Automatically Generating Test Cases Using UML Structure Diagrams

Name Withheld
Department of Computer Science
University of Delaware
Newark, DE, USA

Name Withheld
Department of Computer Science
University of Delaware
Newark, DE, USA

December 10, 2009

1 Abstract

We are proposing a new way to utilize UML diagrams of programs to automatically generate test cases for those programs. Previous work has investigated the use of UML diagrams as a means of generating test data in various ways. Our method combines a few of the most effective of these methods to create a broader algorithm. We feel that our method will allow software developers to generate test cases without any additional effort because they are utilizing a UML model that they have already had to create to develop their software. We test the effectiveness of our method by evaluating its performance on a variety of large Java projects and comparing its results to those of the current state-of-the-art methods. We plan to run our evaluation on a variety of open source and industry software and a large banking software project to prove that our method is more effective than the separate methods that we are combining to create our algorithm.

2 Introduction

Software testing is extremely costly, where a large amount of time is spent in creating manual test cases. To avoid the manual laborious task of software testing, researchers have focused on automating software testing to save cost and time. Every time a user executes a program, it gets tested. So testing is to be done in a way so that the highest possible number of errors can be reduced systematically. There have been many attempts to develop an efficient and effective ways to automate software testing to save developers time significantly. We feel that the use of the Unified Modeling Language to automatically generate test cases will allow for the best coverage and requirement-aware test case to be generated while saving developers valuable time.

UML is a standardized modeling language used extensively in the field of software engineering. The best feature of UML is that, it explains all the functionalities of a system with the help of diagrams. This diagrammatic representation is fairly easy to understand for developers while they are working on an application as it includes even the minute details of the application. The latest version of UML has 14 diagrams which are categorized based on two hierarchy levels.[9]

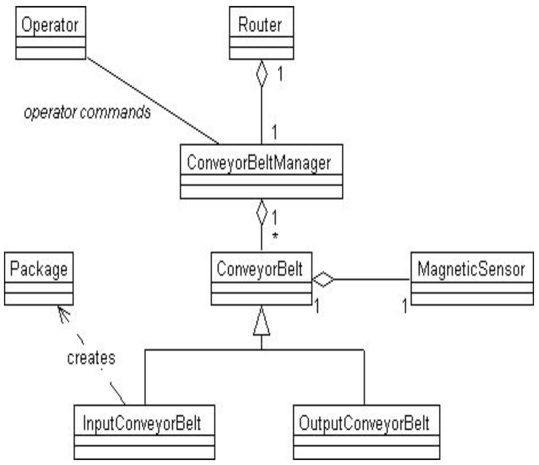
- Behavior Diagram:
  Behavior diagrams provide an explanation of the system as a whole. They models the behavior of system in the form of diagrams. Some of the basic examples of behavior diagrams are Activity diagrams, State Machine Diagrams and Use Case Diagrams.

- Structure Diagrams:
  Structure Diagrams are entirely different form the Behavior diagrams in a way that they do

not model the system as a whole, rather they model what should be there inside the system that is being modeled. Our proposal is based on UML Structure Diagrams, for example: Class Diagram and Object Diagram.

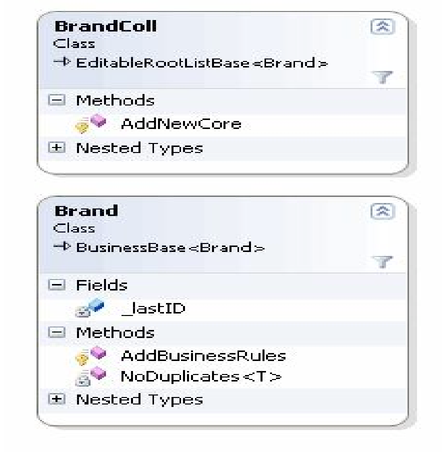
In this proposal we will be concentrating on two specific UML Structure Diagrams:

- Class Diagram:
  Class Diagrams are UML structure diagrams which explain the system by showing its classes and relationship between them and attributes. Class diagrams can be read through Java scripts and by parsing through a document which explains all its entities, relationship in the form of a grammar. Figure 1 shows an example Class Diagram.



