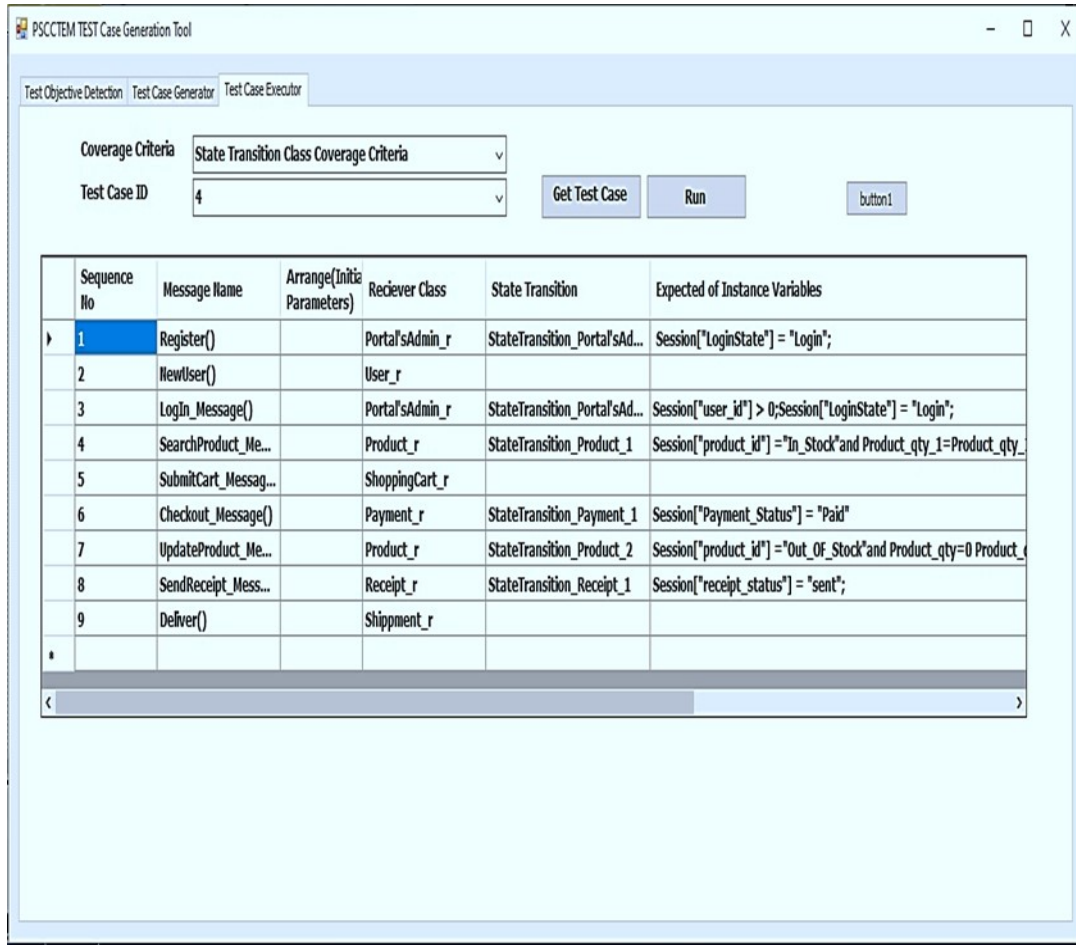


Figure 6: GUI for Test Executor with Case Study of Type State Transitions Property Selected.

## 5. EVALUATION

This section introduces an evaluation of the PSCCOTM approach based on implemented case study. First, the PSCCOTM test model, ontology, and test suite will be generated for the system under test. Second, the initial evaluation of the approach will depend on executing mutation testing to measure the efficiency of the test cases in actual testing of software, and then the approach is compared with test method in [13] to evaluate the fault detection ability of the approach. In the final section the importance of the proposed approach is indicated.

### 5.1. The Case Study

In this section, the PSCCOTM test model and ontology of online shopping portal will be implemented based on the following scenario:

1. Register an account for the customer by site's administrator.
2. Create a new member for the customer.
3. Log in to the portal system by the customer.
4. The member after login can search catalogue for the items.
5. The customer can edit and submit the final cart.
6. Check out and make a payment.
7. The site's administrator can update different items in the product.
8. The system will Send a hard copy receipt or email a soft copy receipt to the customer.
9. Deliver the item to the customer.

#### 5.1.1 Test model for online shopping portal

The PSCCOTM test model of the Online Shopping Portal is generated from collaboration graph and Statechart diagrams.

Then, inheritance information of the corresponding classes in the collaboration graph is extracted from class diagram to be used in creating PSCCOTM test model. In this step the

test model is augmented with inheritance and polymorphic information captured from class diagram, as shown in figure (7).

