

AN ENVIRONMENT FOR NON-DUPLICATE TEST GENERATION FOR WEB BASED APPLICATION

Manjeet Kumar¹ and Dr. Rabins Porwa¹²

¹Research Scholar, Mewar University, Rajasthan
²Associate Professor, ITS, Mohan Nagar, Ghaziabad

ABSTRACT

Web applications quality, portability, consistency, interoperability and dependability are main parameters because software could block completely business. Internet based applications are very complex and heterogeneous also based on different kind of modules which can be implemented in a specific language, so testing is very much require. This paper represents the object oriented based environment. Web applications consist of Web related documents, class and objects and some different kinds of server like HTML, DHTML, and JAVA SCRIPT etc. This paper presents three kinds of layers of test: unit testing, integration testing and system testing. This paper also discusses the different techniques. This paper also based on the reverse engineering techniques which can be used to analyze the software which is probably connected with a model. It also consists a method of generate the test cases. This paper also describes software used to perform some results.

KEYWORD

Automated testing, class, object, oops

1. INTRODUCTION

The object oriented method to software development does is very much required for component based technique. If we talk about Object oriented software which is supposed to be tested with different kind of architecture and different kind of key features. The main testing issues of the object oriented software is updating in terms of functionality and also development of new testing techniques.

Class is the logical unit and abstract data type in object oriented software. Class works as encapsulation in which method and data can be encapsulated. The inheritance feature of the object oriented for reusability. Dynamic binding major feature of the object oriented approach binds the relevant method at runtime. Late binding creates indefiniteness in the testing process since the method to be executed is unknown until runtime. As opposed to the waterfall model used for traditional software development, object oriented software is developed using the iterative and incremental approach. Thus, object oriented testing too becomes iterative and incremental, requiring use of regression testing techniques.

Object oriented software is tested at the unit, integration and the system level. At unit level, class is the basic unit of testing. Different approaches like the state-based testing and data-flow testing use the black-box, white-box and gray-box testing techniques to test the class. The already tested classes interact with each other via relationships like inheritance and aggregation to form a subsystem. Integration testing tests for interface errors in the interacting units of the subsystem.

System testing tests the complete application software. Object oriented software testing is aided by testing tools. The testing tools automate the test case design and execution, and the result evaluation at different stages of testing.

The paper is divided in different sections. In *section 2* we discuss in brief the features of the object oriented software. *Section 3* deals with the testing issues unique to the object oriented software. *Section 4* discusses the testing process used to test the object oriented software at the three levels of testing- unit, integration and system level. It includes a survey of the testing strategies used in the testing of the object oriented software. *Section 5* describes briefly the testing tools developed to test the object oriented software.

Unit test tool like JUnit is very important component of software development environment. The Programming environment let us take example JUnit for testing and controlled code changes. If we really look at manually testing . Manually testing equally important as in automated software testing but the main drawback is behavior of the class under unit test because manual test generation is very time consuming and developers often not include some test inputs. Different companies have different tools for test cases so we are not able to get the desired service and even frameworks do not support to the tool. If we really look at the constrained resources, existing test generation tool does not generate sufficient unit tests. As we already discussed we are wasting time on generating and running redundant tests.



Figure 1: Testing web application overview

If we have a constant set of values for method arguments and we are coordinating with non-redundant test, we required at least one new object state. Here we are talking about new state we required in order to generate non-redundant tests.

2. OVERVIEW OF BINARY SEARCH TREE

We are presenting binary search tree here Figure 4.1 shows the relevant parts of the code. The binary search tree class B1 which has a set of integers. Tree has a pointer to the root node.

```

class B1 implements Set {
    Node r;
    static
    class N1 {
        int val;
        N1 le;
        N1 ri;
    }
    public void add(int val) {
        if (root == null) { root = new N1(); root.val =
        val;
        } else {
            N1 n2 =
            root; while
            (1) {
                if (n2.val < val) {
                    if (n2.ri == null) {
                        n2.ri = new N1(); n2.rig.val = val;
                        break;
                    } else { n2 = n2.rig; }
                } else if (n2.val > val) {
                    if (n2.le == null) {
                        n2.le = new N1(); n2.le.val = val;
                        break;
                    } else { n2 = n2.le; }
                }
            }
        }
        public void del(int val) { ... }
        public boolean has(int value) { ... }
    }
}

```

Figure 2: binary search tree Implementation

We are presenting Jtest tool here and generating random sequences of methods. If we really look at Junit then this is a ideal solution of Unit testing.

```

public class B2 extends T {

```

```

public void test1() {
    B2 tx = new B2();
    tx.add(0);
    tx.add(-1);
    tx.remove(0);
}
public void lp() {
    lp txp = new lp();
    txp.add(654321);
    txp.remove(333);
}
}

```

Figure 3: JTest Tool Implementation

We are adopting by existing tools is to explore all methods randomly to a given length. It may be possible that two tests generate same sequence.

