Future Generation of Communication and Internet of Things (FGCIOT)

Journal homepage: http://fgciot.semnaniau.ac.ir/

Research paper

A Classification Framework of Test Models in Model-based Testing

Arash Sabbaghi 1

1 Department of Computer Engineering,Semnan Branch, Islamic Azad University, Semnan, Iran

a.sabbaghi@semnaniau.ac.ir

Article Info

Article History:
Received
Revised
Accepted

Keywords:
Software testing, Model-based testing, Automatic test case generation, Test models.

*Corresponding Author's Email Address:
a.sabbaghi@semnaniau.ac.ir

Extended Abstract

In model-based testing (MBT), the quality of input models and their relevance with the testing target has a direct impact on the quality of the test suite and the effectiveness of the whole testing process. Choosing inappropriate models may increase the number of MBT steps and may not fulfill the testers' expectations. In this paper, we focus on different input models of MBT and represent a classification framework for them. The classification is performed by considering their nature and testing abilities. We discuss the strengths and weaknesses of test models regarding their potential for generating test cases, and summarize the existing works in the literature based on the proposed classification framework. The aim of this paper is to improve the understanding of model-based test case generation approaches and help the testers to choose appropriate models to exploit test cases with regard to their testing goals and purposes.

Introduction

Due to the increasing complexity of today's software systems, automation of the testing process has become a must [1-4]. In this regard, model-based techniques have received much attention and showed promising results. In MBT, different models can be used as input for test case generation, and each one is suitable for specific testing needs and has its own advantages and drawbacks. In fact, with regard to the testing target, it is very important to choose appropriate models [5]. Testing target may include different levels of testing, different types of system under test (SUT), or different parts of SUT. Not every model can be used for every testing target. For example, for testing real-time systems, input models should incorporate timing constraints [6], or for integration testing, input models must precisely indicate communications between different parts of SUT [7].

Therefore, choosing inappropriate input models may increase the number of MBT steps and may not fulfill the testers' expectations. To the best of our knowledge, there is no research in which draws a complete classification and comparison of input models to show their abilities and potentials in MBT with regard to the testing target.

In this paper, we focus on test models, represent our classification framework, and show their application for MBT. The classification is performed based on the information provided by test models to generate test cases, which categorize test models into five groups: state-based models, interaction-based models, structure-based models, operation-based models, and hybrid models. The models in each category can be used to generate test cases to test

Doi:

SUT from different points of view. In our previous work [8], we described state-based models in detail. In the following, we describe the other categories and discuss their abilities and shortcomings to generate test cases. The aim of this survey is to improve the understanding of model-based test case generation approaches and helps the testers to choose appropriate models to exploit test cases with regard to their testing goals and purposes.

Interaction-based models [9, 10] describe the dynamics of the system behavior and focus on representing interactions between different parts of the system. Structure-based models [11] describe the static aspects of the system. Operation-based models [5, 12] focus on **the actions** within the behavior and depict the operational workflows. Studies in the last category (e.g. [13]) use the combination of the information provided by different models to generate test cases. Each model has its own strengths and weaknesses, and using the information provided by one model to compensate for the lack of such information in another, would make test case generation more efficient.

The rest of the paper is organized as follows: in section 2, we present the interaction-based models. Section 3 introduces structure-based models, and operation- and hybrid models are described in Section 4 and 5, respectively. Finally, section 6 is dedicated to the conclusion.

I. Interaction-based Models

Interaction-based models describe the dynamics of the system behavior and focus on representing interactions between different parts of the system. The most widely used interaction-based models in MBT, include UML sequence diagram [9, 14-17], UML communication diagram [18-20] and UML use case diagram [10, 13, 21-24].

UML Sequence diagram is a graphical scenario language that consists of objects and messages that exchange among the objects in the order they occur in the system over time. A sequence diagram represents various interactions among different objects through the invocation of methods and describes how a set of objects interact to achieve a behavioral goal.

The sequence diagram has two dimensions: the vertical dimension that represents time, and the horizontal dimension that represents object instances. The communication between object instances is denoted by arrows between lifelines. The lifelines are the vertical dashed lines that represent the existence of corresponding object instances at a particular time. Since UML 2.0, a set of interactions can be framed together to form interaction fragments, and multiple interaction fragments can be combined using combined fragments. A combined fragment consists of interaction operands whose type is determined by the interaction operator. An interaction operand is a group of message sequences that run if the guard condition is met. An interaction operand without guard condition always runs. There are different interaction operators such as loop for repetition, alt/opt/break for selection, par for concurrencies, seq for weak sequencing, etc. Weak sequencing allows partial parallel execution between lifelines and means that events on different lifelines from different operands may occur in any order. A combined fragment can also contain nested combined fragments.

