The Report Committee for Hua Zhong

Certifies that this is the approved version of the following report:

Pairwise-Korat: Automated Testing Using Korat in an Industrial Setting

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Sarfraz Khurshid

_____

Lingming Zhang

Pairwise-Korat: Automated Testing Using Korat in an Industrial Setting

by

Hua Zhong, B.E.; M.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2015

Dedication

To my parents, Haiyan Li and Minghe Zhong, my wife, Lei Zhou, and my grandmother, Xiaocai Wang.

Acknowledgements

It is a great honor to have Dr. Sarfraz Khurshid as my graduate advisor. The inspiration of this work comes from the lectures which Dr. Khurshid gave in his Software Verification and Validation class, back to 2013. After completing the class, I started to apply tools I learned from the class to the projects in my work. Whenever I need consultations and suggestions from Dr. Khurshid, he always made time for me and gave me excellent technical advice. Dr. Khurshid inspired me to start this project and guided me throughout the entire process. The work could not be finished without his help. The reader of my report is Dr. Lingming Zhang. Lingming and I worked briefly together at eBay in 2013. He tirelessly provided me guidance on the project during that time. Discussions with Lingming greatly improved the quality of the work.

My former manager at eBay, Russell Lager, gave me great support during my work. He provided lots of helpful technical comments and helped me to balance my work and study.

Besides, I also want to thank my family for their love, understanding and support which have led me through. Without support of my wife, Lei, and my parents, I would not be able to complete the degree.

Abstract

Pairwise-Korat: Automated Testing Using Korat in an Industrial Setting

Hua Zhong, MSE

The University of Texas at Austin, 2015

Supervisor: Sarfraz Khurshid

In this report, we present an algorithm for testing applications which takes structurally complex test inputs. The algorithm, Pairwise-Korat, adopts Korat — an algorithm for constraint-based generation of structurally complex test inputs. Korat takes (1) an imperative predicate which specifies the desired structural integrity constraints and (2) a finitization which bounds the desired test inputs size. Korat performs a systematic search to generate all test inputs (within the bounds) for which the predicate returns true. We present how to generate test inputs in Korat and how to execute test inputs in parallel. The inputs that Korat generates enable bounded-exhaustive testing that checks the code under test exhaustively for all inputs within the given bounds. We also describe a novel methodology for reducing the number of equivalent inputs that Korat generates. Our development of test input generation and the methodology for reducing equivalent inputs are motivated by testing applications developed at eBay. The experimental results show that the Pairwise-Korat achieves great performance in finding defects and increasing test coverage and the algorithm outperforms current manual solutions adopted at the company.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

1.1 Background and Proposed Solution

Software testing plays an important role in software development lifecycle and it is also the dominant method for finding software defects before releasing the software to market [1]. Software testing usually consists of two main parts: (1) test generation, which creates tests to be executed; and (2) test execution, which executes the tests to check the code under test. When dealing with industrial projects, execution is often automated to handle a large number of test requirements. However, test generation is typically manual and thus laborious and often produces inputs that exercise only a small subset of the functionality of the software. The quality of the test cases and coverage is solely based on test case designer's own experience.

1.1.1 Test Generation is Burdensome

So why do test generation need to be performed manually instead of automated? Test generation would be straightforward if desired inputs were simple, e.g., if the input domain is an integer value in the range of (0-100). However, for most programs, inputs are in complex structures. For example, let's consider a web service program which lists an item on a web application. For correct behavior that program might require its input to contain information like username, item description, item price, refund options and

