Model-Based Testing for Contractual Software using Aspects

Bouchaib Falah, Farah Boukfal, Basma Iraqi

1Assistant Professor, School of Science and Engineering, Al Akhawayn University in Ifrane, Ifrane, Morocco
2Graduate Student, School of Science and Engineering, Al Akhawayn University in Ifrane, Ifrane, Morocco
3Graduate Student, School of Science and Engineering, Al Akhawayn University in Ifrane, Ifrane, Morocco

b_falah@aul.ma, f_boukfal@aul.ma, b_iraqi@aul.ma

ABSTRACT

With the development and increasing adoption of the Unified Modeling Language (UML), model-based testing has become one of the most researched and implemented testing areas. Among the different UML models, sequence diagrams are the most commonly adopted for representing the system interactions and communicating the functional requirements, and thus are often used to enable test case generation for a specific program. On the other hand, the “Design by Contract” technique consists of specifying the communication between the system components and the responsibilities and expectations of each one of them. This paper presents an approach to improve test case generation in model-based testing by encapsulating the different system constraints in one contract as an aspect and testing it independently as a unit. The testing of the different system base classes will be then performed through establishing a woven sequence diagram that is simpler to understand and which combines both base classes and aspects’ interactions, enabling an easier and more fruitful test generation process.

Keywords: Model-based testing, aspect-oriented programming, UML, design by contract, sequence diagram, testing

1. INTRODUCTION

Model-based testing is a style of software testing which relies on explicit definition of models to generate test cases. Simply put, a model of software is a depiction of its behavior [6]. This latter can be denoted through the definition of acceptable input, expected output, activities leading to the outcome, data flow through the different actions or logic behind the different methods and functionalities. In fact, models have always been used in software testing. In order to make decisions about the problems that should be addressed in a program and the test cases which are needed to be executed, software testers constantly rely on their own understanding of the software, which is, in a sense, their mental picture or model of the system under test [7]. This being said, the novelty about model-based testing consists in moving the models from the tester’s mind to concrete and formal graphs, diagrams or charts. Therefore, the requirement for model-based testing is to design reusable and shareable models which provide precise and clear description of the program under test [6].

Model-based testing has become particularly popular with the emergence of UML. UML models are of increasing importance in the design and development of any software, especially object-oriented software, as they have been successful in taking models from personalized graphical representations to established diagrams compliant to structured language [6]. Sequence diagrams, among the UML models, are commonly used in software testing. They are particularly interesting because they provide a clear description of the system interactions [10]. They are very useful in communicating the functional requirements and are of great value to black-box testing techniques, especially model-based testing. This being said, model-based testing, supported by UML, is becoming one of the most common research topics in software testing as its illustrations satisfy all the prerequisites for good and fruitful assessment and validation.

In addition to the multiple method calls illustrated in sequence diagrams, there are also guards to most of the message interactions as shown in Fig.1. These guards represent the constraints of the functions to be invoked.

![Figure 1. Sequence Diagram Template](image-url)

The more there are guards in the sequence diagram, the more branching is illustrated. As a result, a large number of paths should be tested since every branch determines a unique path through the system components. Having that sequence diagrams describe complex interactions between multiple objects in the system, the use of guards makes generating scenarios from sequence diagrams a tedious task. Fig. 2 illustrates a sequence diagram that describes the “buy product” use case from a vending machine.
Based on the figure, it is clear that as guards are incorporated into the sequence diagram, the illustration become less intuitive and more ambiguous. This extra complexity defeats the foundations and basic objectives (clarity and simplicity) of a sequence diagram [9]. It also makes the test case generation process complex for any model-based testing technique. In order to illustrate constraints in a system, we suggest using the Design by Contract concept by encapsulating the different system constraints as contracts, implementing them as aspects, and testing them separately as independent units. The use of the aspect-oriented paradigm for model-based software testing will significantly help in simplifying the testing process and enhancing its performance and efficiency while taking full advantage of the features of sequence diagrams.

