JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit

Guoqing Xu and Zongyuang Yang

Software Engineering Lab, The Department of Computer Science,
East China Normal University
3663 N Zhongshan Rd., Shanghai 200062, P.R.China
{gqxu_02,yzyuan}@cs.ecnu.edu.cn

Abstract. Writing specifications using Java Modeling Language has been accepted for a long time as a practical approach to increasing the correctness and quality of Java programs. However, the current JML testing system (the JML and JUnit framework) can only generate skeletons of test fixture and test case class. Writing codes for generating test cases, especially those with a complicated data structure is still a labor-intensive job in the test for programs annotated with JML specifications.

This paper presents JMLAutoTest, a novel framework for automated testing of Java programs annotated with JML specifications. Firstly, given a method, three test classes (a skeleton of test client class, a JUnit test class and a test case class) can be generated. Secondly, JMLAutoTest can generate all nonisomorphic test cases that satisfy the requirements defined in the test client class. Thirdly, JMLAutoTest can avoid most meaningless cases by running the test in a double-phase way which saves much time of exploring meaningless cases in the test. This method can be adopted in the testing not only for Java programs, but also for programs written with other languages. Finally, JMLAutoTest executes the method and uses JML runtime assertion checker to decide whether its post-condition is violated. That is whether the method works correctly.

1 Introduction

Writing specifications of Java Modeling Language has been viewed as an effective and practical way of increasing the correctness and quality of Java programs for JML' great expressiveness and easy grammar which is similar to Java's. In addition, JML allows assertions to be intermixed with Java code [3, 6], which brings convenience to Java programmer. In the past few years, many tools have been implemented to support JML, including the compiler, runtime assertion checker [1] as well as its testing framework [2]. The current JML testing system (JML and JUnit testing framework) can generate the test fixture and the skeleton of the test case class automatically which sets programmers free from writing unit test codes, thus making unit test more accessible to programmers.

A. Petrenko and A. Ulrich (Eds.): FATES 2003, LNCS 2931, pp. 70–85, 2004.
© Springer-Verlag Berlin Heidelberg 2004

1.1 The Problem

However, programmers still need to spend a lot of time writing codes for generating test cases, especially those which represent complicated data structures (i.e. binary-tree, linkedlist). There have been some testing tools which can automatically generate test cases, such as Korat [4] and Alloy Analyzer (AA)[22], but they either supply the whole test case space generated to the test, never caring about how many test cases are meaningless[1] ones, or require programmers to write special predicates to get rid of meaningless test cases. For example, in Korat[4] programmers should write an additional method "public boolean repOk" in the input class to keep the test cases generated meaningful. In our opinions, at first, only identifying meaningless cases when test is run is not enough because a test with many meaningless inputs can tell little about the execution of tested method although maybe this test is based on a very large test case space and it might spend a lot of time dealing with meaningless ones. So avoiding meaningless test case before running the test is very important. In the second place, in many cases, the tester is not just the one who develops the class to be tested or it is a black box test, therefore, handling test cases totally depending on predicates provided by programmers is not a practical way.

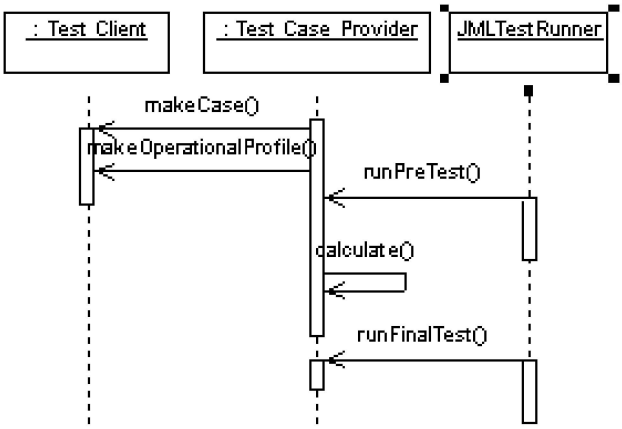

Fig. 1. The work flow of JMLAuto Test

1.2 How JMLAutoTest Deals with These Problems

In this paper, we present JMLAutoTest, an automatic testing framework, which can solve these problems well. Given a method annotated with JML specification, similar to the JML and JUnit testing framework [2], JMLAutoTest firstly generates a JUnit test class (*_JML_Test) which sets up test fixture and a test case provider class

[1] Meaningless test case here means the one which violates the pre-condition of method to be tested.