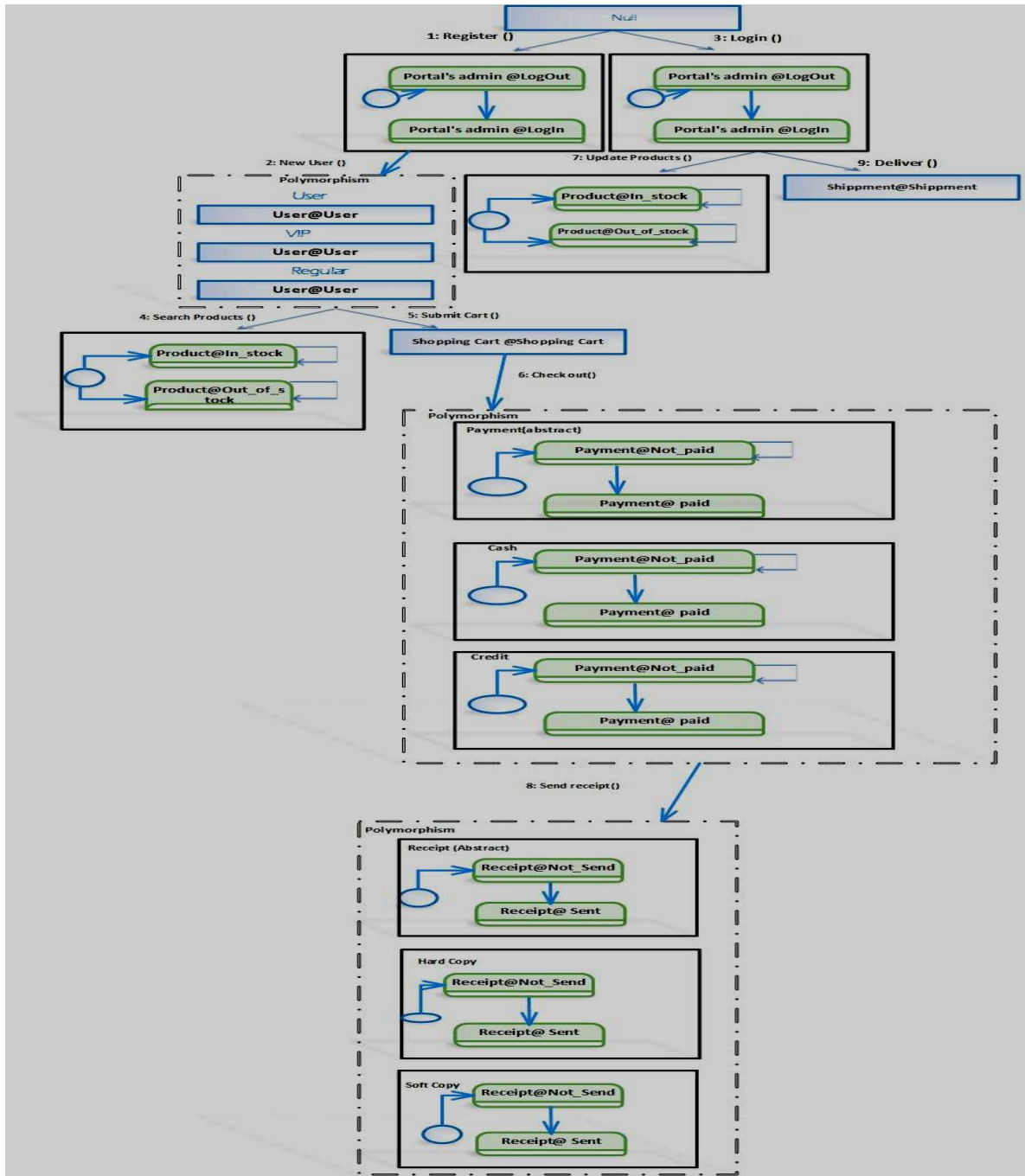


Figure 7: PSCCOTM Test Model for Online shopping Portal



### 5.1.2 PSCCOTM ontology for online shopping portal Online

Shopping portal Ontology defined as classes, subclasses; Relationship between classes implemented using web interface of Protégé. The structure of the Online shopping portal ontology is depicted as follow:-

- Base class named Thing contains all classes.
- Sibling class named sender class contains classes that send the messages.
- Sibling class named receiver class contains classes that receive the messages.

- Sibling class named message association class contains the message expression attributes of the relation between classes.
- Sibling class named state class contains model classes with their states.
- Sibling class named transition class contains state transitions of model classes. Figure (8) illustrates the structure of the online shopping portal.

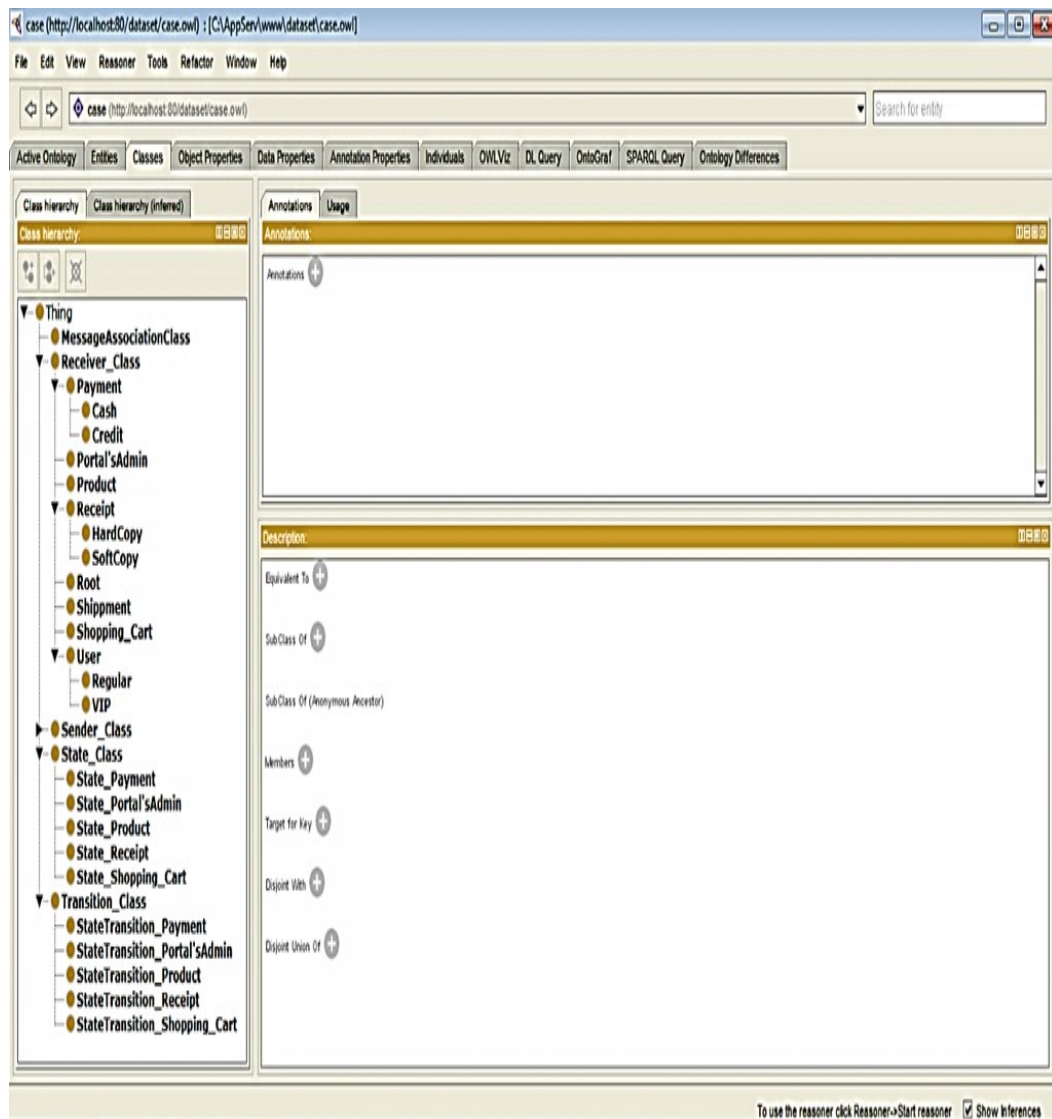


Figure 8: Class hierarchy of PSCCOTM ontology for Online shopping Portal.