2. DESIGN BY CONTRACT

Design by Contract (DbC) is a technique in object-oriented software design adopted to ensure many quality attributes of software, such as reliability, conformance and reusability [3]. The key concepts of DbC consist of clearly stipulating the communication means between the different components of a system to the very last detail, and delineating the responsibilities and expected results of these communication processes. These concepts, or mutual obligations are, what we referred to as, “contracts” and are checked through “assertions”. These latter are Boolean expressions that are inserted in to the code and that should be ‘True’ at all times. Otherwise, bugs are present in the software and need to be reported to the application’s users. On the other hand, a “contract” is a formalization of the obligations and benefits of every system component and is represented by a set of:

- **Preconditions**: Obligations that should be fulfilled prior to being able to invoke the component.
- **Postconditions**: Expected results after the component’s execution.
- **Invariants**: Conditions that must remain unchanged after the component’s execution.
In other words, every method in a class should provide its expected functionality by [3]:

- Verifying that its preconditions are guaranteed prior to its call from some component.
- Guaranteeing that its postconditions are realized upon its exit.
- Maintaining the correctness of invariants throughout its execution.

The design by contract approach perceives the contract to be an essential requirement for ensuring the correctness and consistency of the software. As a result, it states that the contract should be a necessary part of the software design process, especially when dealing with external components of applications. The contract of every method generally includes the following: (1) Preconditions, (2) Postconditions, (3) Invariants, (4) Acceptable and inacceptable input values or types, (5) Return values or types, (6) Errors and exceptions that can occur, (7) Side effect (modification incurred to one or more methods’ state), and optionally (8) The expected performance (required time and space taken).

Applying design by contract to one’s program is very challenging. Inserting the preconditions and postconditions directly into the program’s code can have serious drawbacks on its modularity and reusability as it mixes the business logic code with the nonfunctional requirements’ code. Trying to add, modify or remove these contract’s elements’ assertions cannot be performed without altering all of the program’s code, thus making the code inflexible. This problem can be solved by meeting the following requirements [3]:

- **Transparency**: The preconditions and postconditions code should not be mixed with the underlying application’s business code.
- **Reusability**: The components of the application should be developed in a way that ensures high level of modularity and thus reusability.
- **Flexibility**: The modules representing the “assertions” should be added, modified and deleted gracefully.
- **Simplicity**: The modules representing the “assertions” should be implemented and specified using very simple and clear syntax.

Consequently, one way of simplifying the definition of clear contracts between the different components would be to consider the concept of aspects, which is introduced in the Aspect Oriented Programming paradigm. Representing the different contracts as aspects enables eliminating the production of unexpected results and the manifestation of bugs while keeping the code clean, flexible, and reliable.

### 3. ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) is a programming paradigm which objective is to increase the modularity of a program or application by providing the possibility of separating the “cross-cutting” concerns from core concepts and business processes of the system [4]. Cross-cutting concerns are common secondary requirements that are shared between many classes and cannot be easily decomposed from the rest of the system, resulting most of the cases in their code being duplicated. In AOP, these concerns are abstracted and encapsulated to form the basis of “aspects”, which are common features that are linked to many other parts of the system, and thus are scattered across many methods and classes. In other words, AOP allows the dynamic modification of an application’s static model by including code that is required to fulfill the non-functional requirements without modifying the underlying code, and while keeping the additional code saved in a different single location instead of being scattered across all methods and components that necessitate its integration [4].

To better understand the concept of AOP, we should first be familiar with the following concepts:

- **Cross-cutting concerns**: Represents the secondary requirements that are shared between the different classes of the system. They correspond to behaviors that ‘cut’ across multiple points in the object models.
- **Point-cut**: Represents the set of points of execution (also called join points) in the code where a cross-cutting concern needs to be applied (e.g. Entry and exit points of a method).
- **Advice**: Represents the actual code that will be applied to the existing model to answer the cross-cutting concerns. An advice could be of type before (runs before its join point), after (runs after its join point), or around (runs in place of, or around, its join point).
- **Aspect**: Refers to the combination of defining a point-cut and providing the corresponding correct advice.

As an example, one common cross-cutting concern that we can consider is metrics [2]. Although being able to collect metrics’ information of an application is important, they do not represent anything of concern to the application’s object model as they are irrelevant to its actual functionalities. The metrics neither denote any of the system classes nor realize any of the application business rules. They are simply orthogonal [2]. A metric aspect in Java (to measure the amount of time needed to invoke a particular method) is illustrated in Fig. 3.
To apply the aforementioned aspect code, we should identify when and where to execute it, thus identify the pointcut. Fig. 4 shows a possible pointcut for our metrics’ aspect code in XML:

```xml
<bind>
  <pointcut id="metrics-pointcut" execute="true">
    <on target="com.mc.Metrics" method="withdraw"/>
  </pointcut>
</bind>
```

![Figure 4. Metrics Aspect Pointcut](image)

The point-cut definition implies that the metric aspect’s class will be invoked when calling the method “withdraw” defined in the “BankAccount” class. As specified earlier, the AOP paradigm enables the use of the metrics’ aspect defined throughout the multiple classes and methods of any object-oriented program, while ensuring consistency, modularity, reusability and reliability without complicating the code representing the business logic. The aspects can hence be modified and extended separately without affecting the normal workings of the program.