3. CONCRETE-STATE EXPLORATION

Unit test for oops programs have two parts: object states and method arguments. The first part can be connected with. The second part introduce particular arguments for a method The concrete-state approach presented in this section assumes a fixed set of method arguments have been provided beforehand and invoke these method arguments to explore and set up object states.

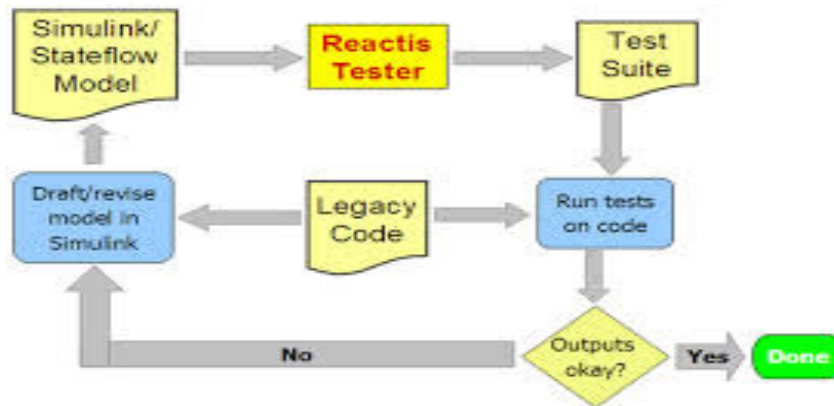


Figure 3: State Implementation

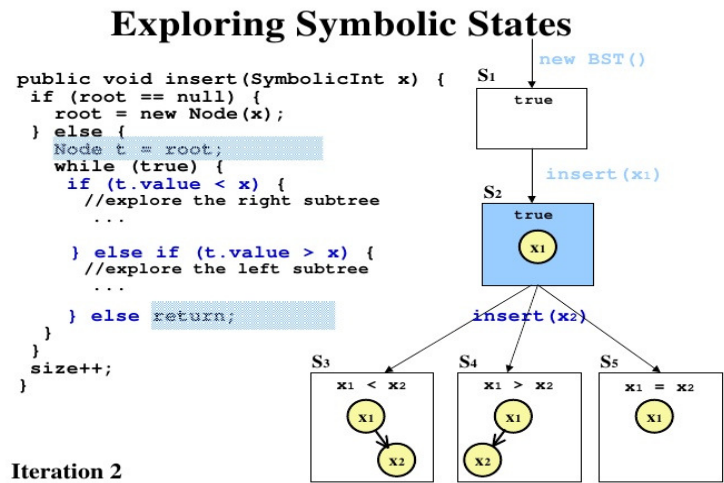
A method-argument state is characterized by a method and the values for the method arguments, where a method is represented uniquely by its defining class, name, and the entire signature. Two method-argument states are equivalent iff their methods are the same and the heaps rooted from their method arguments are equivalent. Each test execution produces several method executions.

This technique is a type of sequential testing. Here we are presenting test generation for each possible input. In order to find generate method state our implementation consists of method arguments. This mechanism is based on huge amount of data. In this thesis we are not including

If we look forward for nonequivalent state then we required test inputs which consist of constructor. If we have empty state then we have to generate new tests for the same. All nonequivalent method-argument states.

4. SYMBOLIC-STATE REPRESENTATION

The symbolic execution consists of inputs in the form of *symbolic* variable instead of formal parameters. In OOPs, state can be generate in terms of symbol. Symbolic states differ from concrete states.



Iteration 2

Figure 2.8: Symbolic Representation

We find a symbolic heap as a graph and each node represent objects and edges represents object fields.