Based on [25], A sequence diagram can be defined formally as a 9-tuple $D = (d, I, E, <, \Sigma_{msg}, M, F, X, Exp)$, where:

$d \in \Sigma_{name}$ is the name of the diagram and Σ_{name} the set of all diagram names;

I is a finite set of object instances (lifelines);

$E = \bigcup_{i \in I} E_i$ is a set of events for lifeline i, s.t. $\forall i, j \in I : E_i \cap E_j = \emptyset$;

$<$ is a set of partial orders which defines for instance line $i \in I$ a set: $<_i \subseteq E_i \times E_i$;

Σ_{msg} is a finite set of message labels l;

M is a set of messages $M \subseteq E \times \Sigma_{msg} \times E$, s.t. for every $\mathrm{m}_1, \mathrm{m}_2 \in M$ with $m_1 = (e_{11}, l_1, e_{12})$ and $m_2 = (e_{21}, l_2, e_{22})$: $m_1 \neq m_2 \Longrightarrow e_{11} \neq e_{12} \neq e_{21} \neq e_{22}$;

F is a set of interaction fragments for which the functions op, ev, sub are defined as:

$op: F \to \Omega \times \mathbb{N}$ associates an operator $\Omega \in \{strict, par, opt, alt, loop, break, seq, critical, ignore, consider, assert, neg\}$ and the number of operands to a fragment;

$ev: F \times \mathbb{N} \to 2^E$ associates a set of events to a pair (id, n) of a fragment $id \in F$ and an operand index number n;

$sub: F \times \mathbb{N} \to 2^F$ associates a set of nested fragments to a parent fragment and an operand index number;

$X = \{X_i\}_{i \in I}$ a set of local variables indexed by object instances $i \in I$.

Exp is a set of expressions, where each expression is associated as a guard to a message or a fragment using the function guard: $M \cup F \to Exp$

Since sequence diagrams describe the interactions among software components, they are a good source for integration testing and detecting interaction faults. Also, sequence diagrams are suitable for the realization of use case specification, thus they are useful for functional system testing too.

The par and seq interaction operators allow sequence diagrams to specify concurrent systems, and because in such systems complexity arises when objects interact with each other [14, 17], they are a good choice for testing concurrent systems and concurrency. Like all models which allow specifying concurrency, the test explosion is a matter of concern when using sequence diagrams to generate test cases. Khandai et al. [17] considered par fragments and proposed to generate test cases for concurrent systems by converting sequence diagram to a Concurrent Composite Graph (CCG). Each node in the CCG represents a sequence of messages within one operation fragment. To avoid the issues like communication deadlock and synchronization, they proposed to use breadth-first traversal on CCG when encountering fork nodes (for concurrent activities) and to use DFS for the rest of the graph (for sequential activities).

The main challenge for test case generation from sequence diagrams with regards to their complex and non-hierarchical structure is extracting the flow of control among the fragments and their nested occurrences. Sequence diagrams do not have a convenient structure for repetition, recursions, and conditions. Combined fragments increase the ability of sequence diagrams for behavior modeling but make scenario representation and their flow analysis a challenging task. The studies in this area generally generate an intermediate form for formalizing and structuring the sequence diagram [20, 26, 27]. It should be noted that the formalization should be carried out in a way that be comprehensive in terms of covering fragments and also retain the default behavior and semantic of the sequence diagram. For example, some formalizations ignore the standard interpretation of weak sequencing and force the synchronization of lifelines on entering and exiting fragments [28]. Cartaxo et al. [26] proposed an approach for feature testing of mobile applications by converting sequence diagrams into a labeled transition system (LTS). They just considered repetitive and conditional sequences in their model and eliminate the message exchange between internal objects since they tend to perform functional testing. The test sequences are generated by the depth-first traversal of LTS. Nayak et al. [20] proposed to convert sequence diagram into a directed graph named structural composite graph (SCG) in order to systematically investigate the comprehensive flow of control by considering loop, alt, opt, break, and par combined fragments. In this transformation, they show all the interaction fragments and flow of control of these operands unambiguously and in a structured way. They mapped the messages within a fragment into a block node, the entrance to a par fragment into a fork node, the exit from a par fragment into a join node, the conditional expression among operands of a fragment into a decision node, and the exit form selection fragments into a merge node. Then test scenarios are generated by the depth-first traversal of SCG. Authors in [27] proposed a toolset for conformance testing using sequence diagram, which supports all interaction operators including weak sequencing by retaining their default semantics. They translated sequence diagram into an extended Petri net that

combine the characteristics of colored and event-driven Petri nets.

Communication diagram [18-20, 29], formerly called collaboration diagram, like sequence diagram, represent the inter-object communications and capture the exchange of messages between objects. Communication and sequence diagrams can represent the same object interactions. The sequence diagram emphasizes on time ordering of messages, while the communication diagram focuses on the structural organization of objects and represents a clear visualization of how objects communicate to perform a behavioral goal. In a communication diagram, the objects are connected by links that represent messages. The links are labeled with unique sequence numbers, which determine the ordering of messages.
Sequence diagrams are more commonly used in practice than communication diagrams [30], and naturally, in the literature, there are much more approaches for generating test cases from sequence diagrams than communication diagrams. The ability of communication diagrams in representing the structure of objects in communications and also their ability to depict the overall design of the system have been considered in the literature for generating test cases. The communication diagram is suitable for integration and cluster-level testing. Authors in [19, 31] developed some coverage criteria for collaboration diagrams and used them to generate test cases for testing implementation and design, respectively.