To validate the semantic consistency of the generated ontology, graphical representation of the portal ontology obtained using the plugin OntoGraf protégé to show the hierarchy of the test model. As shown in figure(9), there is object property called send message, the domain of the property is sender class, and the range of the property is message association class. Another object property is receiving message, the domain of the property is message association class and the range is receiver class. Also, there is object property with name has state between receiver class and state class, the domain of the property is receiver class, and the range is state class. Finally, there are two object properties with name from and to between state class and state transition class to represent the source and destination states of state transition.

Also, in the figure there is individual of message association class called update product message with two object properties; send message and receive message. The domain of the send message object property is the instance of the portal'sAdmin class as sender class and the range of the receive message is the instance of product class as receiver class. Table 2 shows the number of state instances and transition instances for shopping Portal Ontology.

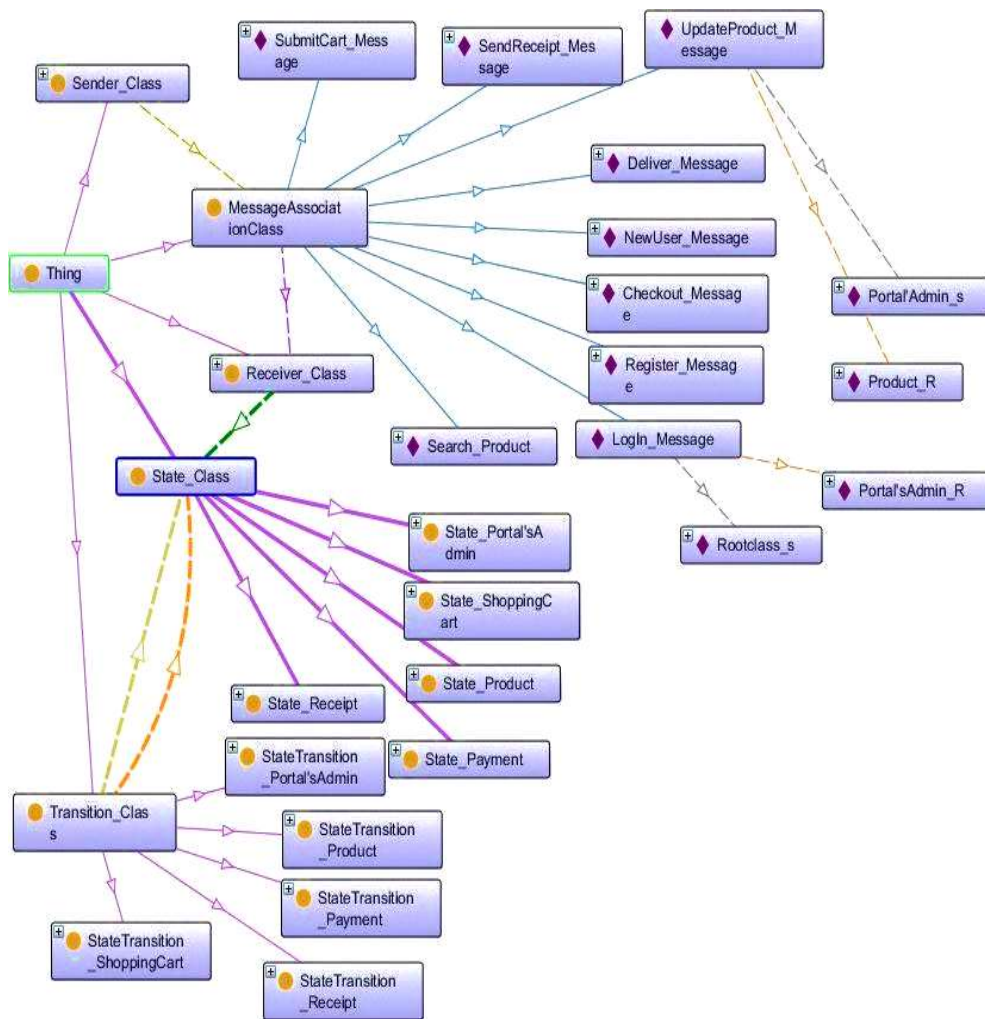


Figure 9: Graphical Representation of Online Shopping Portal Ontology.

Table 2: State Instances and Transition Instances for Portal Ontology.

Class Name	Structure of Portal Ontology			
	Child Classes	Instances of Message Association Class	Number of state instances	Number of transition instances
Portal's Admin	-----	register() log-in ()	2 2	1 1
Portal's User	VIP Regular	New User()	1	1
Product	-----	Search Item() Remove Item()	2 2	2 2
Shopping Cart	-----	Submit Cart()	1	1
Payment	Credit PayPal	Check Out ()	2 2	2 2
Receipt	Hard Copy Soft Copy	Send Receipt()	2 2	1 1
Shipment	-----	deliver ()	1	1

5.1.2.1 Defined coverage criteria for online shopping portal ontology

Based on PSCCOTM ontology constructs shown in table 2 numbers of coverage criteria are formalized to perform the needed level of testing.

- Message Association Coverage Criterion:** This coverage criterion generates single test path from RDF/XML file. This criterion ensures that each message in an end-to-end sequence of messages in collaboration is tested once. However, this is the weakest coverage criterion and can be used only to check if the interactions between classes are taking place correctly.
- All State Transition Instances Coverage Criterion:** This criterion will be implemented by traversing all instances of each state transition class in shopping portal ontology, by taking in the account the probability of combinations between state transitions of model classes. The number of test paths in this criterion is determined by the product of the instances of all state transitions classes. The inheritance relationship will be treated as one vertex in the PSCCOTM ontology. Therefore, there are  $1*1*1* 2*2*1 * 2 *1*1=8$  test paths for this criterion.
- All Inheritance and All transitions Coverage Criterion:** This criterion will be implemented by traversing all instances of transition classes and the all child classes of abstract classes in the portal ontology thus asserts that all child classes and all state transition

instances are traversed at least once. The number of test paths in this criterion is calculated by the product of maximum number of child classes for inheritance relation by the number of state transitions instances for state transition classes in the PSCCOTM ontology. The maximum number of child classes are two, and number of the product of all instances of state transitions are eight. Therefore, there are  $2*8 =16$  test paths for this criterion.