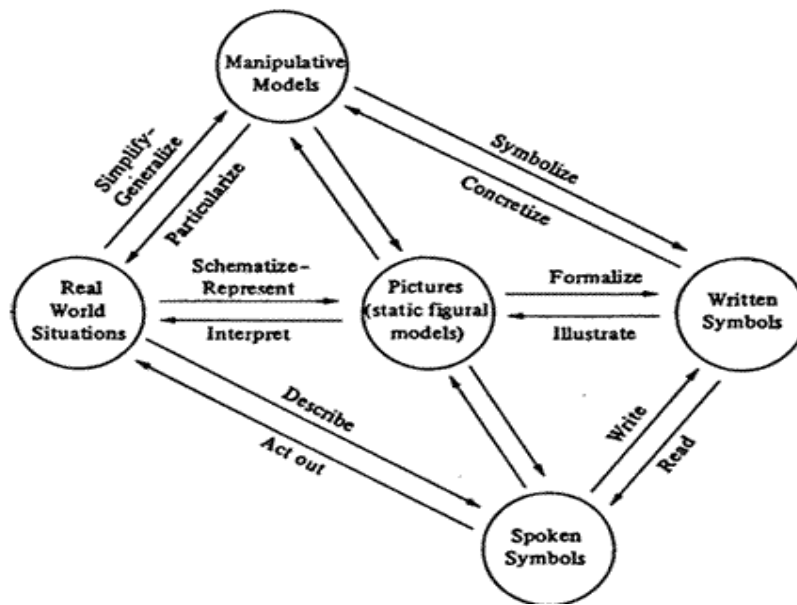


Figure 4: Manipulation Symbolic table

The usual execution of a method starts with a object and method-argument values, and then produces one return value and one concrete state of the receiver object.

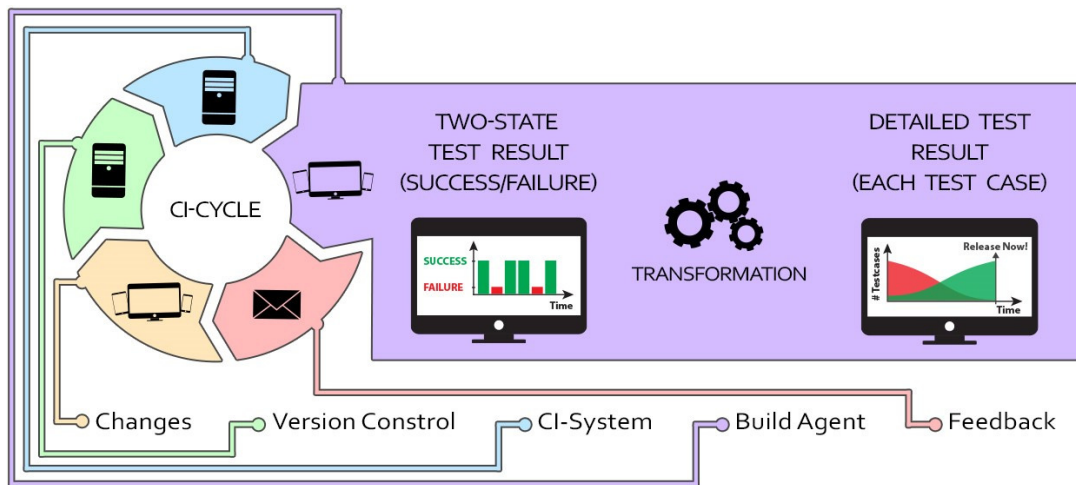


Figure 5: Life Cycle of testing

4.1 Symbolic-State

In this technique I am presenting comparison of two symbolic states. These techniques also support to determine state subsumption

4.1.1 Isomorphism validation

We define heap graph based on node bijection. Here we are looking forward for isomorphic heaps. If we are really interested about it then we can get the method behavior.

This definition supports only object identities and symbolic

4.2 Symbolic-State Representations

In this section we are presenting Symstra approach for symbolic-state space. The state space means all symbolic states that are reachable with the symbolic execution.

Following Code is given below-

```
Set S(Set K, Set N, int Num) {
    Set s = new Set();

    Set f = new Set();
    foreach (constructor in K) {
        RuntimeInfo rrp= symExecAndCollect(constructor);
        s.addAll(rrp.solveAndGenTests());
        fs.addAll(runtimeInfo.getNonSubsumedObjStates());
    }
}
```

Figure 3.1: Algorithm for symbolic sequence

4.3 Observation

Here we are presenting our evaluation for states and generating tests cases. We have performed the experiments on Window XP.