Use case model [10, 23, 24] defines the frontier of the SUT, its development begins early and shows the main functionality of the system at a high level of abstraction. A use case represents different possible sequences of interactions between the external user and the SUT, and comprises a diagram part and a textual description known as use case scenario. The diagram part visualizes the interactions among use cases and actors. The use case scenario informally describes one of the system or subsystem functionalities. Each functionality can be realized inside a software component, like a module, or can be obtained from interactions of several components.
Since use cases show the main functionalities of the SUT, they are a good starting point for MBT, tell the tester what to test, and are good sources for integration, system, and acceptance testing [22]. The pre- and post-conditions of use cases are good sources for generating the initial state and the oracle for test cases, respectively. Also, since the scenarios are modeling the system from the user's perspective in a black-box manner, they may not be well-suited for unit testing.
The existence of a large gap between high-level use cases and concrete test cases makes the full automation of the test case generation process difficult. There are two main challenges for generating test cases from use case diagrams: first, determining sequential constraints and dependencies among use cases, and second, dealing with the informal nature of use case scenarios.
Sequential constraints between use cases can be determined by checking if the post-condition of one meets the pre-condition of the other. Nebut et al. [21] extend use cases with the contracts in the form of OCL. Contracts are specified using pre- and post-conditions. By using the contracts for each use case, they built a Use Case Transition System (UCTS) from which all valid sequences of use cases are extracted. Authors in [32] represent the sequential dependencies between use cases for each actor by an activity diagram in a way that the vertices are use cases and the edges are sequential dependencies between them. Independent use cases modeled in the fork-join constructs. Swain et al. [13] first construct an activity diagram for use case diagram and then convert it to Use Case Dependency Graph (UDG). Use case dependency sequences are generated using UDG.
After deriving use case sequences, it is turn to derive test sequences from use case scenarios. Test scenarios can be generated directly from natural language requirements using Natural Language Processing (NLP) [33], or can be generated indirectly by making use case scenarios more formal in

different ways such as transforming them into state chart diagram [34], sequence diagram [13, 21, 32], activity diagram [32, 35], collaboration diagram [31], or Petri nets [36, 37]. It is desirable that this transformation carry out automatically as the proposed approaches in [22, 33, 35-37]. To this end, some studies proposed to write requirement specifications in strict forms such as Requirements Specification Language (RSL) [22] or Restricted-form of Natural Language (RNL) [36]. Finally, by replacing test sequences generated from use case scenarios in the use case sequences, test scenarios can be generated.

II. Structure-based Models

Structure-based models describe the static aspects of the system. The most widely used structure-based models in MBT include the UML class diagram [11] and UML object diagram [38].

Class diagram captures the static structure of the SUT classes and provides information about class names, class attributes, type of attributes, class cardinality, method signatures, class relationships, multiplicities, inheritance, etc. A class diagram can be defined formally as a 2-tuple $CD = (CN, AN)$ where:

CN is a finite set of classes. Each $C_i \in CN$ is a 2-tuple $C_i = (Attr, M)$ where:

$Attr$ is a set of class attributes $\{< attr_i : type_i, c_i >\}$. Each $attr_i$ is the name of the attribute with the type $type_i$ and c_i is the constraint over $attr_i$.

M is a set of method signatures $\{< m_i(p_1 : type_1, \dots, p_n : type_n), Rtype_i >\}$ where m_i is the name of the method, $p_1, \dots, p_n$ are the parameter names, $type_1, \dots, type_n$ are parameter types and $Rtype_i$ is the return type.

$AN = < C_1, Type, C_2 >$ is a set of associations between classes where $C_1, C_2 \in CN$ and $Type$ is the name of association.

By providing valuable information such as method signatures, which include parameter names, parameter types and return type, class attributes that include their names and types, and constraints in the form of OCL on the class attributes (for representing the range of attributes), and on the operations (pre- and post-conditions), class diagrams are a rich source for representing domain model and complementing other test models. For this reason, they are mostly used to supply complementary information for the testing process such as providing the required information for model augmentation [39], identifying the entities in the system [40], or determining a set of object configurations from which the test is started [31].

Different types of faults that are related to the evaluation of inheritance, object states, and associations between objects can be detected by class testing. For this, Andrews et al. [31] proposed some coverage criteria to generate test objectives as follows: Generalization (GN) criterion, class attribute (CA) criterion, and association-end multiplicity (AEM) criterion.

Another approach to generate test cases directly from class diagrams is to generate a sequence of methods with different ordering by considering the relationship between classes [41, 42]. At the end of the test execution, it is checked whether the resulting states of the involved objects are correct or not. Shanti et al. [41] proposed random ordering of methods by applying the genetic algorithm's tree crossover [43] on the tree structures obtained from class diagrams in order to create a new generation of trees. The generated trees are converted into binary trees and then traversed in DFS order to generate test scenarios. Since class diagrams do not provide behavioral view of the system, such blindly generated test scenarios may not be effective in revealing faults.

Class diagrams can be used to determine an order to integrate and test the classes during integration testing. Integration of classes is often incremental and needs to generate stubs in order to simulate the behavior of classes that have not been already tested. The main challenges in this area are minimizing the number of required stubs and breaking dependency cycles in the class diagram. Traon et al. [44] proposed to generate Test

Dependency Graph (TDG) from class diagrams to determine the ordering of classes and methods for integration testing. Vertexes of the TDG represent a class or a specific method of the class, and directed edges represent dependencies. Different dependencies in the class diagram, such as class-to-class and method-to-class are captured to generate TDG. Class-to-class dependencies can be easily identified through class relationships in the class diagram. A method-to-class dependency exists if a method has an object of a class declared in its signature. The presence of dependency cycles in the class diagrams is the main obstacle to the topological ordering of classes. So, they refined TDG in order to apply graph-based algorithms for breaking cycles and determining the ordering of classes. Zhang et al. [45] proposed an approach for determining optimal class integration test order with the minimum number of stubs, considering abstract classes and polymorphism. They mapped class diagram relationships into Object Relation Diagram (ORD), found out the strongly connected components using Tarjan's algorithm, and then used graph-based heuristic algorithm to break cycles. In addition to the graph-based algorithm to break cycles, search-based algorithms can be employed too.