shipping options. The program may require user to provide the information in an XML file. A sample of such an input file is shown in Figure 1.1.

```
▼<testInput xmlns:ns2="urn:ebay:apis:eBLBaseComponents" rootId="0">
  ▼<item>
     <ns2:Country>AU</ns2:Country>
     <ns2:Currency>AUD</ns2:Currency>
     <ns2:Description>Alloy Generated Item: AUSTRALIA_FIXED_PRICE_ITEM</ns2:Description>
     <ns2:ListingDuration>Days_7</ns2:ListingDuration>
     <ns2:ListingType>FixedPriceItem</ns2:ListingType>
     <ns2:Location>Canberra</ns2:Location>
     <ns2:PaymentMethods>PayPal</ns2:PaymentMethods>
     <ns2:PayPalEmailAddress>xxx@xxx.xx.ebay.com</ns2:PayPalEmailAddress>
    ▼<ns2:PrimaryCategory>
       <ns2:CategoryID>12</ns2:CategoryID>
     </ns2:PrimaryCategory>
     <ns2:Quantity>1</ns2:Quantity>
    ▼<ns2:ShippingDetails>
      ▼<ns2:ShippingServiceOptions>
         <ns2:ShippingService>AU_Regular</ns2:ShippingService>
         <ns2:ShippingServiceCost>3.0</ns2:ShippingServiceCost>
         <ns2:ExpeditedService>false</ns2:ExpeditedService>
         <ns2:ShippingTimeMax>3</ns2:ShippingTimeMax>
         <ns2:FreeShipping>false</ns2:FreeShipping>
       </ns2:ShippingServiceOptions>
      ▼<ns2:ShippingServiceOptions>
         <ns2:ShippingService>AU_Express</ns2:ShippingService>
         <ns2:ShippingServiceCost>0.0</ns2:ShippingServiceCost>
         <ns2:ExpeditedService>true</ns2:ExpeditedService>
         <ns2:ShippingTimeMax>3</ns2:ShippingTimeMax>
         <ns2:FreeShipping>true</ns2:FreeShipping>
       </ns2:ShippingServiceOptions>
       <ns2:ShippingType>Flat</ns2:ShippingType>
     </ns2:ShippingDetails>
     <ns2:Site>Australia</ns2:Site>
     <ns2:StartPrice currencyID="AUD">2.0</ns2:StartPrice>
    ▼<ns2:Title>
       Alloy Item: AUSTRALIA_FIXED_PRICE_ITEM 1381873143372
     </ns2:Title>
     <ns2:PostalCode>95125</ns2:PostalCode>
     <ns2:DispatchTimeMax>0</ns2:DispatchTimeMax>
    ▼<ns2:ReturnPolicy>
       <ns2:RefundOption>MoneyBack</ns2:RefundOption>
       <ns2:ReturnsWithinOption>Days_14</ns2:ReturnsWithinOption>
       <ns2:ReturnsAcceptedOption>ReturnsAccepted</ns2:ReturnsAcceptedOption>
     </ns2:ReturnPolicy>
     <ns2:ConditionID>1000</ns2:ConditionID>
   </item>
   <id>0</id>
 </testInput>
```

Figure 1.1 An XML file to post an item on an online auction site

1.1.2 Generating Tests from Constraints

The key idea in this work is to generate tests from logical constraints. Comparing to manual input generation, it is often much simpler to describe the properties of desired

input data. A key advantage of using input constraints is that the constraints typically cover an entire input domain rather than a small subset of that class. Therefore, a constraint solver can be implemented to generate valid inputs for an entire class rather than a set of concrete inputs. The use of constraints allows test designers to generate a test suite with no bias and covers the entire input domain with a given bound on the input size.

Before we can solve the input constraints, we need to understand the nature of them. There are a number of studies [2,3,4,5] for generating tests from constraints have considered constraints on primitive data, such as integers and booleans. However, in most industrial applications, data with complex structure are pervasive. Such of data are defined by their structural constraints, e.g., in a binary tree, each node has a unique parent and no node has the same node as both left and right child.

1.1.3 Korat

The Korat [6] tool presents an embodiment of how we address these challenges for automated testing of our programs.

Korat is a Java algorithm for constraint-based generation of structurally complex test inputs. Korat performs specification-based testing: given a Java predicate that describes properties of desired input data, Korat performs a backtracking search to explore the input space of the predicate and enumerate all inputs for which the predicate returns true. Korat returns each enumerated input as a desired test input. To test a program, Korat requires the program precondition to generate tests and the postcondition

to verify correctness of the program. Korat enables bounded exhaustive testing: it tests against all non-isomorphic inputs within a given bound on the input size. Bounded exhaustive testing has been proved to be an effective methodology to find bugs in various applications, including a fault-tree analyzer [7], a resource discovery architecture [8], and an XPath compiler [9].

While bounded exhaustive testing is very effective in some software, it is not the case in many industrial applications. The reason is very simple and straightforward: the size of the input space to test an industrial application is usually too complicated and such an exhaustive generation will produce an enormous large number of test inputs. It is infeasible to run such a large number of test cases in one test execution. Besides, an industrial test requirement often requires reasonable cost-benefit compromise between test code coverage and the time/resources expenses. Due to this limitation, many of the generated tests will be categorized as “corner” cases or “negative” test cases in a test plan design and thus should be removed from the plan due to their low priorities.

1.1.4 Pairwise Testing

To solve the above issues, this report presents Pairwise-Korat, a pairwise test generation framework based on Korat. Pairwise testing is a combinatorial method of software testing that, for each pair of input parameters to a system, tests all possible discrete combinations of those parameters.