5.1.3 Test suite generation

The PSCCOTM tool takes the PSCCOTM ontology of the online shopping portal as an input, while the output is a set of test paths for detected test objectives. Examples of Generated Test Paths for these objectives:

5.1.3.1 The Message Association Coverage

This coverage will produce only test path to ensure the correctness of the interactions between classes regardless of the state transitions of objects and generalization relations, GUI for the generated test path illustrated in figure 10.

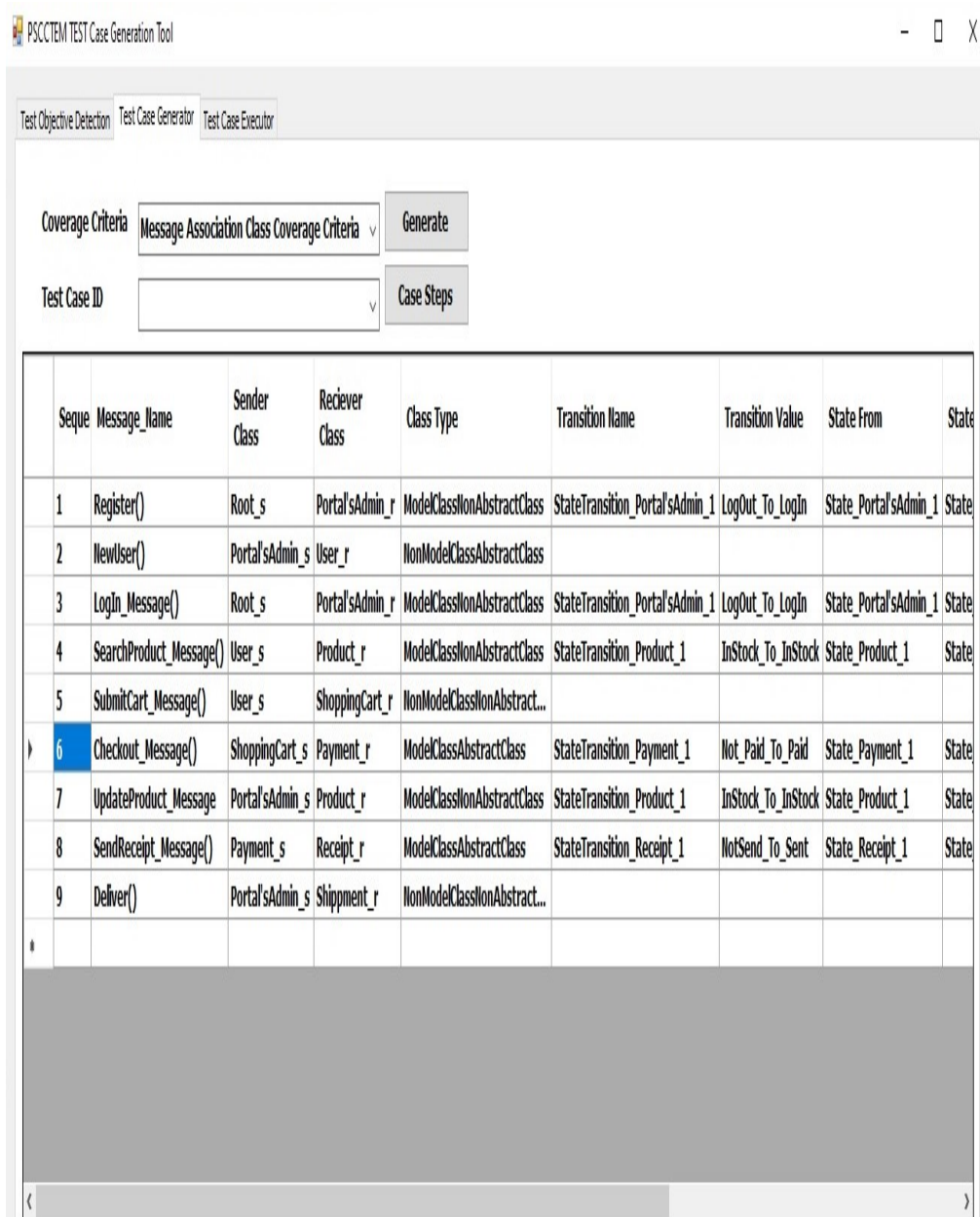


Figure 10: GUI for Generating Test path of Message Association Coverage.

### 5.3.1.2 State transitions class coverage

This coverage will generate 8 paths according to instances number of state transition classes.

GUI for the generated test cases illustrated in figure 11.

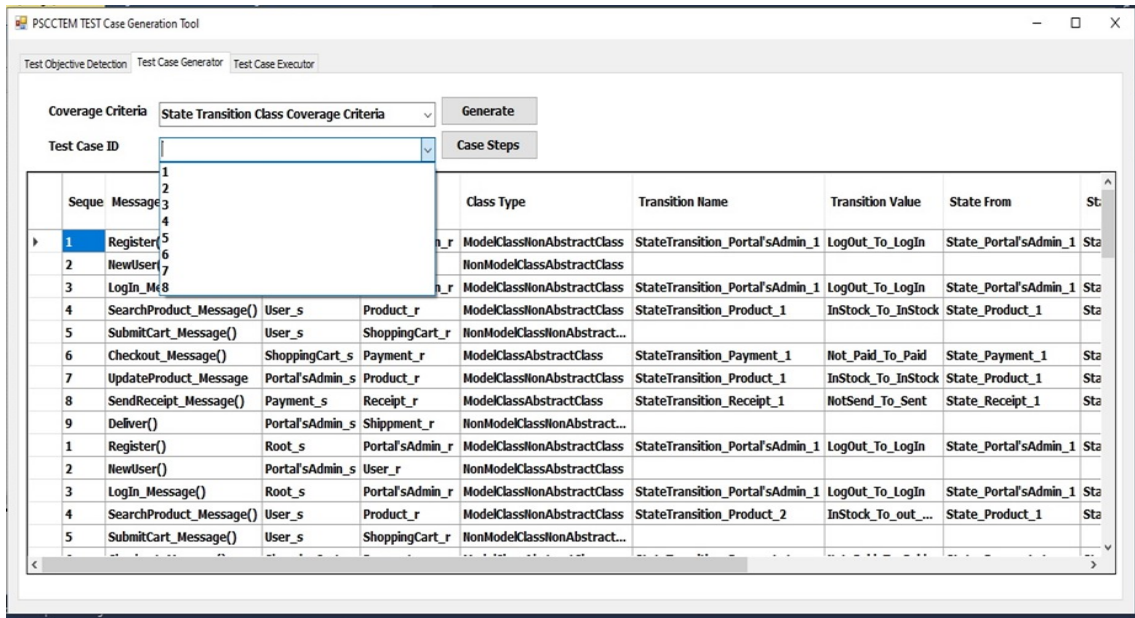


Figure 11: GUI for Generating Test cases of State transitions class Coverage.