[Fig 1: Example of Class Diagram]

- Object Diagram:
  An object diagram is another kind of UML structure diagram that focuses on features related to an object like Object instances and its attributes links between those instances. Similar to class diagram an Object Diagram can be read using java scripts and by parsing through the document that lists all the features in the form of a grammar. Figure 2 shows an example Object Diagram.



[Fig 2: Example of Object Diagram]

The use of UML diagrams allows us to generate test cases early during the software development process. This functionality will allow us to identify the bugs and problems in the overall design of the code early. Hence, this approach will save more time. The use of UML diagrams limit us to testing of software written in Object Oriented languages, but a large number of enterprise level programs are based on this methodology today.

3 Background

The use of UML diagrams to automatically generate test cases began with the work of Chevalley and Thkvenod-Fosse in 2001[3] when they used statistical functional testing to generate test cases from UML state diagrams using transition coverage as the testing criterion. Their work looked forward to working with java and acted as the starting point for further research into the use of UML in automatic test case generation.

Previous work from Samuel, Mall, and Kanth[2] on the automatic generation of test cases from UML communication diagrams investigates transforming a UML sequence diagram into a tree representation and then information is extracted based on post-order traversal and selecting conditional predicates. This method makes it possible to generate cluster level test cases even before the code is written. Their approach has been tested rigorously based on branch and path coverage, full predicate coverage and boundary criteria. Later research from Karma, Kundu, and Mall[1] has proposed a solution that transforms a UML sequence diagram into a graph called the sequence diagram graph (SDG) and augments the SDG nodes with different information necessary to compose test vectors and then transverses this SGD to automatically create test cases. Their method was tested on C++ examples and properly generated test cases with their algorithm that has an n squared worst case running time.

Gnesi, Latella, and Massink[4] investigated generating test cases by generating a semantic model for a behavioral subset of UML state charts using an algorithm based on Algebra and Lambda Calculus that automatically generates test cases using inputs, set of events and possible outputs of state charts. Their algorithm generates a single test case by selecting an event non-deterministically at each call.

A tool, the GenTCase tool, created by Nebut, Fleurey, Traon, and Jezequel[7] works after finalizing a use-case diagram and then searching through the use-case to extract information to aid in generating test cases. Functional and System requirements are extracted using meta-data index search which is matched through a database of existing keywords to generate test cases.

Past work has also looked into generating test Cases using activity diagrams from UML Mingsong, Xizokang, and Xuandong[5] have worked on extracting information from the activity diagram and generating test cases using information like selecting the path coverage of an activity diagram, selecting the execution trace and selecting the activity and transition coverage. They implemented an algorithm to find these correlations in an activity diagram by considering an activity diagram as a petri-net. Their later work in the same field dealt with coverage-driven generation of test cases from activity diagrams where data is extracted from activity diagram through parsing and then provided to a fault generator and a formal model providing test cases as output. This allows for control flow from activity diagrams and data flow from test criteria to be used to extract structure and behavior of the specification [6].

Test Case generation using UML diagrams and models was a success, and soon many new researchers started developing more creative approaches. Most recent work by Lamancha, Usaola, and Velthius[8] has looked into using an automated testing framework to manage variability in the UML testing profile. They propose an extension to the UML testing profile to manage vulnerability in testing software and focus on the test case behavior represented by sequence diagrams and defines an extension to UML interactions for SPL. They also define a model to handle the variability in SPL testing, reusing the metamodel defined by the UML Testing Profile, currently testing using QVT which is a model transformation language.

However, all these research efforts prove that automatically generation of test cases has been well exploited in the case of behavioral UML diagrams. Thus, we hereby try propose a new approach which will not only focus on the system as a whole but will also look at what is inside the model that keeps it running. We will achieve this by exploring the testing based on structural UML diagrams.