Object diagram shows a snapshot of the detailed state of the system at a certain point in time as a collection of objects, each in a particular state, and a link between objects indicating a possible communication. Each object state may be constrained by an assertion or values of its attributes. Object diagrams are more concrete than class diagrams since unlike class diagrams, their elements are in concrete form to represent real-world objects. Object diagrams are mostly used to provide complementary information for the test case generation process, such as specifying an initial configuration [40].

III. Operation-based Models

Operation-based models focus on the actions within the behavior and depict the operational workflows. The most widely used operation-based model in the literature of MBT is the UML activity diagram [5, 12, 46-52].

UML activity diagram is a powerful tool and one of the most important design artifacts used for behavior modeling, which can represent the business workflow of the SUT in different levels of granularity. Activity diagrams capture the key system behaviors and perfectly describe the realization of the system operations in the design phase.

An activity diagram comprises a set of nodes, edges, and swim lanes. Different types of activity nodes occurring in an activity diagram include action nodes, control nodes, and object nodes. Each action node represents a sequence of statements or an operation of the system. An action begins execution when receiving data from its incoming edge, waits for the completion of its computation, and then the execution is directed to the successor nodes. This is due to the adoption of activity diagrams with the token mechanism of Petri net [53]. Control nodes coordinate the control and data flow between other nodes and include branch and merge nodes, fork and join nodes, and initial and final nodes. Activity diagrams support both conditional and concurrent behaviors. Conditional behaviors are specified by branch and merge nodes, and concurrent behaviors are defined by fork and join nodes. A fork node generates multiple tokens for its descendent nodes, and the join node acts as a synchronization point and passes the token to the subsequent nodes only when all the synchronized threads become ready. Object nodes are used for providing inputs and outputs for an action and can be used as oracle too [54]. Edges are defined as the transitions, which represent control flows and data flows between nodes, and swim lanes represent the supplier of the activities. Notice that any construct can be nested

with any other constructs. An activity diagram can be formally defined as a 5-tuple $D = (A, T, F, a_I, a_F)$ where:

$A = \{a_1, a_2, \dots, a_m\}$ is a finite set of action nodes;

$T = \{\tau_1, \tau_2, \dots, \tau_n\}$ is a finite set of completion transitions;

$F \subseteq \{A \times T\} \cup \{T \times A\}$ is the flow relation between action nodes and transitions;

$a_I \in A$ is the initial activity state, and $a_F \in A$ is the final activity state; there is only one transition $\tau \in T$ such that $(a_I, \tau) \in F$; and $(\tau', a_I) \notin F$ and $(a_F, \tau') \notin F$ for any $\tau' \in T$.

Most of the code-oriented structures are available in the activity diagram, and generally, a path in an activity diagram is a possible run-time execution path of the implemented operation [55]. The behavior modeled by activity diagrams is easy to understand. Also, activity diagrams are flexible in the behavior modeling in such a way that they can be used to give a quick overview of the entire system or can be used to depict the internal logic of a complex operation and details of a procedural implementation [50, 52]. Since the activity diagram expresses how the system functionalities can be exercised and implemented, tells the tester how to test, and is a good basis for functional testing.

UML activity diagrams can be used to model the dynamic concurrent scenarios of a group of objects; thereby they are a good choice for testing concurrency and are widely used in this area [50, 56-58]. Also, their proximity to code has made them, in addition to being widely used in the generation of test data, to be a good source for reducing and optimizing test cases too. For example, Chen et al. [59] proposed an approach to reduce randomly generated test cases using activity diagrams. They interpret each activity state in the activity diagram as the execution of one method in the java program and instrument the java program under test according to its activity diagram specification for gathering the program execution traces. The java program execution traces are a sequence of events corresponding to method completions. By running the instrumented java program with randomly generated test cases, a set of program execution traces is obtained. Then by matching the obtained execution traces with the activity diagram according to a specific adequacy criterion, a reduced test set is generated. They considered three coverage criteria: activity coverage, transition coverage, and simple path coverage. For example, with regard to the activity/transition coverage, a test case is selected if its corresponding execution trace contains some activity/transitions which are not covered previously.

The ability of activity diagrams to precisely represent operations has led them to be used in regression testing for detecting changes in the semantics of operations [60, 61]. Ye et al. [60] proposed an approach to identify the changes between two versions of SUT by comparing the old and new activity diagrams. By the comparison, they identified the affected, unaffected, removed, and new paths. For the new paths, they generate new test cases, and for the affected, unaffected, and removed paths, they classified previously generated test cases into retestable, reusable, and obsolete, respectively. Finally, retestable and newly generated test cases are chosen to test the new version of SUT. Since in this approach, changes in the static structure of the SUT may not be detected, Dahiya et al. [61] proposed to identify the changed operations using activity diagrams and searched them in the class and sequence diagrams of the SUT in order to extract their corresponding retestable test cases.