### 5.1.3.3 All transitions and generalization coverage

This coverage will generate 16 paths according to the maximum number of child classes' for inheritance relation product by the maximum number of state transition instances of state transition classes.

GUI for the generated test cases illustrated in figure 12.

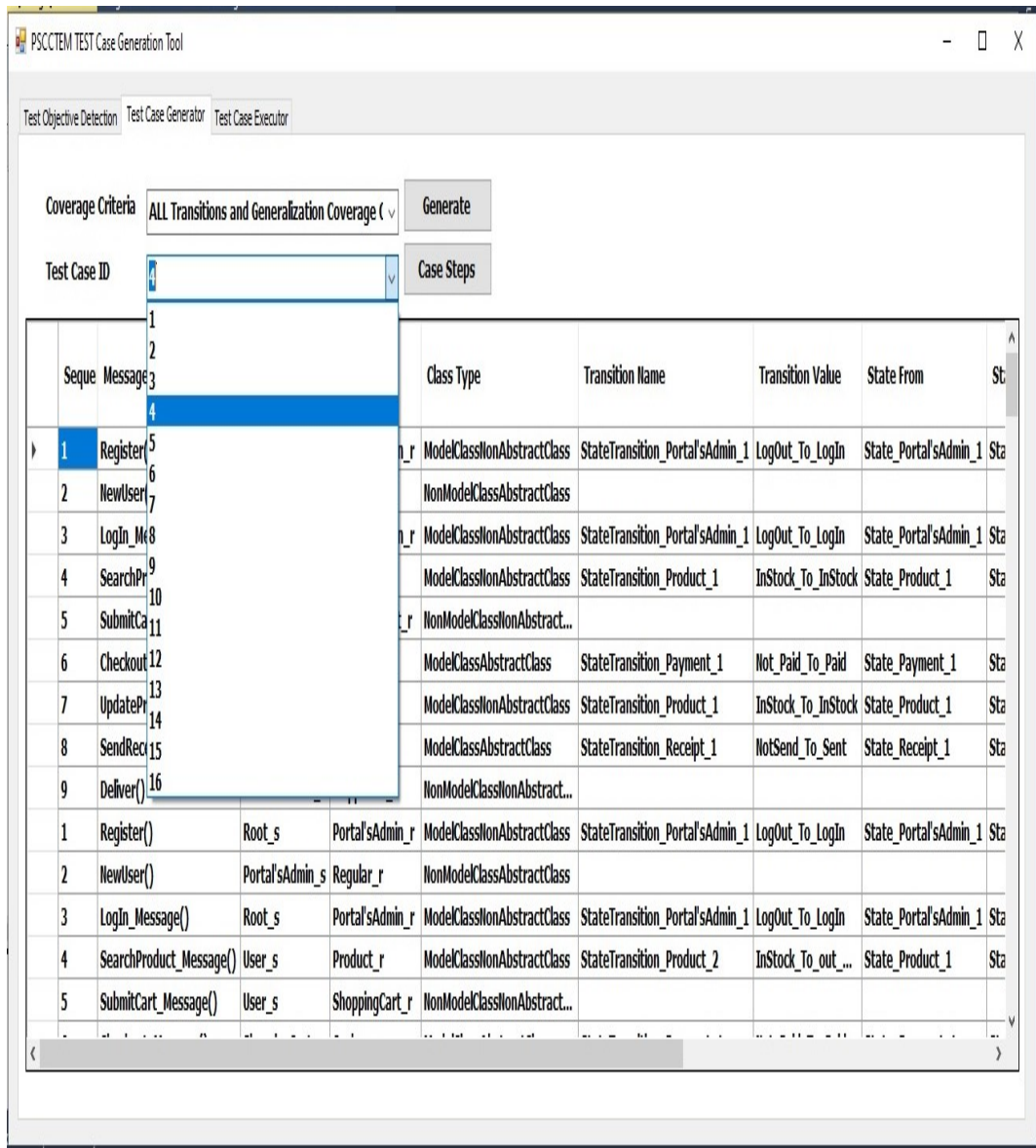


Figure 12: GUI for Generating Test cases of State transitions and Inheritance Coverage.

## 5.2 Experimental Setup

Mutation testing is executed by planting faults using mutation operators. This technique is employed for evaluation testing methods and has been asserted that, its yield useful results [44]. For implementing mutation testing, eight different types of mutation operators were used to plant faults [45]. The criteria for mutation operators selection is the ability to detect the interfacing faults, which are uncovered by interactions between classes. 40 instances of the program (mutant programs) were produced,

with each instance consisting of only one planted fault. Note that 8 random test suites produced for the selected coverage criteria to capture the combinations between the instances of all state transition classes in PSCCOTM ontology. The number of mutants killed for the selected 8 test suites is represented in Table 3 accompanied by the number of paths tested.

Table 3: Number Of Mutants Killed In Each Test Suite

Proposed Test Objective	Mutant Scores for each of the 8 randomly chosen test suites	Number of test paths
Message Association Coverage	27, 28, 30, 29, 27, 28, 29, 27	1
All-State Transition Instances Coverage	31, 35, 35, 35, 35, 37, 37, 37	8
All inheritance and All-State Transition	39, 40, 35, 35, 40, 40, 38, 38	16

In online shopping portal case study, for the selected three test objectives, table 4 summarizes the results by providing the minimum, average, and maximum number of mutants killed by each test objective within the 8 randomly generated test suites. The minimum is computed by getting the lowest value of mutant score of each coverage criterion divided by 40 (the total number of planted mutants). For example, the lowest value of mutant score for message association coverage is 27, which is divided by 40 to get the minimum 67.5% of mutant detection. For the average percentage of mutant detection, the average number is calculated first divided by the total number of seeded faults. The maximum percentage is calculated by taking the largest number of mutant score divided by the total number of seeded faults.