4 Challenges and Goals

4.1 Challenges

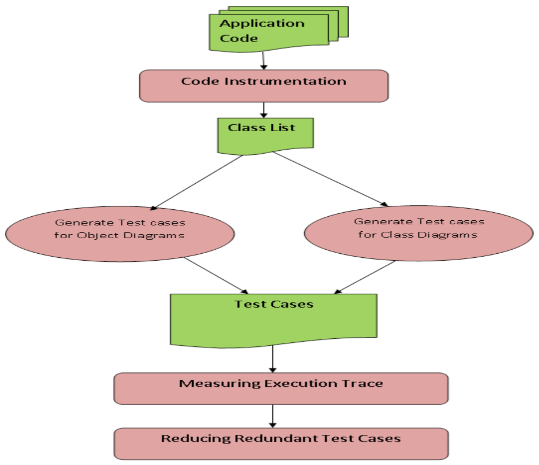
We face three major challenges when striving to automaticlay generate test cases for web applications through the use of UML. Our first challenge is that we still have a manual step of code instrumentation for our approach. Although this will be a small set which can be implemented by the developer given a set of instructions, it might become cumbersome while using a large application for testing. We are currently working to find a way to automate this process. Our next challenge arises due to the fact that Java is a complex language with a large amount of features. While automatically generating test cases for class diagrams we might face issues with abstract methods or classes or multiple inheritances. We need to make sure that all these features can be accessed easily from our approach. Lastly we have the issue that arises when a test case runs on a particular method it makes sure to test the functionality of only that method. Consider the method under test is calling another method, for which a test case is not defined. There is some work done before to handle such conditions while automatically generating test cases. [10]

4.2 Goals

Our three major goals for our approach are an overall reduction in testing time, a maximization of code coverage and and efficient implementation of our tool. Our approach will be concentrated on reducing the time required for creating and generating test cases, hence saving tremendous resources of development team. Code Coverage is a key factor in analyzing the quality of test cases. We plan to measure the branch coverage, statement Coverage and overall code coverage of our implementation. We also plan to integrate our approach into a tool which will provide an easy way for developers to implement our method on their own projects. This tool will focus on generating test cases and executing them in an efficient and developer friendly manner.

5 Proposed Research

Our approach is divided into six major sections including generating test cases and reducing irrelevant test cases. We plan to design and implement our approach for Java Applications as Java strictly follows paradigms for Classes and Objects. This technique will be implemented as two different algorithm which will be embedded in a tool for automation. Figure 3 provides an overview of our proposed approach


Application Code
Code Instrumentation
Class List
Generate Test cases for Object Diagrams
Generate Test cases for Class Diagrams
Test Cases
Measuring Execution Trace
Reducing Redundant Test Cases

[Fig 3: Proposed Approach]

5.1 Code Instrumentation

To capture the execution trace, we plan to instrument the code under test. Program instrumentation is a popular dynamic testing method. We plan to track the number of times the instrumented code runs. It will also provide us information used for reducing redundant test cases. We are planning to use the same approach as used by Mingsong et al[5] in Automatic test case generation for UML Activity diagram. The program under test will act as an input for this phase. The instrumented code will make sure that all the classes that are being accessed during execution get added to a Class List. We will use this generated set for creating test cases as explained in later phases. Code instrumentation needs to be done manually by the developers.

5.2 Test Cases for Object Diagrams

This step will be responsible for creating test cases as per the Object Diagrams. We will be using the Class List generated in previous step to achieve this. We will extract the features of all the classes mentioned in Class List by parsing through the required Java file. These features will mainly be the method names, their parameters and return types, return type for the class etc. By extracting all these features we are basically accessing Object Diagrams. To confirm the correctness of extracted parameters users can match the generated features with the given Object Diagram. All these features are represented in Object Diagrams as shown in Figure 2. We will then generate abundant test cases based on the extracted information. Class List can have certain class names more than once. This may lead to more number of test cases on the same class or method. We do not handle this issue here as we will be reducing the number test cases in further steps.