The reasoning behind pairwise testing is as the followings. The simplest bugs in a program are generally triggered by a single input parameter. The next simplest category

of bugs consists of those dependent on interactions between pairs of parameters, which can be caught with pairwise testing [10]. Bugs involving interactions between three or more parameters are progressively less common [11], while at the same time being progressively more expensive to find by exhaustive testing, which has as its limits the exhaustive testing of all possible inputs [12].

One of the main strengths of combinatorial technique is that it enables a significant reduction of the number of test cases without compromising functional coverage. Many testing methods regard all-pairs testing of a system or subsystem as a reasonable cost-benefit compromise between often computationally infeasible higher-order combinatorial testing methods, and less exhaustive methods which fail to exercise all possible pairs of parameters. For example, consider the case of N=10 binary parameters. An exhaustive set of tests involves 2^{10} tests, whereas the all-pair setting would involve just 6.

1.1.5 Complete Proposed Solution

The key insight in this report is that even though it is not feasible for Korat to explore an entire input space, we can still apply Korat to search for a subdomain of the space and then systematically select pairwise test cases from the generated candidates. The proposed framework first adopts Korat to search for a set of candidate inputs based on a series of filters defined in Korat's Java predicate. These filters are designed in such a way that only a selective number of negative test cases will be included in the candidate domain. The candidate inputs will then be placed into a pairwise filter and a set of

pairwise test cases will be selected from the candidates. Those pairwise test cases serve as the final tests. In chapter 3, we can see that the filtered tests achieve a high code coverage and is very effective in finding defects from the program under tested.

1.2 Examples

To explain Korat and constraint based search, we take the example of a binary tree. We first describe the working of Korat on this structure with three nodes. We then explain how Pairwise-Korat is applied to reduce the number of generated tests.

1.2.1 Binary Tree

Consider a Java implementation of a binary tree given in Figure 1.2. The static nested class Node models actual nodes in the binary tree. Each Node has a left and a right field, pointing to its child nodes. The BinaryTree class has a root field pointing to the root of the binary tree and an integer size, which stores the total number of reachable nodes. There are two structural constraints. One is acyclicity along left and right fields. The second is that the number of reachable nodes equals the size field. To verify these two constraints, a Java predicate is created and the implementation is given in Figure 1.2. Such an imperative predicate is conventionally called repOk [13]. In object oriented language domain, these constraints are often called class invariants.

```
public class BinaryTree {
   public static class Node {
      Node left;
      Node right;
   }
   private Node root;
   private int size;

   public boolean repOK() {
      if (root == null)
         return size == 0;
      // checks that tree has no cycle
      Set visited = new HashSet();
      visited.add(root);
      LinkedList workList = new LinkedList();
      workList.add(root);
      while (!workList.isEmpty()) {
         Node current = (Node) workList.removeFirst();
         if (current.left != null) {
            if (!visited.add(current.left))
               return false;
            workList.add(current.left);
         }
         if (current.right != null) {
            if (!visited.add(current.right))
               return false;
            workList.add(current.right);
         }
      }
      // checks that size is consistent
      return (visited.size() == size);
   }
}
```

Figure 1.2 A Java definition of binary trees and its repOk method. This method implements the two constraints: acyclicity along all paths and equality of size field and number of reachable fields.

1.2.2 Test Generation using Korat

To generate test inputs, Korat requires two Java methods: (a) a repOk method that checks the class invariants and (b) a set of bounds called finitization. Finitization method tells Korat how to bound the input space. The statements in the finitization method specify bounds on the number of objects to be used to construct instances of the data structure, as well as possible values stored in the fields of those objects. For example, the finitization in the binary tree example can take one object of class BinaryTree, three objects of class Node, and a fixed value of 3 for size field. A detailed implementation can be found at Figure 1.3.

```
public static IFinitization finBinaryTree() {
   IFinitization f = FinitizationFactory.create(BinaryTree.class);
   IObjSet nodes = f.createObjSet(Node.class, 3, false);
   f.set("root", nodes);
   f.set("Node.left", nodes);
   f.set("Node.right", nodes);
   IIntSet sizes = f.createIntSet(3, 3);
   f.set("size", sizes);

   return f;
}
```

Figure 1.3 A Java definition of finitization method to bound binary trees

The first line creates an "empty" finitization using FinitizationFactory.create factory method by passing it class under test as an argument. This line specifies that there is only one object of class BinaryTree.