Table shows the lists of 11 Java classes that we use in the experiments. The some classes were previously used in evaluating our redundant-test detection approach.

Table 3.2: data generation

Class	Methods under test	Some private methods	Line of code	No of branches
IntStack	push,pop	-	46	11
UBStack	push,pop	-	63	18
BinSearchTree	add,remove	removeNode	101	38
BinomialHeap	insert,extractMin delete	findMin,merge unionNodes,decrease	323	77
LinkedList	add,remove,removeLast	addBefore	267	18
TreeMap	put,remove	fixAfterIns fixAfterDel,delEntry	383	200
HeapArray	insert,extractMax	heapifyUp,heapifyDown	91	33

We approach following data detection technique for the same.

5. CONCLUSION

Here we proposed Symstra, approach for non-redundant tests that contains high branch and intra-method path coverage for complex data structures. We finally discuss how this technique is useful for this research.

Specifications. The work in this thesis including the Symstra approach has been developed to be used in the absence of specifications. Still we don't have any specification.

Performance: We already discussed about performance parameter which is non-technical attribute for the same.

REFERENCES

- [1] Abbot 0.13.1, 2008. <http://abbot.sourceforge.net/>.
- [2] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In Proc.29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 4–16, 2008.
- [3] Paul E. Ammann, Paul E. Black, and William Majurski. “Using model checking to generate tests from specifications”. In Proc. 2nd IEEE International Conference on Formal Engineering Methods, page 46, 2010.
- [4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. “Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications”. Addison-Wesley Professional,, 2009.
- [5] David Abramson, Ian Foster, John Michalakes, and Rok Socic. Relative debugging: a new methodology for debugging scientific applications. *Communications of the ACM*,39(11):69–77, 2010.
- [6] Ken Arnold, James Gosling, and David Holmes. “The Java Programming Language. Addison-Wesley Longman Publishing Co., Inc., 2010”.
- [7] Agitar Agitator 2.0, November 2004. <http://www.agitar.com/>.
- [8] Dorothy M. Andrews. Using executable assertions for testing and fault tolerance. In Proc. the 9th International Symposium on Fault-Tolerant Computing, pages 102–105,2010.
- [9] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing:A model checker for concurrent software. In Proc. 6th International Conference on Computer Aided Verification, pages 484–487, 2007.
- [10] Alberto Avritzer and Elaine J. Weyuker. Deriving workloads for performance testing. *Softw. Pract. Exper.*, 26(6):613–633, 2003.
- [11] Thomas Ball. A theory of predicate-complete test coverage and generation. Technical Report MSR-TR-2004-28, Microsoft Research, Redmond, WA, April 2004.
- [12] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Proc. 16th International Conference on Computer Aided Verification, pages 515–518, July 2004.
- [13] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In Proc. 26th International Conference on Software Engineering, pages 326–335, 2004.
- [14] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 220–233, 2006.
- [15] Kent Beck. *Extreme programming explained*. Addison-Wesley, 2008.
- [16] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2006.
- [17] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press,2003.
- [18] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 2004.
- [19] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State generation and automated class testing. *Software Testing, Verification and Reliability*,10(3):149–170, 2005.
- [20] Robert V. Binder. Object-oriented software testing. *Commun. ACM*, 37(9):28–29, 2003.
- [21] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In Proc. International Symposium on Software Testing and Analysis, pages 123–133, 2002.

- [22] Thomas Ball and James R. Larus. Efficient path profiling. In Proc. 29th ACM/IEEE International Symposium on Microarchitecture, pages 46–57, 2006.
- [23] Tom Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, 2000.
- [24] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pages 203–213, 2008.
- [25] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In Proc. International Symposium on Software Testing and Analysis, pages 39–48. ACM Press, 2008.
- [26] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2006.

AUTHORS

Manjeet Kumar received MCA. degree in Computer Science from K.N.I.T Sultanpur, (2004), and M-Tech degree in IT(2010) from Karnatika University. At the moment he is Ph.D. scholar at Mewar University, Rajasthan. His research interest includes Automated software testing, Web technology.
Email: manjeet 2005@gmail.com,