5.3 Test Cases for Class Diagrams

In this step we will concentrate on creating test cases based on Class Diagram. The approach used will be similar to the previous step. The only difference here will be that instead of extracting features of the class, we will be extracting the relationship of the class with other classes. By extracting the relationship between classes we will be having a mapping of class diagram. A user will be given the flexibility to compare the extracted information with given class diagrams. We can then create abundant test cases based on the related classes. Again as we will have classes more than one time in Class List we will keep generating test cases for them. These redundant test cases will be reduced in later steps.

5.4 Measuring Execution Trace

After running the test cases generated above we get the execution trace of the program under test. We extract this information with the help of instrumented code. By execution trace we mean, we calculate the number of time a method is called, a class gets instantiated etc. Using execution trace we analyze the control flow of the application. This information will help us in reducing the abundant test cases to a test suite with relevant test.

5.5 Reducing Redundant Test Cases

In this phase we will begin with the test cases generated by UML Object diagrams and then with Class diagrams. We will be having several test cases created for a single object diagram and class diagram. We will start with the test cases generated by Object diagrams and compare them with the test cases generated by class diagram. If we find that a test case from the Object diagram contradicts the test case of class diagram, we will alert the user that there is an inconsistency in their UML diagram. This conflict can lead to a problem with the theory or structure that their software is based upon. We will then union the set of test cases generated and will remove any redundant test case generated by a component, to make sure the tests will run in the most efficient manner. This algorithm will handle the multiple test cases generated for object and class diagram.
Our algorithm:

```
1. Gather test cases from information in class diagrams and place in test set A
2. Gather test cases from information in object diagrams and place in test set B
3. Search for components of A that conflict with components of B
4. If conflict = true, result=error
5. Search for components of B that conflict with components of A
6. If conflict = true, result=error
7. If result !=error ,return test set = union (set A, set B)
```

We will make sure while implementing this algorithm that the statement coverage and complete code coverage is not hindered.

5.6 Analysis of Behavior of Class Diagram

We will also provide a complete analysis on the behavior of Class diagrams. We can predict the efficiency and nature of testing on Class Diagrams. We analyze the different relationships in class diagrams and measure the complexity of application based on their class diagram. This behavior can be evaluated based on inheritance relationships and class relationships like association, aggregation, realization etc. This approach will be implemented by analyzing the code coverage and line coverage. We will evaluate this based on how reachable is the application code in case of high level dependencies between classes. We can assure that there has not been any tremendous work done on analyzing behavior of Class Diagram using application testing.

5.7 Tool Implementation

As a final stage of our approach we plan to implement this approach, as a real time tool. This tool will reduce the testing time to a large extent by performing automated testing based on Class Diagrams and Object Diagrams. Our tool will be used extensively for Java applications and will perform testing on a reduced set of most relevant test cases. For the Java platform the tool will be using the JUnit library for testing in Java. This will be responsible for running the tests on its own and providing users with the list of all test cases implemented as a test suite in Java.

6 Evaluation Plan

We will evaluate our method by assessing the code coverage achieved by the automatic test cases generated by our algorithm. Code coverage is the measure of amount of code covered by executing a test suite. As we are testing individual classes, objects and components, we plan to get a better coverage than achieved by testing behavior models. We have planned to perform the experiment on fifty Java projects selected from both the open source community and industry. These applications will have code ranging from 500 lines to 100000 lines and will include both projects commonly used in related studies and new applications we have found.

We have also planned to run our automatic test case generation tool on a live software application developed in Java for an investment banking firm. We have chosen a banking firm as they perform critical transaction all throughout. We plan to compare our results by testing the open source application with the other strategies implemented by researchers working on behavior models. To test the real time application we will collect data from the firm developing the investment banking application after they have completed the manual testing. We will compare as how close to manual testing our tool can reach.

The last portion of our evaluation will test our method's ability to find faults that we have seeded in our set of test programs. We will seed a variety of errors in our programs and record the total number of faults found correctly, false positive faults, and faults missed. We will also evaluate four other state of the art tools on these programs to determine if our method has higher precision and recall values than the other current state of the art methods.