Then, a set of three nodes is created by calling createObjSet method. This method takes several parameters:

- class of objects to be created,
- number of objects of the given class to be created,
- whether to include null or not,

which means that the second line creates a set of 3 Node objects which contain 3 instances of class Node.

The next thing to do is to associate certain fields with newly created object set. Fields BinaryTree.root, Node.left and Node.right are all of type Node and it is ok to have them all associated with this object set. That is what next three lines do.

Only field that is left to be bounded is BinaryTree.size so we simply create an IntSet with a single value of 3 and assign it to the field size. The above program will generate a total number of 5 valid nonisomorphic binary trees of 3 nodes.

1.2.3 Pairwise-Korat

The above example successfully explores all valid structures of binary trees with 3 nodes. But in an actual implementation, each node will also be assigned with a set of individual values. If each node takes 100 discrete values, the valid input size will grow from 5 to 5×10^6. The tests we present in this report have more complex structures and significant larger number of individual values for each node.

To reduce the input size, we first identify a series of constraints which removes those tests considered as “duplicate tests”. For example, let’s consider a binary tree with

3 nodes N0, N1 and N2, and each node takes a set of integer values from -2 to 2. If we want to test a program which returns true if any of the node has a value of 0, we may consider the following 2 test cases as duplicate test cases: [{N0=1, N1=-1, N2=1}, {N0=1, N1=1, N2=-1}]. Both of the two test cases might cover the same path in the program and only one of them is needed in a test requirement. Pairwise filters can then be applied on the generated tests to further shrink the input size.

PROPOSED METHODOLOGY

The purpose of implementing Pairwise-Korat is to generate test inputs for three Java applications developed for eBay online auction site. The input structures of these three tests are complex by nature and thus can't be generated by common combinatory test generators. For example, one of the projects requires inputs to provide several shipping instances in its input structure. The number of the shipping instances is flexible but the types of shipping methods are decided by the country and item price. One of these example inputs can be found in Figure 1.1. Due to the complex structures of these inputs files, the current test suite is generated manually by test engineers and only covers a small portion of the program. Besides the low coverage, the manual generation process is also very time-consuming and ineffective. As introduced in the above sections, Korat is a tool for generating structurally complex test inputs for Java applications and is an ideal candidate to be applied to replace current manually generated tests. However, since the input spaces of the applications are too big to enumerate, it is infeasible to apply Korat directly to those applications to generate test inputs. Thus, to enable automated test generation for these three industrial applications, we introduced a series of structural constraints and a pairwise test case filter to reduce the size of the generated tests.

2.1 Korat Generation

Korat requires a Java class declaration to generated instances of the class [14]. The current test inputs for the above projects, on the other hand, are stored in XML and text

files (as shown in Figure 1.1 and 2.6). To bridge this gap, we define three Java classes to be passed to Korat to generate input instances. One class implementation can be found in Figure 2.1.

```
public class SPSInput implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 9208165659797530794L;

    public int site;
    public int program;
    public int evaluationType;
    public Level level = new Level();
    public UserAttribute ua = new UserAttribute();
    public ETRS3Attribute ea = new ETRS3Attribute();

    public ETRS3Attribute getEa() {
        return ea;
    }

    public void setEa(ETRS3Attribute ea) {
        this.ea = ea;
    }

    public int getSite() {
        return site;
    }

    public void setSite(int site) {
        this.site = site;
    }
```

Figure 2.1 Java class containing test input fields

After the input class is implemented, the fields inside the class need to be bounded in finitization method. Though it is quite straightforward to define the boundaries in the finitization method, most of the fields have a very large valid input range, and it is not applicable to bound the fields with these input ranges directly (billions