The challenges of activity diagrams for MBT can be listed as follow: 1- the presence of loops and concurrent activities result in path explosion, which should be managed to generate adequate test scenarios in a reasonable time [62]. 2- because of the non-structural properties such as fork-join constructs and nested combination of control structures that may exist in activity diagrams and may lead to ambiguous interpretation of them [54], it is difficult to identify all possible test scenarios, so flattening the diagram [63] and transforming them into a well-structured form seems necessary [50, 54, 57].

To cope with path explosion caused by loops, researchers mostly employ some coverage criteria

such as basic path, in which loops are executed for a limited number of times (for example, exact once [56], at least once [64], or at most once [55, 65]). Sapna et al. [66] addressed path explosion caused by concurrent activities and proposed to enforce an ordering among them by considering their interleaved execution inside a fork-join construct. Imposing domain dependency between activities helped them to cope with path explosion and generate the optimal number of test scenarios by discarding illegal or irrelevant combinations of activities. To reduce the number of test cases for concurrent systems, Kim et al. [58] proposed to keep only the behaviors related to testing by eliminating the internal processing activities and focusing on the external interaction of the system. They converted the activity diagram into an input-output explicit activity diagram (IOAD), which explicitly shows the input to and the output from the system and omits the non-external inputs and outputs. IOAD is then used to construct a directed graph from which the test scenarios were extracted. Arora et al. [67] proposed to use a bio-inspired approach named Amoeboid Organism Algorithm (AOA) to generate test scenarios for the concurrent section of activity diagrams. AOA draws its inspiration from the internal mechanism of the slime mould Physarum Polycephalum. They showed that their approach outperforms ant colony optimization and genetic algorithm [43] in terms of reducing redundancy and increasing the number of test scenarios, respectively.

In order to cope with complex dependencies that arise within nested structures, and also identifying more test scenarios, Nayak et al. [54] proposed to convert activity diagram into a well-formed structure. They first classified the various control constructs in the activity diagram into loop constructs, selection constructs, and fork constructs. Each control construct is denoted as a minimal region with a distinguished entry node and exit node, which can be analyzed independently of other constructs. Next, they converted the activity diagram into a model called intermediate testable model (ITM) using the classified control constructs. The conversion is done by mapping each minimal region into a composite node in successive steps in order to retain the nesting relation of control constructs. Therefore, the ITM would be a concise representation of the activity diagram in which each of its composite nodes encloses a control construct. In order to generate test scenarios, the base path from the initial node to the final node is extracted from ITM. The base path can be considered as a hyper test scenario. Then in a recursive manner, the composite nodes are expanded by choosing one of their internal paths. The number of internal paths to be replaced is determined by the coverage criteria. The internal paths are generated using depth-first search in the control construct graph. Authors in [50] proposed to convert the activity diagram into a standardized structure, which is a set of extended AND-OR binary trees (EBTs). The transformation is performed based on a set of transformation rules, and its goal is to eliminate fork and join elements and represent branches and concurrent flows as EBTs. The derived EBTs are then traversed to generate test scenarios.

IV. Hybrid Models

Studies in the last category use the combination of the information provided by different models to generate test cases. Each model has its own strengths and weaknesses, and using the information provided by one model to compensate for the lack of such information in another, would make test case generation more efficient.

Based on the way of utilizing the information to generate test cases, the test models used in these studies can be generally divided into two groups: compound models [7, 68] and complementary models [13, 69].

Studies in the first category, integrate the selected models to form a new compound model which can be used to generate better test cases. Sumalatha et al. [68] utilized activity and sequence diagrams, converted them into activity and sequence diagram graphs, and then inserted the activity diagram graph into the sequence diagram graph to form an activity-sequence graph. Test cases are generated by

traversing the activity-sequence graph in breadth first order. The idea is that activity diagrams describe the flow of activities inside the objects, represent the implementation of an operation, and can realize the messages of sequence diagrams (methods). Sarma et al. [70] proposed to integrate use case diagram and sequence diagram by converting them to use case diagram graph (UDG) and sequence diagram graph (SDG). In the integrated diagram named system testing graph (STG), each UDG is connected to its SDG, and test cases are generated by traversing STG. Their approach covers use case initialization faults, use case dependency faults, and also scenario and interaction faults. Their achievements from model integration are twofold: first, the model integration helped them to decide whether a test driver needs to apply a specific test suite or not. Second, the information needed for model augmentation can be mined once and used in different levels such as use case and sequence diagram. Ali et al. [7] suggested combining the UML collaboration and state diagram to perform integration testing of classes. The idea is that the interactions between objects should be exercised for all possible states of the objects involved. Therefore, the generated test cases aim at detecting faults that may arise due to invalid object states during object interactions. They have integrated UML collaboration and state diagram to form a graph called SCOTEM (State Collaboration Test Model). Each vertex in the SCOTEM corresponds to an instance of the class in a distinct abstract state. The object states were extracted from the state diagram. Each edge in SCOTEM can be a message or transition edge. A message edge represents a call action between two objects extracted from the collaboration diagram, and a transition edge represents a state transition of an object which is available in the state diagram. The test paths are generated by traversing the SCOTEM graph. In order to perform efficient integration testing while considering object states, Swain et al. [71] proposed to generate a State-Activity Diagram (SAD) by synthesizing the UML statechart diagrams of different objects involved in a particular use case with an activity diagram. SAD is generated by considering the activities in the activity diagram and the corresponding actions in the UML statechart diagram to form state-activity nodes. They also considered the synchronization of activities over multiple interacting objects by defining AND-OR nodes and waiting activities in SAD. The test scenarios are generating by the depth-first traversal of SAD. Their evaluation using mutation analysis showed the effectiveness of the generated test cases in finding seeded integration faults in the source code. Authors in [72] proposed to generate and prioritize test scenarios by merging activities in the activity diagram with the sequence of messages in the communication diagram. The idea is that communication diagrams represent the high-level design while activity diagram is closely related to the internal structure of the code. In fact, activity diagrams can represent the procedural flow of method execution corresponding to the messages between objects in the communication diagram. By this combination, message and activity paths can be both covered, which can lead to revealing more faults and performing more efficient prioritization. Pilskalns et al. [73] utilized class diagrams, sequence diagrams, and OCL expressions to find inconsistencies among them. The OCL pre- and post-condition constraints and invariants are used as oracle in the oracle procedure. They first construct a Directed Graph (DG) from each sequence diagram by considering opt, alt, and loop fragments. Each vertex of the DG represents either a message or a sub-DG (a combined fragment) and edges represent the ordering between either messages in sequence diagram or sub-DGs. The class diagram and OCL constraints are also used to generate Class and Constraint Tuple (CCT). CCT encapsulates all the information provided by class diagrams such as hierarchy, associations, attributes, etc. The DG and CCT are then combined to form Testable Aggregate Model (TAM). This is done by replacing each class name in each vertex with the corresponding CCT. This combination allows them to apply the test adequacy criteria for both class diagram and sequence diagram.