(*_JML_TestCase) which is a sub class of the test class. In addition, JMLAutoTest generates the skeleton of another class called test client (JMLTestClient), into which programmers can easily set class domains for classes of arguments of method to be tested or field domains for the fields in those classes. Then when test is run, the test case provider can get test cases from test client automatically.

JMLAutoTest can avoid most meaningless test cases by running the test in a double-phase way (runPreTest and runFinalTest in Figure 1). Double-phase way is a statistic based testing approach. Firstly, programmers should make a standard to divide the whole test case space into several partitions. This standard is somewhat like the "Operational Profile" in Cleanroom testing [5, 15], so here we also call it operational profile. Programmers can make the operational profile in the method makeOperationalProfile (shown in figure 1) in the test client. After the test case provider gets the generated test cases, it passes the whole test case spaces to the method makeOperationalProfile. Then this method divides the test case space into several partitions according to the criteria made by programmers and returns these partitions. During the first phase, tests with each group of test cases chosen from these partitions are run respectively. Each group only contains a relatively small number (a few dozen) of test cases.

Then based on the statistical principle we can know which partition of test case space produces the most meaningless test cases and which produces the second most ... Thus, the probability of meaningless test cases contained in each partition can be determined after the first phase test (pre-test). During the second phase, a large number of test cases should be taken out from each partition depending on proportion obtained after the first phase test. From this point of view, meaningless test cases can be avoided to a certain extent and programmer who runs the test only need to make the operational profile without knowing the details about the method to be tested. But the validity of this way is based on the quality of operational profile which is used to create different partitions. This way of testing can be applied to not only Java programs, but also programs with other languages.

The rest of this paper is organized as follows. Section 2 presents the algorithm and principle that JMLAutoTest uses to generate test case. Section 3 describes how double-phase testing works. Section 4 presents the test oracle generation for testing methods. Section 5 illustrates some experimental results. Section 6 reviews related work and Section 7 concludes.

2 Test Case Generation

The whole procedure of test case generation in JMLAutoTest includes three parts: finitization, validity checking and nonisomorphism. Figure 2 gives an overview of the algorithm of JMLAutoTest test case generation for the type "jmlautotest.example". JMLObjectSequence is a utility class defined in the package org.jmlspecs.model [3, 6] which implements a sequence to contain objects. We use class Finitization (Section 2.1) to finish the work of generation. After a test case candidate is generated, JMLAutoTest uses JML runtime assertion checker to check its validity (Section 2.2)

in order to kill it if Invalid. Then JMLAutoTest visits existing test cases to make sure that this candidate is not isomorphic to a certain test case existing in the test case space (Section 2.3).

```
// The following method is defined in the test client
public JMLObectSequence makeCase_jmlautotest_example () {
  JMLObectSequence cases = new JMLObjectSequence () ;
  Finitization f= new Finitization (jmlautotest.example.class);
//Create the value domain1 which contains 5 instances of
//the class "example.field1.class" for the field //"field1"
  JMLObjectSequence valuedomain1
   = f.createObjects(example.field1.class, 10);
// set value domain for the field "field1"
  f.set (example.field1, valuedomain1);
  f.generate() ;
  cases = f.getResult() ;
  return cases;
  }
  }
```

Fig. 2. The overview of the method makeCase_* in the test client

2.1 Finitization

JMLAutoTest provides a class Finitization for programmers to generate a finite test case space of any kinds. The whole process of the working of a finitization includes two parts: setting the value domain and generating.

Set Value Domain for Fields of the Input Class. Programmers can bind a certain field with a set of bounds by setting the value domain for the field. Then JMLAutoTest will create a candidate object by assigning to each field all possible values from its corresponding domain. The field domain is represented by an object either of the type JMLObjectSequence which contains objects or of the type JMLPrimSequence which is another utility class to contain values of primitive types. All of these domains are inserted into a hash table for use in generating.