Table 4: Mutation Score Against Test Objective

Evaluation Criteria	Test Objective		
	Message Association Coverage	All-State Transition Instances Coverage	All Inheritance and All-State Transition
Minimum	67.5 %	77.5%	87.5%
Average	64.06 %	77.18%	95.3%
Maximum	75%	92.5%	100%

### 5.2.1 Test Results and Discussion

- Message Association coverage:** The message association coverage criterion is able to detect from 27 to 30 faults out of a total of 40 faults. Message association coverage is able to detect all types of faults, but it cannot completely detect all faults that are seeded in the inheritance classes. The selected path may not cover all child classes which contain the inserted faults.

- All-State Transition Instances Coverage:** This criterion produced a better result that showed a 13.12% increase in fault detection on average than message association Coverage. All-state transition instances coverage detected all types of faults as well, except child classes which contain the inserted faults. Depending on the generated test paths and randomly selected test suites, the total number of detected faults ranged from 31 to 37 faults.

- All inheritance Coverage and All-State Transition Instances:** For this criterion, the total number of detected faults ranged from 35 to 40. This criterion is supposed to be an acceptable compromise for those numerous occasions when all-path Coverage is cost- or time- prohibitive. The test result reveals that Coverage produced a better result than previous coverage criterion by an average of 18%.

The results of validation of the proposed approach show that, all types of inserted faults can be detected by the generated test paths using the PSCCOTM tool. Such test paths consisted of all instances of message association class in the PSCCOTM ontology that are not mutually exclusive. By considering the combinations between state transitions of objects, such state faults could be completely detected depending on the proposed coverage criteria.

All test suites generated by message association coverage detected 64.06%% of faults on average; All-State Transition Instances Coverage detected 77.18% of faults on average; All inheritance and All-State Transition coverage criterion detected 95.3% on average. Figure 13 depicts a bar chart representation of the minimum, average and maximum mutation scores for each test objective.

- PSCCOTM approach is an ontology based approach to store test model elements. PSCCOTM approach is implemented through ease of use tool that manipulates ontology file of type RDF/XML to generate abstract test suite.
- The PSCCOTM tool was implemented through a user-friendly interface that supports defined coverage criteria and ease of use.
- Regarding using Unified Modeling Language (UML), PSCCOTM approach uses UML for building test model.

- Regarding automation, PSCCOTM tool depends on an internal Generator and Executor modules to automatically generate and execute test cases from RDF/XML file.
- Although the [13] can detect from 85% of mutants on maximum, the All generalization and All-State Transition coverage criterion in PSCCOTM approach is able to detect 100% of mutants.

Table 4 : Comparison Of Psccotm And Automated Test Approach

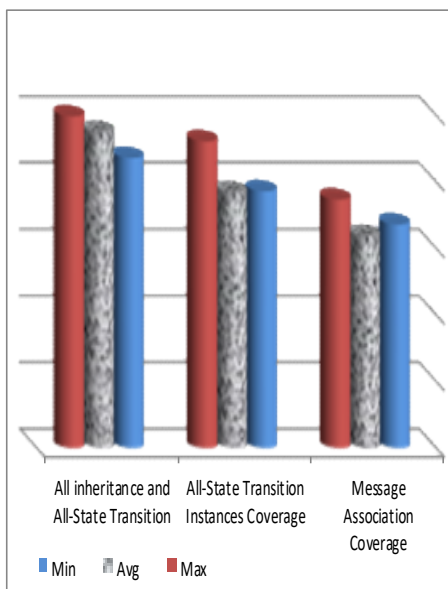


Figure 13: Mutation Score against Test Objective.

### 5.2.2 PSCCOTM And ATCG

To validate the PSCCOTM approach, the approach is compared with test approach in [13] to evaluate the fault detection ability of the PSCCOTM approach. The comparison between them is based on standard evaluation criteria formulated by Havva Gulay Gurbuz1&Bedir Tekinerdogan (2017) [46] as shown in Table 4, the comparison results can be illustrated as follows:

Since the PSCCOTM focuses not only on integration testing as in [13], but also, it focuses on faults that are caused by generalization, polymorphism and the objects in the correct states. The comparison is showing that the proposed approach has powerful capability in mutant detection for object oriented characteristics (abstraction, inheritance, and

Criteria	PSCCOTM Approach	[13] automated based Approach
<b>Model specification</b>	Generation of PSCCTEM test model based on UML to capture the functionality of the system	UsingUMLcollaborati on diagram that doesn't reflect all system picture
<b>Abstract testcase generation</b>	Manipulations of RDF/XML ontology file to generate the test paths.	apply this step by traversing the graph using algorithm
<b>Type of generated test elements</b>	Test suite (test sequences, and test oracles are generated beside of the test cases.)	Test case only
<b>Approach to generate test elements</b>	Proposed a ontology - based testing tool called PSCCOTM	Algorithm only
<b>Test selection criteria</b>	Define our own criteria.	Not specified
<b>Test case specification</b>	define test cases formally	define test cases formally
<b>Test execution</b>	done automatically	done manually
<b>mutants killed</b>	100% when complying with All-State Transition and All generalization coverage criterion	Asserted using C1 metric for testing coverage ,assuming 85% of all faults are revealed

polymorphic methods). In the experiment, the selected test paths of third coverage criterion are covered all possible combinations of state transitions in selected child classes. This is obvious by reading the percentage mutant detection on maximum, the percentage of mutant detection in third coverage criteria on maximum is 100% for selected test paths. The evaluation of the proposed approach indicates that, the pesticide paradox at great extent can be eliminated using it. Test cases can be easily updated according to system modification with effective fault detection.

### 5.3 PSCCOTM And Related Work



According to the results of evaluation of the PSCCOTM approach, the commonalities and differences between existing automation testing techniques and the proposed approach can be highlighted. With regards to this evaluation, none of the existing automation testing techniques have the ability to capture the characteristics of object oriented software. In contrary, PSCCOTM approach is building test model that captures the characteristics of object oriented software by integrating number of UML diagrams in test model. The existence techniques did not have intermediate layer for modeling test model elements into OWL ontology to eliminate the needs for refactoring testing tool. In PSCCOTM, transformation rules are proposed based on ontology definition metamodel for building this layer. In addition to that, PSCCOTM manipulates RDF/XML file of the ontology to generate test cases automatically. The PSCCOTM approach defined their own coverage criteria that have the ability to detect all errors.