The physical integration of test models is helpful in covering a broader range of errors and wider regions of SUT. On the other hand, choosing appropriate test models to be integrated as well as the way of integration is a challenging task.

Studies in the second category use the information of different models as a complementary piece of knowledge and generate test data based on the collected information. In other words, these works integrate the information provided by test models instead of physically integrating the models. Kumar et al. [69] extracted the information from class, activity, and state diagrams and store them in separate tables, and finally utilized them to generate test cases. Authors in [40] utilized class, object, and state diagrams to generate test cases in a way that the class diagrams identify the entities in the system and explain the structure of signal events, the state diagrams, one for each class, explain how these entities may evolve and specify test purposes, and the object diagrams specify an initial configuration. Andrews et al. [31] utilized class and communication diagrams to generate test cases. The class diagram is used to define a set of object configurations for starting the test and checking the output, and the communication diagram is used to generate test scenarios. Pilskalns et al. [74] used the information from class and sequence diagrams to detect and correct errors in the design phase. Overall, this approach allows obtaining test information for model augmentation from different sources and also helps to generate test cases for testing different aspects of the system by utilizing different models. The main challenge of this approach is the complexity of creating a valid and effective relationship between information from different test models with different natures.

Conclusion

Model-based testing is an effective approach to automate test case generation process by utilizing different models of the system under test, which can be used to test the final product as well as to validate and improve the models themselves. In the meantime, the selection of input models according to the testing goals and purposes has a very important role in the success of this approach.

In this paper, we focus on different input models of MBT and represent a classification framework for them. The classification is performed based on the information provided by test models, and also their potential to generate test cases. We compared and showed the strengths and weaknesses of test models according to their effectiveness for generating test cases, and showed which models are more appropriate for covering the desired testing target.

References

1. Sabbaghi, A., H.R. Kanan, and M.R. Keyvanpour, FSCT: A new fuzzy search strategy in concolic testing. Information and Software Technology, 2019. 107: p. 137-158.
2. Sabbaghi, A. and M. Keyvanpour, A novel approach for combinatorial test case generation using multi objective optimization, in Computer and Knowledge Engineering (ICCKE), 2017 7th International Conference on. 2017, IEEE.
3. Sabbaghi, A. and M.R. Keyvanpour, A Systematic Review of Search Strategies in Dynamic Symbolic Execution. Computer Standards & Interfaces, 2020: p. 103444.
4. Sabbaghi, A., M.R. Keyvanpour, and S. Parsa, FCCI: A fuzzy expert system for identifying coincidental correct test cases. Journal of Systems and Software, 2020: p. 110635.
5. Ahmad, T., et al., Model-based testing using UML activity diagrams: A systematic mapping study. Computer Science Review, 2019. 33: p. 98-112.
6. Hessel, A., et al., Testing real-time systems using UPPAAL, in Formal methods and testing. 2008, Springer. p. 77-117.
7. Ali, S., et al., A state-based approach to integration testing based on UML models. Information and Software Technology, 2007. 49(11): p. 1087-1106.