Generate Test Case Space. There are two kinds of method generate in class Finitization. One is to generate test case space for common classes and another is to generate test cases for special classes which implement a certain linked data structure such as binarytree and linkedlist.

Figure 3 illustrates how to generate test case for the class which implements a linked data structure (linked list). The invariant clause at the beginning will be presented in Section 2.2. The method *generate* here is with two arguments. The first one is the name of the first node (or the root node) field in this recursive structure. The second one is an array of String which contains names of pointer fields in this data structure. In the example of linkedlist shown in Figure 3, the first argument of method *generate* is the string "first", the name of field *first* in class LinkedList which represents the first node of a linked list. The second one is a string array which only contains "pointer" , the name of the pointer field in LinkedList.

In the process of this kind of generation, the field *first* and the pointer field *pointer* share the same value domain. Each element of this value domain can only be used once except null. After one element is used, it will be removed from the domain. A special stack *visitedStack* is kept during the recursion to contain used elements of the value domain. Another stack called *fNames* is used to contain names of pointer fields. In the next recursion, one element in the value domain was assigned to the pointer field represented by the first element in *fNames* of the object represented by the first element in *visitedStack.*

For example, when JMLAutoTest is generating a binary tree, the situation of two stacks and value domain are illustrated in Figure 4. At the beginning, stack *visited-Stack* only contains binarytreeObj which is an object of input class BinaryTree. The value domain contains four nodes: N_0, N_1, N_2 and null. During the first recursion, N_0 is assigned to the field *root* which is represented by the first element in *fNames* of the Object binarytreeObj represented by the first element in *visitedStack*. That means let binarytreeObj.root = N_0. Then the first element in both *fNames* and *visitedStack* is removed. Also, the used element N_0 is inserted into *visitedStack* and its two pointer fields *left* and *right* are inserted into *fNames*. Recursion follows this algorithm until it reaches two states. One is that the value domain only contains null and another one is that all elements in *visitedStack* are null. The first state means finding a candidate while the second one means a failure.

To accelerate the generation of a linked data structure, programmers can choose to generate the test case space in a fast way. If the value of field acceleratingEnabled in the class Finitization is true, JMLAutoTest only considers a certain structure once regardless of other non-pointer fields. JMLAutoTest implements this optimization by assigning values to the pointer field only twice. For the first time, a non-null value in the value domain is assigned to the pointer field, and for the second time let the pointer field be null. Generating all cases of a binary tree with 7 nodes costs 1 second while in the normal way, it costs more than 1000 seconds because candidates handled by JMLAutoTest in the normal way are much more. However, the test case space generated in the fast way is so limited that it only contains all kinds of the structure without caring about non-pointer fields.

2.2 Validity Checking

After a candidate object is generated, JMLAutoTest checks its validity to judge whether it can be used as a test case. The invariant clause in Figure 3 says that if the field *first* is null, then the field *length* must be zero or if *first* is not null, then the field *length* must equal the real number of nodes in this list (Method *getLength* returns the value of field *length* and method *toObjectSet* functions at transforming the linked list to a set of nodes. Both of them are omitted in Figure 3). The validity checking in JMLAutoTest totally depends on the instance invariant specified in the input class.

The Invariant Clause in JML. An invariant [1] is a condition that remains true during the execution of a segment of code. A instance invariant, which constraints both static and non-static states of program execution, can refer to both static and

instance members. In JML, invariants belong to both pre-state and post-state specifications which are checked in both pre-state, i.e., right after a method call and argument passing but just before the evaluation of the method body and post-state, i.e., right after the evaluation of the method body but just before the method returns.

```
//@ public invariant (first == null&& getLength()==0)
//@       || (first!= null &&getLength() == toObjectSet().size());
public class LinkedList{
  public Node first; // the first node of a linked list;
  protected int length;  // the length of this list
  …
  }
  public class Node{
  public int ID;  // node ID
  public Node pointer; // a pointer pointing to the next node
}

public JMLObjectSequence  makeCase_LinkedList(){
     Finization f = new Finization (LinkedList.class) ;
//create 3 instances of Node with an argument array [0,1,2] for the
//constructor
    JMLObjectSequence  nodes  =  f.  createObjects(Node.class,  new
          JMLPrimSequence(new int[]{0,1,2}) , 3);
    f.add (nodes, null); // add null to this domain.
    f.set ("first", nodes); // set the value domain for "first"
// set domain for the field "length"
    f.set ("length", new JMLPrimSequence (1,4));
  }
// Generate candidates  recursively
   f.generate("first", new String[]{"pointer"});
   return f.getResult();
}
```