In most of recent approaches test cases are executed manually, but in PSCCOTM, test cases are executed automatically. This is done by proposed new class in the PSCCOTM ontology that contains the expected results of execution. A limitation of PSCCOTM approach, it captures generalization relations only between classes. PSCCOTM approach can be extended to capture multiple inheritances between classes and other additional relations between classes.

## 6. CONCLUSION AND FUTURE WORK

This research presented Polymorphism State Collaboration Class Ontology Test Model approach (PSCCOTM) that helps the user to update the test paths easily according to system modifications for overcoming pesticide paradox in inter-class testing of object oriented applications. The PSCCOTM approach developed new testing phase that retains information and knowledge in ontologies for enhancing the limitations of automation testing reducing the need for tuning testing tools on continual basis.

Domain ontology was constructed, which described the vocabularies related to a software engineering domain. The ontology retained the structure of a Polymorphism State Collaboration Class Test Model (PSCCTM) by defining the test model's structural elements and the relationships between them. In this research, the rules for transforming UML PSCCTM test model into

behavioral model ontology, proposed based on the Ontology Definition Metamodel (ODM). Using defined coverage criteria in the previous phase, PSCCOTM approach provides flexibility to define test objectives, which are high-level descriptions of test cases, according to system modifications omitting using the same test cases to overcome pesticide paradox. New tool is introduced to automate the generation and execution of test cases. The PSCCTM tool has three major functions: (1) test objective detection, (2) the test case generation, and (3) test suite Execution. PSCCTM tool supported a graphical user interface for each function. To detect test objectives, the PSCCOTM tool depends on the load method in the XML library for loading RDF/XML of the PSCCOTM ontology to extract the instances of message association class. Test paths can be generated from the test objectives using the Test Case Generator tab of the PSCCOTM tool as demonstrated in this work. The Test Executor executes concrete test cases by filling the test data in the functions calls of test paths. Each test case is then executed on the implementation and the execution results are logged in the PSCCOTM tool for comparing the results of a test run with the expected results. Redundancy elimination operation in PSCCOTM tool is used to check whether a test case already exists in the test-suite to avoid duplications of test cases. The main contribution for the PSCCTEM tool is the selection subset of all test paths that uncovers all defects. This is clear by reviewing the percentage of mutant detection in third coverage criterion on maximum, which is 100% for selected test paths. The experimental results of PSCCOTM execution indicates that powerful ability in generation and updating of test paths based on defined coverage criteria. For PSCCOTM ability to detect faults, the execution results show high percentage of faults detection.

Regarding the future work, new cases studies need to be implemented to confirm the attained results. Also, PSCCOTM implementation can be extend to support other types or levels of testing including system testing, as this work concentrated on inter-class testing only. Execution of test cases needs to be enhanced by prioritizing generated test cases using machine learning techniques for efficient testing process.

## REFERENCES:

- [1] "Software Engineering: Principles and Practices". 2nd Edition, page number 257.
- [2] V. Basili, B. Perricone, "Software errors and complexity, an empirical investigation. In Software engineering metrics". McGraw-Hill, Inc., New York, NY, USA, pp.168-183, 1993.
- [3] K. Naik, P. Tripathy, "Software Testing and Quality Assurance: Theory and Practice", 2008.
- [4] The Software Experts (2008). "Software Process Models". [http://www.the-software-experts.de/e\\_dta-sw-process.htm](http://www.the-software-experts.de/e_dta-sw-process.htm).
- [5] Beizer, B., (1990). "Software Testing Techniques (2nd Ed.)". Van Nostrand Reinhold Co., New York, NY, USA.
- [6] S. Singh, A. Kaur, K. Sharma and S. Srivastava, "Software testing strategies and current issues in embedded software systems". International Journal of Scientific & Engineering Research, 3(4), 1342-1357, 2013.
- [7] D.S. Chaudhary, "Defect clustering and pesticide paradox". PIT Solutions Private Limited, 2015.
- [8] P.P. Mahadik, M.D. Thakore, "Survey on automatic test data generation tools and techniques for object oriented code". International journal of innovative research in computer and communication engineering, 2016, 4(1), pp. 357-364.
- [9] S. Rajvir, S. Anita and B. Rajesh, "Test Case Generation Tools – A Review". International Journal of Electronics Engineering (ISSN: 0973-7383) Volume 10. Issue 2 pp. 586-596, 2018.
- [10] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M.J. Harrold and P.M. Minn, "An orchestrated survey of methodologies for automated software test case generation". Journal of Systems and Software, 86(8), 2013.
- [11] Q. Zhipeng, W. Lisong, K. Jiexiang, G. Zhongjie, H. Wang, W. Yin and S. Xiangyu, "A Method of Test Case Generation Based on VRM Model", 2021 IEEE 6th International Conference on Computer and Communication Systems (ICCCS), pp. 1099-1107, 2021.
- [12] S. Rajvir, B. Rajesh and S. Anita, "Demand Based Test Case Generation for Object Oriented System", IET Software. 13. 10.1049/iet-sen.2018.5043, 2019.
- [13] R. Anbunathan and A. Basu, "Combining genetic algorithm and pairwise testing for optimized test generation from UML ADs", The Institution of Engineering and Technology (IET Software). Volume 13, Issue 5, pp. 423-433. DOI:10.1049/iet-sen.2018.5207, Print ISSN 1751-8806, Online ISSN 1751-8814, October 2019.
- [14] S.K. Barisal, S.S. Behera and S. Godbole, "Validating object-oriented software at design phase by achieving DC". International Journal of System Assurance Engineering and Management, Volume 10, Issue 4, pp 811-823: 811. <https://doi.org/10.1007/s13198-019-00815-8>, August 2019.
- [15] A. Kaur and V. Vig, "Automatic test case generation through collaboration diagram: a case study". International Journal of System Assurance Engineering & Management, 9: 1-15. published at Springer, 2018.
- [16] A. Shah, A. Bukhari, M. Humayun, N. Jhanjhi and F. Abbas, "Test Case Generation using Unified Modeling Language", In: Proc. of International Conference on Computer and Information Sciences (ICCIS)-Ieee, Sakaka, Saudi Arabia, pp. 1-6., 2019.
- [17] Yi. Sun, Y. Xiaohua, L. Jie, Y. Tonglan, X. Zhuoran, W. Zhiqiang and CH. Zhi, "Automatic integration testing through collaboration diagram and logic contracts". Journal of Physics: Conference Series. 1187. 042043. 10.1088/1742-6596/1187/4/042043, 2019.
- [18] M. Vivanti, "Dynamic Data Flow Testing". Doctoral Dissertation, Faculty of Informatics of the Università della Svizzera, Italiana, 2016.
- [19] J. Chen, F. Kuo, Y.T. Chen, D. Towey, C. Su and R. Huang, "A Similarity Metric for the Transactions on Reliability. vol. 66, no. 2, pp. 373-402, June 2017.
- [20] A. Arcuri, L. Briand, "Adaptive random testing: An illusion of effectiveness", In Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '11, pages 265-275. ACM, 2011.
- [21] M. Dimašević, F. Howar, K. Luckow, Z. Rakamarić, "Study of Integrating Random and Symbolic Testing for Object-Oriented Software", In: Furia C., Winter K. (eds) Integrated Formal Methods. Lecture Notes in Computer Science, vol 11023. Springer, Cham, 2018.