of tests will be generated and system will run out of memory quickly if we choose to use the these input ranges). To resolve the difficulties, instead of using a continuous input range, we identify and select a few important data points to bound those fields. To help further understanding the approach, let's consider a concrete example. For a field named "ItemPrice", the input domain of this field could range from 0 to an arbitrary large number. If the item price is greater or equal to 500, then the item will be put into a special category. Since only a selective number of values can alter the execution path of the program, a single value can be chosen as a candidate to replace a certain input range. In this example, we end up choosing five values of (0, 1, 499, 500, 10000) to represent the input range of the ItemPrice field. We apply same process to all continuous fields in the finitization methods. An implementation of the finitization method to bound the fields can be found in Figure 2.2.

```
public static IFinitization finSPSRepOK(int nodesNum, int minSize, int maxSize) {
    IFinitization f = FinitizationFactory.create(SPSRepOK.class);

    IObjSet entry = f.createObjSet(SPSInput.class, 1, false);

    IIntSet nSites = f.createIntSet(0, 1, 3);
    IIntSet nPrograms = f.createIntSet(0, 1, 2);
    IIntSet nEvalTypes = f.createIntSet(0, 1, 0);
    IObjSet level = f.createObjSet(Level.class, 1, false);
    IObjSet ua = f.createObjSet(UserAttribute.class, 1, false);
    IObjSet ea = f.createObjSet(ETRS3Attribute.class, 1, false);

    IIntSet nTargetLevel = f.createIntSet(0, 1, 1);
    IIntSet nListOfNotMet = f.createIntSet(0, 1, 1);

    IBooleanSet isCreateMonthBeforMajorWindow = f.createBooleanSet();
    IIntSet txnCount = f.createIntSet(0, 1, 3);
    IIntSet feedbackPercentage = f.createIntSet(70,20,99);
    IBooleanSet businessSeller = f.createBooleanSet();
    IIntSet powerSellerLevel = f.createIntSet(0, 1, 0);
    IIntSet daysOnSite = f.createIntSet(0, 1, 3);

    IIntSet gmv = f.createIntSet(0, 1, 3);
    IIntSet defectTransaction = f.createIntSet(0, 1, 3);
    IIntSet closedCase = f.createIntSet(0, 1, 3);
    IIntSet validTrackingUpldHandling = f.createIntSet(0, 1, 1);



    f.set("spsinput", entry);
    f.set("SPSInput.site", nSites);
    f.set("SPSInput.program", nPrograms);
    f.set("SPSInput.evaluationType", nEvalTypes);
    f.set("SPSInput.level", level);
```

Figure 2.2 An implementation of the finitization method

2.2 Filter Constraints

Even though the modification we complete in finitization methods greatly reduced the number of generated tests, the number of tests is still too large to run in the automation framework. Besides, most of the generated tests are negative test cases [15, 16] and thus have low priorities. Executing such a large number of negative tests is time

consuming and the tests often exercise same parts of the program. To reduce the number of negative test cases, we add a series of constraints in the repOk methods to achieve this goal. Figure 2.3 shows an implementation of repOk method.

```
public boolean repOK() {
    /*if (spsinput.getSite() !=0)
        return false;*/
    /*if (etrsInput.getUa().isCreateMonthBeforMajorWindow() == false || etrsInput.getUa().isBusinessSeller() == false)
        return false;
    if (etrsInput.getLevel().getTargetLevel()==0 && etrsInput.getLevel().getListOfNotMet()!=0)
        return false;*/
    if (etrsInput.getUa().getDaysOnSite()==170)
        return false;
    /*if (etrsInput.getUa().getFeedbackPercentage()==70 || etrsInput.getUa().getFeedbackPercentage()==90 )
        return false;
    if (etrsInput.getLevel().getTargetLevel()!=0 && etrsInput.getLevel().getListOfNotMet()==0 )
        return false;*/
    if (etrsInput.getProgram()==1 && etrsInput.getCurrCode() != 0)
        return false;
    if (etrsInput.getProgram()==2 && etrsInput.getCurrCode() != 1)
        return false;
    if (etrsInput.getProgram()==3 && etrsInput.getCurrCode() != 2)
        return false;
    if (etrsInput.getProgram()==4 && etrsInput.getCurrCode() != 0)
        return false;
    if (etrsInput.getProgram()==0 && etrsInput.getCurrCode() != 0)
        return false;
    if (etrsInput.getDefectCnt()>10 && etrsInput.getDefectCnt() < 1000)
        return false;
    if ((etrsInput.getClosedCasesCnt()>10 && etrsInput.getClosedCasesCnt() < 19) ||
            (etrsInput.getClosedCasesCnt()>20 && etrsInput.getClosedCasesCnt() < 31))
        return false;
    if (etrsInput.getValidTkWithHandlingTimeRate()>0 && etrsInput.getValidTkWithHandlingTimeRate() < 70)
        return false;
    if (etrsInput.getLowDsrIad()>10 && etrsInput.getLowDsrIad() < 1000)
        return false;
    if (etrsInput.getLowDsrShipTime()>10 && etrsInput.getLowDsrShipTime() < 1000)
        return false;
    if (etrsInput.getNegFeedbkCnt()>10 && etrsInput.getNegFeedbkCnt() < 1000)
        return false;
    if (etrsInput.getNeutralFeedbkCnt()>10 && etrsInput.getNeutralFeedbkCnt() < 1000)
        return false;
    if (etrsInput.getSellerInitCancelTrxCnt()>10 && etrsInput.getSellerInitCancelTrxCnt() < 1000)
        return false;
    if (etrsInput.getSnadRetCnts()>10 && etrsInput.getSnadRetCnts() < 1000)
        return false;
    if (etrsInput.getOpenCnts()>10 && etrsInput.getOpenCnts() < 1000)
        return false;
    return true:
```