Fig. 3. Generate the test case space for class LinkedList

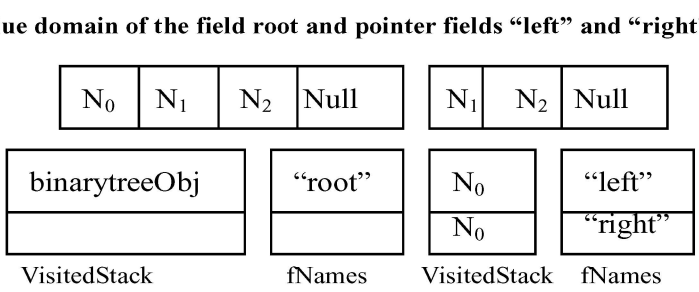

Fig. 4. The difference of situations of value domain and stacks between at the beginning and after a recursion when generating a binarytree

The Invariant Clause in JML. An invariant [1] is a condition that remains true during the execution of a segment of code. A instance invariant, which constraints both static and non-static states of program execution, can refer to both static and instance members. In JML, invariants belong to both pre-state and post-state specifications which are checked in both pre-state, i.e., right after a method call and

argument passing but just before the evaluation of the method body and post-state, i.e., right after the evaluation of the method body but just before the method returns.

```
public void checkInv$instance$T(){
  Boolean rac$v = false;
  [P, rac$v ]
  if (! rac$v){throw new JMLInvariantError() ;}
}
```

Fig. 5. The method to check the instance invarianttranslated by JML runtime assertion checker

```
public boolean checkValidity(Object obj){
 try{
   //get the name of the input class
    String className = obj.getClass().getName();
   //get the name of the method of checking invariant
    String invName = "checkInv$instance$"+className;
    Method thisMethod = obj.getClass().getMethod(invName, new
    Class[]{}); // get the method of checking invariant
   //invoke this method
    thisMethod.invoke(obj, new Object[]{});
    }
 catch(java.lang.NoSuchMethodException ex){
    throw      new      Exception("code      for      class
    "+obj.getClass().getName() +" was not compiled with jmlc so
    no assertions will be checked");
    } // There is no such a method in the class
 catch(JMLInvariantError ex){
    return false;   // Invariant has been violated.
    }
 return true; // Okay
}
```

Fig. 6. Check whether the candidate is valid

Let T be a type with a set of instance invariants, P1…Pn .The invariants are first conjoined to form a single invariant predicate, i.e., P ≡ P1^ ...^ Pn. The conjoined invariant predicates are translated into instance invariant methods, whose general structures are shown in Figure 5. The notation [P, rac$v] denotes translated code that evaluates the predicate P and stores the result into the variable rac$v. The invariant methods evaluate the conjoined invariant predicates and throw invariant violation errors if the predicates do not hold.

Invariant Checking in JMLAutoTest. Figure 6 illustrates how invariant is checked in JMLAutoTest. If the method *checkInv*$*instance*$*T* is not found in the input class, this class was not compiled by JML compiler that a new exception was thrown.

If a JMLInvariantError exception is caught when the method is invoking, this candidate is not valid because the invariant is violated. If no exceptions are caught, this candidate is valid and the method *checkValidity* returns true. If no invariants specified in the input class, the method *checkValidity* always returns true.

2.3 Nonisomorphism

At the end of the process of generation, JMLAutoTest explores the space of generated test cases to make sure the candidate is not isomorphic to a certain existing test case. We do not define what isomorphism is in JMLAutoTest. Our solution is totally based on the method *equals* defined in the input class.

Let Obj be the candidate object and let S be the set of test cases generated. Obj is isomorphic to an existing test case iff $\exists c \in S$. Obj.equals(c). There is an advantage of this solution that programmers can easily change the criteria to make different kinds of test cases by modifying the *equals* method.