- [22] H. Pei, K. ai, B. Yin, P. A. Mathur and M. Xie, "Dynamic Random Testing: Technique and Experimental Evaluation", in IEEE Transactions on Reliability, vol. 68, no. 3, pp. 872-892, Sept. 2019.
- [23] G. Denaro, A. Margara, M. Pezzè, M. Vivant, "Dynamic Data Flow Testing of Object Oriented Systems". DOI:10.1109/ICSE.2015.104, Electronic ISBN: 978-1-4799-1934-5, 2015.
- [24] C. J. King, "Symbolic execution and program testing", Communications of the ACM, 19(7):385-394, 1976.
- [25] B. Zhang, C. Feng, A. Herrera, V. Chipounov, G. Candea and C. Tang, "Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing". IET Software, vol. 12, no. 6, pp. 507-519, 2018.
- [26] T. Chen, X. Zhang, Sh. Guo, H. Li, Y. Wu, "Dynamic symbolic execution for automated test generation". Future Generation Computer Systems, Volume 29, Issue 7, Pages 1758-1773, September 2013.
- [27] [https://doi.org/10.1007/978-3-540-79124-9\\_10](https://doi.org/10.1007/978-3-540-79124-9_10).
- [28] N. Tillmann, J. de Halleux, "Pex-White Box Test Generation for .NET", Lecture Notes in Computer Science, vol 4966. Springer, Berlin, Heidelberg, 2008.
- [29] C. Yeh, H. Chung and S. Huang, "Target-Aware Symbolic Fuzz Testing", In. IEEE 39th Annual Computer Software and Applications Conference, Taichung, pp. 460-471, 2015.
- [30] B. Zhang, C. Feng, A. Herrera, V. Chipounov, "Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing", In IET Software, vol. 12, no. 6, pp. 507-519, 2018.
- [31] S. Cha, S. Hong, J. Kim, J. Lee and H. Oh, "Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics". The Department of Computer Science and Engineering, Korea University, Seoul, Korea, 2019.
- [32] S. Cha, S. Hong, J. Bak, J. Kim, J. Lee and H. Oh, "Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics", in IEEE Transactions on Software Engineering, 2021.
- [33] P. McMinn, "Search-Based Software Testing: Past, Present and Future", in IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, pp. 153-163, 2011.
- [34] Sina. Shamshiri, M. J. Rojas, L. Gazzola, G. Fraser, Ph. McMinn, L. Mariani and A. Arcuri, "Random or Evolutionary Search for Object-Oriented Test Suite Generation". Published online in Wiley InterScience ([www.interscience.wiley.com](http://www.interscience.wiley.com)), 2017.
- [35] S. Sheoran, N. Mittal and A. Gelbukh, "Artificial bee colony algorithm in data flow testing for optimal test suite generation", Int J Syst Assur Eng Manag, 2019.
- [36] M. Panda, S. Dash, A. Nayyar, M. Bilal and R. M. Mehmood, "Test Suit Generation for Object Oriented Programs: A Hybrid Firefly and Differential Evolution Approach, " in IEEE Access, vol. 8, pp. 179167-179188, 2020, doi: 10.1109/ACCESS.2020.3026911.
- [37] J. E. Rapos, J. Dingel, "Incremental Test Case Generation for UML-RT Models Using Symbolic Execution", in IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 962-963, 2012.
- [38] J. Bozic, Y. Li, and F. Wotawa, (2020). "Ontology-driven Security Testing of Web Applications". IEEE International Conference on Artificial Intelligence Testing (AITest), pp. 115-122, doi: 10.1109/AITEST49225.2020.00024.11.
- [39] F. Wotawa, J. Bozic, and Y. Li, (2020). "Ontology-based Testing: An Emerging Paradigm for Modeling and Testing Systems and Software". IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 14-17, doi: 10.1109/ICSTW50294.2020.00020.
- [40] Verma, Kunal | Kass, Alex, "Model-Assisted Software Development: Using a 'semantic bus' to automate steps in the software development process". Semantic Web, vol. 1, no. 1-2, pp. 17-24, 2010.
- [41] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, Z. Ziqbala, A. Nadeem. Muhammad, "A State-based Approach to Integration Testing based on UML Models". Information and Software Technology, vol. 49, pp. 1087-1106, 2007.
- [42] C. Calero, F. Ruiz, M. Piattini, "Ontologies in Software Engineering and Software Technology". published by springer, 2015.
- [43] "Ontology Definition Metamodel", Technical report, Object Management Group. OMG Document Number: formal/2014-09-02 Standard Document URL : [http:// www. omg. org/spec/ODM/1.1/](http://www.omg.org/spec/ODM/1.1/). September 2014.

- [44]H.J. Andrews,C.L. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments“,in proc of the IEEE 27th International Conference on Software Engineering (ICSE). St. Louis, Missouri, USA, pp. 15-21, 2005.
- [45]G. Fraser and A. Zeller, “Mutation-Driven Generation of Unit Tests and Oracles,” IEEE Transactions on Software Engineering, vol. 38, no. 2, pp. 278-292, March-April 2012.
- [46]H.G.,Gurbuz,B.,Tekinerdogan,“Model-based testing for software safety: a systematic mapping study“. Software Qual J 26, 1327–1372, 2018. <https://doi.org/10.1007/s11219-017-9386-2>.