8. Sabbaghi, A. and M.R. Keyvanpour. State-based models in model-based testing: A systematic review. in Knowledge-Based Engineering and Innovation (KBEI), 2017 IEEE 4th International Conference on. 2017. IEEE.
9. Panda, N., A.A. Acharya, and D.P. Mohapatra, Regression testing of object-oriented systems using UML state machine diagram and sequence diagram. International Journal of Computing Science and Mathematics, 2020. 12(2): p. 132-146.
10. Khalifa, E.M., D. Jawawi, and H.A. Jamil, An Efficient Method to Generate Test Cases From UML-USE CASE DIAGRAM. 2019.
11. Assunção, W.K.G., et al., A multi-objective optimization approach for the integration and test order problem. Information Sciences, 2014. 267: p. 119-139.
12. Shirole, M. and R. Kumar, Constrained permutation-based test scenario generation from concurrent activity diagrams. Innovations in Systems and Software Engineering, 2021: p. 1-11.
13. Swain, S.K., D.P. Mohapatra, and R. Mall, Test case generation based on use case and sequence diagram. International Journal of Software Engineering, 2010. 3(2): p. 21-52.
14. Mallick, A., N. Panda, and A.A. Acharya, Generation of test cases from uml sequence diagram and detecting deadlocks using loop detection algorithm. International Journal of Computer Science and Engineering, 2014. 2: p. 199-203.
15. Ansari, G.A., Use of Firefly Algorithm in Optimization and Prioritization of Test Paths Generated from UML Sequence Diagram. International Journal of Computer Applications, 2017. 975: p. 8887.
16. Dehimi, N.E.H. and F. Mokhati. A Novel Test Case Generation Approach based on AUML sequence diagram. in 2019 International Conference on Networking and Advanced Systems (ICNAS). 2019. IEEE.
17. Khandai, M., A.A. Acharya, and D.P. Mohapatra. A novel approach of test case generation for concurrent systems using UML Sequence Diagram. in 2011 3rd International Conference on Electronics Computer Technology. 2011. IEEE.
18. Samuel, P., R. Mall, and P. Kanth, Automatic test case generation from UML communication diagrams. Information and software technology, 2007. 49(2): p. 158-171.
19. Abdurazik, A. and J. Offutt. Using UML collaboration diagrams for static checking and test generation. in International conference on the unified modeling language. 2000. Springer.
20. Nayak, A. and D. Samanta, Automatic test data synthesis using uml sequence diagrams. Journal of Object Technology, 2010. 9(2): p. 75-104.
21. Nebut, C., et al., Automatic test generation: A use case driven approach. IEEE Transactions on Software Engineering, 2006. 32(3): p. 140-155.
22. Straszak, T. and M. Śmiałek, Model-driven acceptance test automation based on use cases. Computer Science and Information Systems, 2015. 12(2): p. 707-728.
23. Bhuyan, P., A. Ray, and M. Das, Test Scenario Prioritization Using UML Use Case and Activity Diagram, in Computational Intelligence in Data Mining. 2017, Springer. p. 499-512.
24. Hamza, Z.A. and M. Hammad. Generating Test Sequences from UML Use Case Diagram: A Case Study. in 2020 Second International Sustainability and Resilience Conference: Technology and Innovation in Building Designs (51154). 2020. IEEE.
25. Sieverding, S., C. Ellen, and P. Battram, Sequence diagram test case specification and virtual integration analysis using timed-arc Petri nets. arXiv preprint arXiv:1302.5170, 2013.
26. Cartaxo, E.G., F.G. Neto, and P.D. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. in 2007 IEEE International Conference on Systems, Man and Cybernetics. 2007. IEEE.
27. Faria, J.P. and A.C. Paiva, A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets. International Journal on Software Tools for Technology Transfer, 2016. 18(3): p. 285-304.
28. Bouabana-Tebibel, T. and S.H. Rubin, An interleaving semantics for UML 2 interactions using

Petri nets. Information Sciences, 2013. 232: p. 276-293.
29. Kaur, A. and V. Vig, Automatic test case generation through collaboration diagram: a case study. International Journal of System Assurance Engineering and Management, 2018. 9(2): p. 362-376.
30. Dobing, B. and J. Parsons, How UML is used. Communications of the ACM, 2006. 49(5): p. 109-113.
31. Andrews, A., et al., Test adequacy criteria for UML design models. Software Testing, Verification and Reliability, 2003. 13(2): p. 95-127.
32. Briand, L. and Y. Labiche, A UML-based approach to system testing. Software and systems modeling, 2002. 1(1): p. 10-42.
33. Wang, C., et al. Automatic generation of system test cases from use case specifications. in Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015.
34. Ryser, J. and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. in Proc. 12th International Conference on Software and Systems Engineering and their Applications. 1999.
35. Gutiérrez, J.J., et al. Visualization of use cases through automatically generated activity diagrams. in International Conference on Model Driven Engineering Languages and Systems. 2008. Springer.
36. Sarmiento, E., et al., Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets. Electronic Notes in Theoretical Computer Science, 2016. 329: p. 123-148.
37. Ding, Z., M. Jiang, and M. Zhou, Generating petri net-based behavioral models from textual use cases and application in railway networks. IEEE Transactions on Intelligent Transportation Systems, 2016. 17(12): p. 3330-3343.
38. Prasanna, M. and K. Chandran, Automatic test case generation for UML object diagrams using genetic algorithm. Int. J. Advance. Soft Comput. Appl, 2009. 1(1): p. 19-32.
39. Sarma, M., D. Kundu, and R. Mall. Automatic test case generation from UML sequence diagram. in 15th International Conference on Advanced Computing and Communications (ADCOM 2007). 2007. IEEE.
40. Cavarra, A., et al. Using UML for automatic test generation. in Proceedings of ISSTA. 2002. Citeseer.
41. Shanthi, A. and D.G.M. Kumar, Automated test cases generation for object oriented software. Indian Journal of Computer Science and Engineering, 2011. 2(4): p. 543-546.
42. Mondal, S.K. and H. Tahbildar, Automated test data generation using fuzzy logic-genetic algorithm hybridization system for class testing of object oriented programming. International Journal of Soft Computing and Engineering, 2013. 3(5): p. 40-49.
43. Khiabani, A. and A. Sabbaghi. PHGA: Proposed hybrid genetic algorithm for feature selection in binary classification. in 2017 9th International Conference on Information and Knowledge Technology (IKT). 2017. IEEE.
44. Le Traon, Y., et al., Efficient object-oriented integration and regression testing. IEEE Transactions on Reliability, 2000. 49(1): p. 12-25.
45. Zhang, Y., et al., An approach of class integration test order determination based on test levels. Software: Practice and Experience, 2015. 45(5): p. 657-687.
46. Arora, P.K. and R. Bhatia, Agent-based regression test case generation using class diagram, use cases and activity diagram. Procedia Computer Science, 2018. 125: p. 747-753.
47. Kamonsantiroj, S., L. Pipanmaekaporn, and S. Lorpunmanee. A memorization approach for test case generation in concurrent uml activity diagram. in Proceedings of the 2019 2nd International Conference on Geoinformatics and Data Analysis. 2019.
48. Hashmani, M.A., M. Zaffar, and R. Ejaz, Scenario based test case generation using activity diagram and action semantics, in Human Factors in Global Software Engineering. 2019, IGI Global. p. 297-321.