3 Double-Phase Testing

This section presents how JMLAutoTest avoids meaningless test cases. After the generation of the test case space presented by the previous section, there might be many meaningless ones in the space whereas individual candidate itself is valid. The major idea of double-phase testing is to use two phases of testing based on statistics.

Double-phase testing is especially fit for the black box test and the test with a large test case space although it will spend some time running pre-test. If the test case space is not very large (maybe only contains a few dozen of cases), we do not need to use the double-phase testing. Programmers can decide whether to use this method by choosing whether to run the test with an argument "-pre". Running the test case provider (class *_JML_TestCase) without any arguments means the test will be run in a conventional way.

3.1 Making an Operational Profile

There is a method makeOperationalProfile in the test client. Programmers can modify this method to divide the whole test case space into several partitions. JMLAutoTest can generate the skeleton codes of this method. Test case spaces of different types are put into a hash table spaces in the test case provider. Then this hash table is passed to the method makeOperationalProfile in test client. For example, we can put linked lists which contain more than three nodes into partition_LinkedList[0] and put those which contain less than three nodes into partition_LinkedList[1]. Variable *partition_LinkedList* is a JMLObjectSequence array, each element of which represents a partition of this test case space. Finally partitions of each type are put into a hash table *partitions* which will be returned by this method.

3.2 The First Phase

During the first phase, a small number of test cases taken from each partition randomly make several groups. The percentage of test cases which should be taken out from each partition is provided by the programmer as the second argument of the main method (i.e., if arguments are "-pre 0.3", the percentage is 30%.). Tests with each group are run respectively. Figure 7 illustrates the generated codes for the first-phase test. If the pre-test is not disabled by programmer, let *isPre* (a flag to show whether it is a pre-test or a final test) be true.

Taking Test Cases Out from Partitions. The technique of getting these test cases is based on the following principle.

Let p be the percentage provided by the programmer. Let P be a partition of a certain test case space, and Let C be a sub set of P. C is the set of cases which should be taken out iff C.size == $\lceil$ P.size*p $\rceil$ and $\forall$ n $\in$ [0, C.size-1], C.elementAt(n)== P.elementAt($\lfloor$ n/p $\rfloor$). The operator == is Java's comparison by object identity. The definition above illustrates that the process of getting test cases is totally based on the average distribution in statistic.

Algorithm Used in Running the Pre-test. In section 3.1, we have presented that each partition of test case space of a certain type is represented by an element in a JMLObjectSequence array. We use an algorithm to record the number of meaningless test cases during the first phase that we shift the sequence of elements in a partition backward and let the first unit contain the number of meaningless cases. At first, we put zero into its first unit.

When the test suite is run, the sum of number of meaningless test cases taken from a certain partition in the current test and the value in its first unit is put into this unit of the partition. So during the first phase, the value in the first unit of a partition changes for several times. Through comparing the final value in the first unit of a partition with one another in the same test case space, we can get the approximate proportion of meaningless cases in this partition among all those meaningless in the whole test case space.

Method *runPreTest* shown in Figure 7 is a recursive method which explores every partition in test case space of each type. We keep a hash table *arg_ind* which represents a vector of indices of partitions in test case space of each type. We continue with the example method *findSubList*. There are two test case spaces generated for the type *LinkedList* and *Node*. If test case space of *LinkedList* has been divided into two partitions and the space of *Node* has been divided into three partitions, in the first phase, test suite should be run for six times. Each time before test suite is run; the value of *arg_ind* should be changed to show the indices of partitions in these two test case spaces. At the beginning, *arg_ind* is the vector (0, 0) which means test cases of both type *LinkedList* and *Node* should be taken out from No.0 partition in the two test case spaces. During the first recursion, *arg_ind* should be changed to (0, 1) and the

recursion ends when *arg_ind* reaches (1, 2) which means all partitions have been visited. When test suite is run, the init_vT methods presented in the next section get the test cases from different partitions according to the vector represented by *arg_ind*.

```
//The following codes appear in the test case provider
//(*_JML_TestCase.java) for testing the method "findSubList".

//an object of test client
  JMLTestClient  myClient = new JMLTestClient();
//Initialize hashtable param in order to receive test cases
  Hashtable param = new Hashtable();
// Get test cases from JMLTestClient
  param.put("LinkedList",
             myClient.makeTestCases_LinkedList());
// Divide the whole test case space into several partitions
  Hashtable args_findSubList
            = myClient.makeOperationalProfile(param);
// If the pre-test has not been disabled by programmer.
  if (args.length==2 && args[0]=="-pre"){
// A flag to show whether it's a pre-test or a final-test
  boolean isPre = false;
//Percentage of test cases which should be taken from each
//partition
  double percentage = Double.parseDouble(args[1]);
// run test suite for several times
  runPreTest( percentage );
```