Figure 2.3 Structural constraints defined in repOK method to reduce the number of negative test cases

We will also illustrate our approach through an example. In the AddItem API project, a complete test input is required to provide two parameters: CountryCode and CurrencyCode. If we pass “US” as the CountryCode and “USD” as the CurrencyCode to the API, the API will send out an error message complaining that the country and

currency doesn't match with each other. Thus only a selected combination of the two parameters can trigger positive test flows, rest of the combinations exercise the same negative test flow. If we have 10 CountryCode and 10 CurrencyCode, Korat will generate 10 positive tests and 90 identical negative tests from the two parameters. By adding a constraint in the repOK method, Korat generates 11 tests, containing all 10 positive tests and 1 negative test. As shown in Figure 2.3, for each project, we implemented a set of such constraints to reduce the number of negative test cases.

2.3 Pairwise Filter

After two iterations of test reduction (filter constraint and finitization), Pairwise-Korat successfully reduce the number of tests from billions to a few hundred thousand without sacrificing much of the code coverage. However, the input size is still too big to fit in the current automation framework. It will take the framework up to days to execute all those tests. One might argue that this issue can be resolved by parallel testing. However, while parallel testing may help reducing the execution time for API tests, it is not the case when it comes to UI testing. There are two reasons: (1) UI automation is very time consuming and a single test could take 10-20 minutes to run, (2) Since each automated UI test case requires a web driver, a large number of UI tests can only be executed on a testing grid. Even in a large corporation, a testing grid usually contains only a few hundred machines (VMs). It could take an entire testing grid up to a week to execute one hundred thousand UI tests. So we need to further reduce the number of tests.

We adopt pairwise testing strategy as our final test reduction step to reduce the input size. The proposed algorithm first retrieve two random fields inside a Java input class (shown in Figure 2.1), and then find all unique combinations of the two fields and use the values as unique keys. The program will then iterate through all input instances generated by Korat and removed those instances which have the same key values. The list of instances will be stored as a merge candidate. The same iteration is repeated for all pairs and each iteration will generate a merge candidate. After all iterations complete, all candidates will be merged together and duplicated instances will be removed from the final output. This is not an optimum solution to find pairwise tests but it guarantees 2-pairwise coverage. The algorithm to implement the pairwise filter is illustrated in Figure 2.4, and the actual implementation is shown in Figure 2.5.

```
function pairwiseGen
   pairs ← GetAllPairs()
   for each pair in pairs
      for each instance
         key ← GenerateKeyValue(pair)
         if key is in keylist
            remove instance
         else
            addKeytoKeylist()
            addInstancetoInstancelist()
         end if
      end for
   end for
   mergeInstancelists()
end function
```

Figure 2.4 Algorithm for generating pairwise test case

```
public class AllPairwise {

    public static List<String> fieldNames = new ArrayList<String>();
    public static List<Pair> fieldPairs = new ArrayList<Pair>();
    public static List<SPSMetrics> list = new ArrayList<SPSMetrics>();
    public static List<ETRS3Metrics> spsList = new ArrayList<ETRS3Metrics>();

    public static List<SPSMetrics> getAllPairs(List<SPSMetrics> input) {
        System.out.println("Performing pairwise comparison...");
        Class<?> c = SPSMetrics.class;
        Field[] fields = c.getDeclaredFields();
        for (Field field : fields) {
            fieldNames.add(field.getName());
        }

        for (int i = 4; i < fieldNames.size(); i++) {
            for (int j = i + 1; j < fieldNames.size(); j++) {
                fieldPairs.add(new Pair(fieldNames.get(i), fieldNames.get(j)));
            }
        }

        for (Pair pair : fieldPairs) {
            // System.out.println(pair.attr1 + ", " + pair.attr2 + ", " +
            // fieldPairs.size());
            list.addAll(PairwiseGenerator.pairsieSpsInput(input, pair.attr1,
                    pair.attr2));
        }

        Set<SPSMetrics> set = new HashSet<SPSMetrics>();
        set.addAll(list);
        list.clear();
        list.addAll(set);

        System.out.println("End of pairwise comparison. Number of complete pairwise test cases are: "
                + list.size());

        return list;

    }
```

Figure 2.5 Implementation of the pairwise filter

The pairwise filter reduces the number of tests to 568 (SDB), 624 (AddItem), and 356(eMBG). The size of tests is ideal for automated testing.