In relation with the previously defined DbC, AOP represents the best approach that can be adopted to represent the different system constraints, which are the preconditions, postconditions, and invariants. In order to satisfy the above-mentioned DbC solution requirements, we considered the implementation of three distinct components which are shown in Fig. 5 and are defined as:

- The application that does not contain any element that is related to the contract
- The contract implementation that specifies the different preconditions, postconditions, and invariants
- The ‘bridge’ object between the application’s code and the contract that applies the contract at the right execution moment and following a correct logic. This represents the aspect.

These components guarantee the implementation of a flexible solution that allows its users to apply and/or remove contracts without altering the contract or the application code in a very transparent manner.

4. TESTING TECHNIQUE

Starting from the concept of design by contract, representing all the system constraints in one aspect allows encapsulating the testing of all of these elements in a single attempt, requiring the unit testing of the aspect. This advantage is clearly illustrated in contrasting Fig. 6 and Fig. 7, which represent the standard implementation of a contract and the aspect implementation of a contract, respectively.

![Figure 6. Standard Implementation of the Contract](image)
Converting the woven sequence diagram into an AOF tree, we transform it into an AOF graph that focuses on aspects, objects, and messages that interact directly with the method to be tested. In order to construct the graph, coverage criteria should be applied for the sequence diagrams. We consider three type of coverage: condition coverage [12], polymorphic coverage [12], and loop coverage [12]. Condition coverage enables identifying whether some branch of conditional behavior has been affected by the execution of an aspect. Polymorphic coverage intends to exercise every potential object type (subclasses) that might be affected by an aspect. Loop coverage verifies whether or not aspects have been triggered.

The AOF graph is a directed acyclic graph defined as a \((V, E)\) where \(V\) is a set of vertices and \(E\) is the set of edges between the vertices. Every vertex has four fields: (1) two indices (the level number and node number within that level), (2) a class or aspect identity, (3) messages, (4) constraints. Having the woven sequence diagram of the method we would like to test, we start by building a vertex for every behavior block that interacts with the method and assigning to every vertex its corresponding fields' values. A behavior block refers to a sequence of adjacent messages to a same object, a conditional structure or a repetition structure. We, then, connect each pair of adjacent vertices with an arc depending on their relationship in the sequence diagram (sequential or not). For every vertex, we perform a set of actions. If:

1. The vertex’ behavior is conditional, it is replaced with two vertices labeled with node 1 if the condition is true, and 2 otherwise.
2. The behavior affected by the aspects is polymorphic, the vertex is expanded at the same level depending on the different subclasses of that class.
3. The behavior is a repetition, the vertex is expanded as in the first case.

Once the AOF graph is generated, the AOF tree is created by performing the depth-first search traversal of the graph. As stated earlier, every path in the AOF tree from the root to a leaf node represents a test template. The added value of performing the unit testing of aspects prior to weaving the sequence diagrams is guaranteeing that the aspect is functioning properly (executing its advice correctly). As a result, any fault that appears during testing of the AOF paths would be ascribed to the base classes only.

5. CASE STUDY

In order to assess our proposed testing technique, we applied it to a car rental system. To simplify the example, only two classes are represented in the aspetical class diagram in Fig. 8, each of which has a joint point to an “AccessControl” aspect.

---

Figure 7. Aspect Implementation of the Contract

Every aspect-oriented model consists of class diagrams, aspect diagrams and sequence diagrams. Our idea consists of performing unit testing of the aspects representing the contracts, as if they were independent components. Once the aspects are validated, we proceed with weaving the advice’s sequence diagram into the method’s sequence diagram (methods on which the given advice is applied), and generating an Aspect-Object Flow tree from which test cases will be generated to check whether or not an aspect-oriented program answers its expected cross-cutting behaviors [11]. An aspect-oriented model consists of class/aspect diagrams and sequence diagrams for class methods and aspect advice. Cross-cutting concerns can be represented in a class/aspect diagram where every aspect has three elements: (1) the pointcut (set of join points), (2) the advice, and (3) the introduction. To create the woven diagram, we perform the following steps [11]:

1. Identify the different join points and corresponding aspects in the sequence diagram of the method based on the class/aspect diagrams defined.
2. Obtain the sequence diagram of the advice on each join point.
3. Integrate the advice sequence diagram at the join point into the base class sequence diagram according to the advice type (before, after, or around).