Fig. 7. Generated codes in test case provider (*_JML_TestCase) for the first phase test

3.3 The Second Phase

During the second phase, test cases from partitions should be reorganized and the final test is run.

Reorganization of Test Cases. After the first phase test, the first unit of each partition has contained the number of all meaningless test cases taken from this partition. Although this number might be more than the real one, it can reflect the real situation of each partition in a certain test case space. After the disposal of these numbers, we can get the proportion of test cases which should be taken out from each partition of a certain test case space in the final test. Let S be the set of such proportions. Let C be the set of values contained in the first unit of each partition in a test case space. The following algorithm is used in JMLAutoTest to get the test cases in the final test.

Algorithm: s = sum(C);
forall n∈ [0, C.size-1], S.elementAt(n)= (s – C.elementAt(n))/s;

Finally, in a certain test case space, let T_1 be the set of test cases taken from partition P_1, T_2 be the set of cases taken from the partition P_2… Tn be the set taken from partition Pn. The operation of taking test cases from Pi is also based on the average distribution with the proportion S.elementAt(i-1). Then we have the equation

$\forall$ i$\in$ [1 , n], Ti.size == $\lfloor$Pi.size * S.elementAt(i-1)$\rfloor$. Let T be T_0 U T_1 …U Ti. Then T is the set of test cases to be supplied in the final test.

```
/** The following method is defined in the test class and
    is overridden in test case provider class. */
public void init_vLinkedList(){
if (isPre){ // if it is in the first phase
/** Get the index of partition from which test cases should
     be taken */
   int           ind=    ((Integer
                 arg_Ind.get("LinkedList")).intValue();
/** Initialize vLinkedList with cases from the partition
     whose index is represented by ind */
   ...
}
else { // It is a final test
/**  Initialize vLinkeList with cases from all partitions
      according to the obtained proportions. */
 }
}
```

Fig. 8. The method init_vT in test case provider class

4 Test Oracle Generation

JMLAutoTest uses the same way of generating test oracles as JML+JUnit testing framework [2] which combines JML [3] and JUnit [7] to test individual method.

4.1 Setting up Test Fixture

The test fixture for the class C is defined as:

C[ ] receivers; T_1[ ] vT_1; ... ; T_n[ ] vT_n;

The first array named receivers is for the set of receiver objects (i.e., objects to be tested) and the rest are for argument objects.

The receiver and argument objects are initialized by the method init_receivers and init_vTi in the test case provider class. Figure 8 describes generated codes in method init_vLinkedList for initializing vLinkedList. If it is in the first phase, test cases are taken from the partition, the index of which is represented by the value in the hash table arg_ind and test cases should be taken out from all partitions and mixed together in the second phase.

4.2 Testing a Method

For each instance (i.e., non-static) method of the form:

T M(A1 a1,: : :, An an) throws E1,..., Em { /* ... */ }

of the class C, a corresponding test method testM is generated in the test class C_JML_Test. Let n be the value of vT1.length * vT2*…* vTn.length. Then, the

method to be tested is executed for n times until each element in each array (vTi) has been visited. Pre-condition of the target method is checked firstly. If the pre-condition has been violated and the current test is the pre-test (isPre==true), let the variable meaningless shown in figure 8 increase. If the post-condition of the method is violated, JMLAutoTest handles it in different ways in two phases. During the first phase, this exception is ignored and the test continues since we just want to know the number of meaningless test cases not caring about whether the execution fails or succeeds. But during the second phase, this exception should be thrown to let programmer know execution of the method is not correct.

5 Experimental Results

This section presents performance of JMLAutoTest on testing a method. To monitor the process of test case generation and testing a method, JMLAutoTest uses a class JMLTestDataStat to record some key data. We use method findSubTree(BinaryTree parentTree, Node thisNode) whose function is to find a sub tree whose root is represented by *thisNode* in the parentTree as the benchmark for which we show JMLAutoTest's performance.