2.4 Input Format Conversion

After test size reduction, the system obtains a set of test suites stored as Java instances. The final step is to convert those Java instances to appropriate formats to integrate with the automation framework. AddItem is an API testing project and it

requires XML input files. SDB and EMBG are testNG [17] projects and they require csv input files. It is straight forward to generate XML files from Java objects and Pairwise-Korat adopts open source framework openCSV to write the Java instances to csv files. A sample generated csv input file is shown in Figure 2.6.

country	currency	Description	ListingDurati	ListingType	Location	PayPalEmail	Quantity	Site	Title	PostalCode	ConditionID
AU	AUD	Alloy Generated Item: AUS	Days_7	FIXED_PRICE_ITEM	Canberra	ecaf_api_se	1	AUSTRALIA	Alloy Item: A	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	1	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	3	US	Alloy Item: U	95125	1000
US	USD	Alloy Generated Item: US_	Days_7	FIXED_PRICE_ITEM	SanJose	ecaf_api_se	2	US	Alloy Item: U	95125	1000

Figure 2.6 A test suite stored in .csv file

EXPERIMENTAL RESULTS

This section presents the performance results of the Pairwise-Korat. The test generation is performed on a Mac machine with a 2.5GHz Intel Core i7 processor and 16 GB RAM, using Java SDK 1.8.0 JVM. To evaluate the performance of Pairwise-Korat, we implemented it in three projects developed in eBay. We first present Pairwise-Korat's performance for test case generation, then compare it with the existing tests that manually generated by test engineers, and finally present Pairwise-Korat's performance on code coverage. We will also analyze some distinct bugs found by Pairwise-Korat.

Project	Code Coverage	# of Tests generated	# of Korat generated tests	Total time
SDB	95%	568	386695	80.43s
AddItem	100%	624	387175	109.31s
eMBG	83%	356	152615	31.23s

Table 3.1 Performance of Pairwise-Korat

From the Table 3.1, we can see that Pairwise-Korat achieves very high code coverage on SDB and AddItem projects. The Pairwise filter successfully reduced the number of tests from 10^5 to 10^2. Although it is not infeasible to run Korat generated tests directly, the performance of Korat is robust. E.g., Korat generated 387175 tests in less than 2 minutes. SDB and Additem are legacy projects and thus it is easier to identify importance values for Finitization methods. eMBG is a new project and identifying data

points is more difficult due to the lack of implementation details. A smaller number of tests are generated because less data points are selected in the Finitization method.

Project	Existing Tests		Pairwise-Korat		
	Coverage	**Test Input #**	**Coverage**	**Test Input #**	**# of Defects found**
SDB	85%	315	95%	568	5
AddItem	91%	277	100%	624	3
eMBG	78%	128	83%	356	28

Table 3.2 Performance comparison between existing manual solution and Pairwise-Korat

We create Table 3.2 to compare the performance of Pairwise-Korat with existing manually generated tests. Pairwise-Korat outperforms current manually generated tests in code coverage. Pairwise-Korat also reveals new defects from the programs. SDB and AddItem are legacy projects and Pairwise-Korat successfully revealed defects from those two live projects. When implement Pairwise-Korat, eMBG was still under development and the tests generated by Pairwise-Korat uncovered 28 new defects from the project. There are two reasons that could account for the differences between Pairwise-Korat and the manual solution. Since existing tests are generated by test engineers, human bias could affect the generated tests and some scenarios could be left out in the test plan. Another reason is that often Pairwise-Korat generates a much greater number of instances

than human does, since manual generation takes a greater amount of time and efforts. The only way for the existing tests to match up on the coverage is to add more tests.

After comparing the performance of Pairwise-Korat with existing manual solution, it is safe to claim that automated test generation using Pairwise-Korat not only removes the laborious human effort from test generation, but also reduces human bias and thus can achieve higher test coverages.

DEFECT ANALYSIS

We select a few classic defects found by Pairwise-Korat to further study the performance of the framework. The analysis also provides us a direction for future enhancement.

Defect A: In SDB project, one User Interface component is not displayed (broken) when the value of DSR field is less than 5. The root cause of this defect is that the component flag is triggered by the value of DSR field instead of lowDSR field. In a correct behavior, the flag should be controlled by lowDSR field and the component should be displayed when the value of DSR field is less than 5.