Once the woven sequence diagram is created, the next step consists of generating the corresponding test cases to test interactions between the different classes and aspects. This is done by following the next steps [11]:

4. Converting the woven sequence diagram into an Aspect-Object Flow tree (AOF) according to the coverage criteria for sequence diagrams. Every path in the AOF tree from the root to a leaf node represents a test template.
5. Collecting, for each test template, the constraints involved in the path.
6. Assigning values or objects to the input variables to create concrete test cases.
Having that the requirements of the car rental system state that the price attribute inside the car is password protected, the aspect is incorporated in order to ensure that the password is correct before getting the price. Apart from checking this precondition, the aspect should also verify that the postcondition, which is getting non Null values for the price, holds, as well as retain the state invariant, which is the connected/logged state for the user who entered the password in the first time in order to avoid entering the password multiple times whenever the user needs to check some prices.

In order to generate the flow graph, let’s consider the following features of the Car rental system:

- The charge to be paid for renting a car can change if the rental period requested is within the discounted period or not
- Price is an abstract class. Its concrete classes are StudentPrice, RegularPrice and NewCarPrice.

Applying the instructions previously described to generate the flow graph, the "get charge" graph would be as shown in Fig. 9.

Traversing this graph in a depth-first search fashion would lead to the generation of the flow tree in Fig. 10, which will be the basis for test case generation.
As an example, if test cases are needed for the path \((1.1 - 2.1 - 3.1 - 4.1)\), the objects that will be used are rental, such that \(\text{periodRented} < \text{discountedPeriod}\), and the StudentPrice, so that the price to be computed is for the Student category.

6. RELATED WORK

Several papers have already suggested black-box testing techniques based on UML sequence diagrams, but none of them has already taken the issue of guards’ constraints into consideration while developing their testing approach.

In his paper entitled “Model-Driven Testing Approach based on UML Sequence Diagram”, Grigorjevs [5] suggests a method to transform sequence diagrams to UML testing profiles (UML extension to support testing artifacts). To achieve this objective, model-driven architecture principles are applied in order to generate a model for easy test case generation. The transformation is performed by processing the source sequence diagrams in XMI (UML model representation in XML), and the transformation rules are put in practice using an automated tool [5].

Nayak and Samanta [8] have also worked on the sequence diagram-based testing. In their “Synthesis of Test Scenarios using UML Sequence Diagrams” paper, the main goal was to identify test scenarios by determining flows of control in sequence diagrams. Therefore, test cases generated are not only representative of a single scenario; they also denote how multiple scenarios relate. This paper suggests building a directed scenario graph whose nodes represent blocks of interaction within a particular scenario. Once the graph is successfully developed, further test cases are generated by traversing the graph and walking through its directed edges.

Last, there are many other papers which focus on the integration of sequence diagrams with other UML diagrams in order to generate test cases. The most common combination is the one that gathers sequence and state diagrams. For Bertolino and Marchetti [1], this integration allows deriving a comprehensive test suite as early as possible in the software development cycle. In their paper entitled “Introducing a Reasonably Complete and Coherent Approach for Model-based Testing”, Bertolino and Marchetti [1] claim that this arrangement is very efficient in solving all limitations of other model-based testing techniques. Thanks to this combination, the authors assert that the approach suggested by the paper considers accuracy and effort reduction since the compound solution allows making up the restraints of each model by the other and taking full advantage of the qualities of both diagrams.

7. CONCLUSION AND FUTURE WORK

The aspect-oriented paradigm has been viewed, since its beginning, as a revolution to software design methodologies, but it has rarely been considered for software testing. However, for testing styles like model-based testing, where there is a strong correlation with the design models, it is very interesting to take advantage of the features of AOP and make improvements to testing techniques accordingly. This being said, this paper suggests implementing design by contract using AOP where every contract is represented as an aspect. The system will be then tested by performing aspects’ unit testing and generating simpler sequence diagrams, in which fewer guards are used. Finally, flow graphs and trees are used to make scenarios’ extraction from the diagrams and test case generation a more straightforward process.

As future work, since the workings of this testing technique are all based on manual work, it would be interesting to automate the whole process, starting from the aspects’ unit testing to the generation of the flow graph and tree and the extraction and execution of test cases.

REFERENCES