5.1 Generating Test Cases and Dividing Test Case Spaces

Figure 9 describes the definition of the BinaryTree and Node. The invariant clause tells us if root is null, the size must be 0. If root is not null, the size must equal the number of total nodes in the tree. What Figure 10 illustrates is the JML specifications for the method public *BinaryTree findSubTree*(*BinaryTree* parentTree, *Node* thisNode). The pre-condition of the method requires that neither of its arguments can be null and there must be a node in parentTree whose ID equals the ID of thisNode.

We generate the test case space of type BinaryTree with a few nodes whose ID are ranging from 5 to 8. We also generate the case space of type Node which contains 12 nodes whose IDs are from 0 to 11.

For the test case space of type BinaryTree, We do not divide it and leave it as the only partition. For the space of type Node, We divide it into two partitions. The first one contains nodes whose ID varies from 0 to 5 and the second one contains the rest.

5.2 Test Results

Table 1 shows JMLAutoTest's performance when we test the method with binary trees containing nodes from 5 to 8. We generate the test case space of BinaryTree in the fast way, so the number of candidates considered is close to that of test cases generated. We use the arguments "-pre 0.25" to run the test. Note that for all kinds of binary trees listed in table 1, almost all test cases in the final test are meaningful.

```
public class BinaryTree{
//@ public invariant (root ==null &&
//@     getSize() ==0)||(root!=null&&getSize()!=0
//@      &&toObjectSet(root).size() == getSize());
public Node root;
  protected int size;
  public int getSize(){… }
  public JMLObjectSequence toObjectSet(){…}
… }
public class Node{
   public Node left;
   public Node right;
   public int ID;  }
```

Fig. 9. The Definition of class BinaryTree and Node

```
/*+@ public normal_behavior
     @ requires parentTree!=null && thisNode!=null &&
      @(\exists Node n; @parentTree.toObjectSet().has(n);
      @n.ID== thisNode.ID);
      @ assignable \nothing;
      @ ensures \result.root.ID == thisNode.ID ;
     @+*/
```

Fig. 10. The pre-condition of method findSubTree

Then we make a comparison between the performance of testing in double-phase way and the conventional way (Table 2). Note that for all binary trees with more than five nodes, total time of the test in double-phase way is less than the corresponding time in the conventional way and the more test cases are, the more time double-phase testing can save. Although some meaningful test cases have also been filtered out in double-phase testing, the test case space is still large enough to test the correctness of the method.

6 Related Work

There are now quite a few testing facilities and approaches based on formal specifications developed and advocated by many different research groups. One of the earliest papers by Goodenough and Gerhart [8] presents its importance. Approaches like automated test case generation from UML statecharts [9,21] and Z specifications [10,20] present ways of generating test cases from formal specifications. There are also some tools which can generate Java test cases like the TestEra framework [11,22] which requires programmers to learn a specification language based on which, test cases can be generated. All these specifications do not generate test cases for Java programs annotated with JML specification which is widely accepted as the ancillary tool tailored for Java to keep the correctness of programs.

Several researchers noticed that if a program is formally specified, it should be possible to use the specification as a test oracle [12, 13, 14]. Cheon and Leavens [2] present the JMLUnit framework which can generate test oracles for JUnit [17] from

JML specifications. This framework uses JML runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. However it has not automated the generation of test cases which is still a labor-intensive for programmers.

Table 1. Performance of JMLAutoTest for testing the method findSubTree with arguments "-pre 0.25" (test cases is generated in the fast way).

nodes in binary tree	generated binary trees	candidates considered	meaningful cases in the final test	total test cases in the final test
5	42	64	410	492
6	132	196	1572	1572
7	429	625	5136	5136
8	1430	2055	17148	17148

Table 2. Performance comparison between the double-phase testing in JMLAutoTest and the conventional way in JMLUnit testing framework.