Pairwise-Korat generated test instances with the value of lowDSR are greater than 5 but DSR value is less than 5. The existing tests doesn't have such a test case since lowDSR is expected to be less or equal to DRS. But such scenario could happen in real life when Database inserted incorrect records into those fields, and we shouldn't display a broken page in such a scenario.

Defect B: In AddItem project, system puts an item in a lower priority category when item meets a higher standard. The root cause of this defect is that the program failed to convert the local currency to US dollars correctly. The program should convert the local currency to US dollars and then evaluate the level if the item.

Pairwise-Korat created one input for this currency and it uncovered this defect. Existing tests cover only a selected number of currencies and failed to uncover this defect.

Defect C: In eMBG project, system displays a notification message designated for US on UK and Germany sites. The root cause of this defect is that the system failed to add condition check when displaying this notification message on UK and Germany sites. In a correct behavior, different messages should be shown on the pages.

Similar to the above case, Pairwise-Korat created inputs which invoke this message on UK and Germany sites. Since the condition of showing such a message is very complicated, existing tests don’t cover this message on all sites.

CONCLUSIONS AND FUTURE WORK

This report presents Pairwise-Korat, a test generation framework for automated testing of industrial applications. Built on top of Korat, a tool for constraint-based generation of structurally complex test inputs for Java programs, Pairwise-Korat implements a series of test reduction methods to reduce the size of raw inputs generated by Korat. The updated tests are then converted to different format to serve as test inputs for 3 industrial projects developed at eBay. Given a list of generated test inputs, Pairwise-Korat uses filter constraint and updated finitization method to remove redundant tests from the list. Pairwise-Korat then adopts a pairwise filter to select 2-pairwise tests from the updated list. Finally, Pairwise-Korat outputs the generated Java instances to different files to be used as test inputs. This report illustrates the use of these test input files for testing several industrial applications. The experimental results show that it is feasible to generate test cases for industrial applications using automated method, even when the search space for raw inputs is very large. This report also compares Korat with the existing manual generated test inputs. The experiments also show that Korat generated test inputs achieved higher code coverage than the manually generated test inputs.

A future enhancement of the work is to make Korat generating pairwise test cases directly instead of applying a pairwise filter on the generated instances. This approach could largely reduces the test generation time and allows user to specify a much larger input space by defining less constraints and larger boundaries. Another future enhancement could be to further reduce the number of test cases generated by Pairwise-Korat. Since most of the defects we find can be revealed by multiple inputs in the

generated file, future changes can be made on the finitization methods to remove those test cases to reduce the input size.

References

[1] B. Beizer. Software Testing Techniques. International Thomson Computer Press, 1990

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, September 1976.

[3] J. C. Huang. An approach to program testing. ACM Computing Surveys, 7(3), 1975.

[4] James C. King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[5] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. IEEE Transactions on Software Engineering, 2(4), 1976.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In ISSTA ’02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 123–133, New York, NY, USA, 2002. ACM.

[7] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In ISSTA ’04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pages 133–142, New York, NY, USA, 2004. ACM.

[8] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. Automated Software Engg., 11(4):403–434, 2004.

[9] K. Stobie. Model based test generation and abstract state machine language, 2003. http://www.sasqag.org/ pastmeetings/asml.ppt

[10] Black, Rex. Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. New York: Wiley. p. 240. 2007

[11] D.R. Kuhn, D.R. Wallace, A.J. Gallo, Jr. . "Software Fault Interactions and Implications for Software Testing" (PDF). IEEE Trans. on Software Engineering 30 (6). 2004

[12] Practical Combinatorial Testing. SP 800-142. Natl. Inst. of Standards and Technology. 2010.

[13] B. Liskov and J. Guttag. Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A Tool for Generating Structurally Complex Test Inputs. *29th International Conference on Software Engineering (ICSE 2007)* Research Demo Paper. Minneapolis, MN. May 2007

[15] BCS SIGIST: Standard Glossary of Testing Terms (British Standard BS 7925-1)

[16] Beizer, Boris. Software Testing Techniques, van Nostrand Reinhold, 1990

[17] Beust, Cédric and Suleiman, Hani. Next Generation Java Testing, 2007

Vita

Hua Zhong was born in Hefei, Anhui, P.R.China. He received the degree of Bachelor of Engineering from East China Normal University in August 2007 and the degree of Master of Science in Computer Science from University of Alabama at Birmingham in December, 2009. During the following years, he was employed as a software engineer in Houston and Austin. In January 2013, he entered the Software Engineering Program of the Graduate School at The University of Texas at Austin.

Permanent email address: hzhong1121@gmail.com

This report was typed by the author.