We will also compare the results of our test to the different components of UML diagrams used alone to generate test cases to assure that the combination of these methods are statistically significant. We will only compare the ability to find seeded faults and the overall code coverage of these methods versus the ability to find seeded faults and the overall code coverage of our method because we are fairly sure that the our combined method will be slower than any one singular method attempted previously. We will require that our method does perform out method in a reasonable amount of time, preferably less than the sum of the time it would take to run each of the component methods individually.

Our approach is entirely different from Automatic Test Case Generation for UML activity diagram as mentioned in [5]. We are performing our analysis on Class and Object diagram whereas the former uses UML activity diagram. Activity diagrams provide information about system as a whole, and is a part of BlackBox Testing. Our approach is entirely based on Classes and Objects and can be considered as White Box testing. We will be using a modified version of the Code Instrumentation technique to reduce the irrelevant test cases from an algorithm first implemented by Mingsong et al.[5].

7 Foreseen Contributions

Our tool will allow users to more accurately test their applications and improve the speed of their testing to improve software roll out times and save development companies considerable funds during their testing phase. The use of our tool will be easy for users to integrate into their software development because it requires no extra work and relies completely on documentation and conventions that are already used in Java development.

We are hoping that our implementation of our algorithm will allow developers and testers to quickly and effectively create test cases that are only testing the specifications of the project instead of all possible cases. These black box tests are more efficient in time use than creating white box tests for the entire project to be tested. In the future we plan to expand this tool to work for multiple languages and add in more elements of UML information as we find that they add significance to the model through further expansion and testing of our tool.

References

[1] Monalisa Sarma, Debasish Kundu, Rajib Mall, *Automatic Test Case Generation from UML Sequence Diagram.* Advanced Computing and Communications, International Conference on, pp. 60-67, 15th International Conference on Advanced Computing and Communications (ADCOM 2007), 2007.

[2] Samuel, P., Mall, R., and Kanth, P. 2007. *Automatic test case generation from UML communication diagrams.* Inf. Softw. Technol. 49, 2 (Feb. 2007), 158-171. DOI= http://dx.doi.org/10.1016/j.infsof.2006.04.001.

[3] Philippe Chevalley, Pascale Thvenod-Fosse, *Automated Generation of Statistical Test Cases from UML State Diagrams.* Computer Software and Applications Conference, Annual International, pp. 205, 25th Annual International Computer Software and Applications Conference (COMPSAC'01), 2001.

[4] Stefania Gnesi and Diego Latella and Mieke Massink and Via Moruzzi and I Pisa. *Formal test-case generation for uml statecharts.* Proc. 9th IEEE Int. Conf. on Engineering of Complex Computer Systems. 2004. pp.75-84. IEEE Computer Society.

[5] Mingsong, C., Xiaokang, Q., and Xuandong, L. 2006. *Automatic test case generation for UML activity diagrams.* In Proceedings of the 2006 international Workshop on Automation of Software Test (Shanghai, China, May 23 - 23, 2006). AST '06. ACM, New York, NY, 2-8. DOI= http://doi.acm.org/10.1145/1138929.1138931.

[6] Chen, M., Mishra, P., and Kalita, D. 2008. Coverage-driven automatic test generation for uml activity diagrams. In Proceedings of the 18th ACM Great Lakes Symposium on VLSI (Orlando, Florida, USA, May 04 - 06, 2008). GLSVLSI '08. ACM, New York, NY, 139-142. DOI= http://doi.acm.org/10.1145/1366110.1366145.

[7] Clementine Nebut, Franck Fleurey, Yves Le Traon, Jean-Marc J?z?quel, *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, vol. 32, no. 3, pp. 140-155, March, 2006.

[8] Lamancha, B.P.; Usaola, M.P.; Velthius, M.P., *Towards an automated testing framework to manage variability using the UML Testing Profile.* Automation of Software Test, 2009. AST '09. ICSE Workshop on , vol., no., pp.10-17, 18-19 May 2009.

[9] http://en.wikipedia.org - Unified Modeling Language, Diagram Overview

[10] Ciupa, Alex Pretschner, A. Leitner, Manuel Oriol, B. Meyer *On the Predictability of Random Tests for Object-Oriented Software.* Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation ICST workshop on,. pp 72-81, 2008