	Double-phase way			Conventional way		
nodes in binary tree	**meaningful /total in final test**	**time in the first phase (s)**	**total time of the test (s)**	**meaningful /total in final test**	**time in the first phase (s)[2]**	**total time of the test (s)**
5	410/492	0.079	0.266	420/1008	0	0.219
6	1572/1572	0.188	0.422	1584/3168	0	0.468
7	5136/5136	0.36	0.766	6006/10296	0	1.25
8	17148/17148	0.703	2.016	22880/34320	0	3.484

Boyapati, Khurshid and Marinov describe Korat [4] which can finish automated generation of test cases for Java programs with formal specifications. Korat generates linked data structures based on additional Java predicates. However Korat requires that the programmer who runs the test must know well about the details of the program to be tested, therefore it is not fit for a black box test. Also, it can not keep meaningless test cases from being handled.

There are quite a few approaches to applying the statistical models to Testing [16,18,19]. Statistical testing has been widely adopted during the development of the Cleanroom software[5] in test cases acquisition, results evaluation and reliability modeling. So it is not a new idea to use the statistical analysis in testing. But in JMLAutoTest the novelty lies in applying the statistical analysis to filtering out meaningless cases. This idea can also be used in testing of programs written with other languages.

[2] All the time in this column is zero because there is no the first phase test in conventional testing.

7 Conclusions

This paper presents JMLAutoTest, a novel testing framework designed for Java programs annotated with JML specifications.

JMLAutoTest automatically generate three classes for a target method. In the test client, testers can generate test cases for any kinds of types including linked data structures and common types in either a fast way or a normal way very easily. JMLAutoTest verifies the validity of a candidate by checking its invariant with JML runtime assertion checker.

JMLAutoTest provides a double-phase testing way for the test of a method. It is the statistic based testing which filters out meaningless test cases without requiring testers to know the details of the method to be tested. According to the operational profile made by the tester, the generated test case space can be divided into several partitions. During the first phase, a small number of test cases are taken out from each partition. Then the test suite is run for several times to record the number of meaningless cases of each group. Based on statistical principles, we should estimate the approximate proportion of the meaningless test cases in each partition. During the second phase, test cases taken out from each partition according to these calculated proportions are mixed together and supplied to the test. Time spent visiting meaningless test cases in the final test is saved.

References

1. Y.Cheon. A Runtime Assertion Checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Apr. 2003.
2. Y.Cheon and G.T.Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01-12, Department of Computer Science, Iowa State University, Nov, 2001.
3. G.T.Leavens, A.L.Baker, and C.Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).
4. C. Boyapati, S. Khurshid and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123-133., July 2002.
5. M.Deck and J.A.Whittaker. Lessons learned from fifteen years of Cleanroom Testing. *Software Testing, Analysis, and Review (STAR) 97*, May 5-9, 1997.
6. G.T.Leavens, A.L. Baker and C. Ruby. JML: A notation for detailed design. In *Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
7. K. Bech and E. Gamma. Tested infected: Programmers love writing tests. Java Report, 3(7), July 1998.
8. J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.
9. J.Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, Oct. 1999.

10. H. M. Horcker. Improving software tests using Z specifications. *In Proc. 9th International Conference of Z users, The Z Formal Specification Notation*, 1995.
11. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc, 16th IEEE Engineering (ASE)*, San Diego, CA, Nov, 2001.
12. D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proc, ISSTA 94*, Seattle, Washington, Aug. 1994.
13. D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc, ISSTA 94*, Seattle, Washington, August, 1994.
14. P. Stocks and D. Carrington. Test template framework: A specification-based test case study. In *Proc, the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 11–18, Jun. 1993.
15. J. D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14-32, Mar.1993.
16. NIST/SEMATECH E-Handbook of Statistical Methods. http://www.itl.nist.gov/div898/handbook/, May, 2003.
17. JUnit. http://www.junit.org.
18. R. Chillarege. Software testing best practice. Technical Report RC 21457, Center for Software Engineering, IBM Research, Apr.1999.
19. D. Banks, W. Dashiell, L. Gallagher, C. Hagwood, R. Kacker and L. Rosenthal. Software testing based on statistical methods. National Institute of Standards and Technology Information Technology Laboratory, Gaithersburg, MD, Mar.1998.
20. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
21. J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series, 1998.
22. D. Jackson, I. Shlyakhter, and Il Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE),* Limerick, Ireland, June 2000.