49. Jaffari, A. and C.-J. Yoo, An Experimental Investigation into Data Flow Annotated-Activity Diagram-Based Testing. Journal of Computing Science and Engineering, 2019. 13(3): p. 107-123.
50. Sun, C.a., et al., A transformation-based approach to testing concurrent programs using UML activity diagrams. Software: Practice and Experience, 2016. 46(4): p. 551-576.
51. Fernandez-Sanz, L. and S. Misra, Practical application of UML activity diagrams for the generation of test cases. Proceedings of the Romanian academy, Series A, 2012. 13(3): p. 251-260.
52. Kurth, F., S. Schupp, and S. Weißleder. Generating test data from a UML activity using the AMPL interface for constraint solvers. in International Conference on Tests and Proofs. 2014. Springer.
53. Murata, T., Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 1989. 77(4): p. 541-580.
54. Nayak, A. and D. Samanta, Synthesis of test scenarios using UML activity diagrams. Software & Systems Modeling, 2011. 10(1): p. 63-89.
55. Linzhang, W., et al. Generating test cases from UML activity diagram based on gray-box method. in 11th Asia-Pacific software engineering conference. 2004. IEEE.
56. Lei, B., L. Wang, and X. Li. UML activity diagram based testing of java concurrent programs for data race and inconsistency. in 2008 1st International Conference on Software Testing, Verification, and Validation. 2008. IEEE.
57. Sun, C.-a. A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications. in 2008 32nd Annual IEEE International Computer Software and Applications Conference. 2008. IEEE.
58. Kim, H., et al. Test cases generation from UML activity diagrams. in Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007). 2007. IEEE.
59. Chen, M., et al., UML activity diagram-based automatic test case generation for Java programs. The Computer Journal, 2009. 52(5): p. 545-556.
60. Ye, N., et al. Automatic regression test selection based on activity diagrams. in 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion. 2011. IEEE.
61. Dahiya, S., R.K. Bhatia, and D. Rattan, Regression test selection using class, sequence and activity diagrams. IET Software, 2016. 10(3): p. 72-80.
62. Chen, M., P. Mishra, and D. Kalita. Coverage-driven automatic test generation for UML activity diagrams. in Proceedings of the 18th ACM Great Lakes symposium on VLSI. 2008.
63. Vieira, M., et al. Automation of GUI testing using a model-driven approach. in Proceedings of the 2006 international workshop on Automation of software test. 2006.
64. Teixeira, F.A.D. and G.B. e Silva, EasyTest: An approach for automatic test cases generation from UML Activity Diagrams, in Information Technology-New Generations. 2018, Springer. p. 411-417.
65. Sharma, C., S. Sabharwal, and R. Sibal, Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams. arXiv preprint arXiv:1410.4838, 2014.
66. Sapna, P. and H. Mohanty. Automated scenario generation based on uml activity diagrams. in 2008 International Conference on Information Technology. 2008. IEEE.
67. Arora, V., R. Bhatia, and M. Singh, Synthesizing test scenarios in uml activity diagram using a bio-inspired approach. Computer Languages, Systems & Structures, 2017. 50: p. 1-19.
68. Sumalatha, V.M. and G. Raju, Uml based automated test case generation technique using activity-sequence diagram. International Journal of Computer Science Applications, 2012. 1(9).
69. Kumar, R. and R.K. Bhatia. Interaction diagram based test case generation. in International Conference on Computing and Communication Systems. 2011. Springer.

70. Sarma, M. and R. Mall. Automatic test case generation from UML models. in 10th International Conference on Information Technology (ICIT 2007). 2007. IEEE.

71. Swain, S.K., D.P. Mohapatra, and R. Mall, *Test Case Generation Based on State and Activity Models.* J. Object Technol., 2010. **9**(5): p. 1-27.

72. Swain, R.K., et al., *Prioritizing test scenarios from UML communication and activity diagrams.* Innovations in Systems and Software Engineering, 2014. **10**(3): p. 165-180.

73. Pilskalns, O., et al., *Testing UML designs.* Information and Software Technology, 2007. **49**(8): p. 892-912.

74. Pilskalns, O., et al. Rigorous testing by merging structural and behavioral UML representations. in UML. 2003. Springer.

1202610200