

Copyright

by

Aman Pervaiz

December 2007

**Implementing Deryaft for Generating Invariants of Structurally  
Complex Data**

**by**

**Aman Pervaiz**

**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**

**December 2007**

**Implementing Deryaft for Generating Invariants of Structurally  
Complex Data**

**Approved by  
Supervising Committee:  
Sarfraz Khurshid**

---

**Dewayne E. Perry**

---

## **Dedication**

I dedicate this report to Dr. Sarfraz Khurshid and his Software Verification and Validation Research Group.

## **Abstract**

# **IMPLEMENTING DERYAFT FOR GENERATING INVARIANTS OF STRUCTURALLY COMPLEX DATA**

Aman Pervaiz, MSE

The University of Texas at Austin, 2007

Supervisor: Sarfraz Khurshid

This report presents an implementation of a previously presented framework to systematically detect structural properties within structures. The implementation is in java and uses the standard java reflection to detect the basic fields and qualities. The implementation expects a list of objects as input and extracts a variety of properties and performs dynamic analysis on the extracted properties. The objects are treated as graphs with a root and also patterns in the primitive fields are also analyzed. The implementation then generates a repOK for the given list of structures and tests that repOK against that list. The repOK can be used for structural enumeration in Korat and Alloy Analyzer.

## Table of Contents

<b>List of Figures</b> .....	ix
<b>Chapter 1: Introduction</b> .....	<b>1</b>
<b>Chapter 2: Technologies Used</b> .....	<b>2</b>
JRE	2
Eclipse.....	2
Reflection API .....	2
<b>Chapter 3: Design</b> .....	<b>3</b>
<b>Chapter 4: Implementation Details</b> .....	<b>5</b>
Inspection.....	5
Preliminary Analysis.....	6
Detection Rules File.....	10
Priority Queue Based Algorithm .....	13
Data structure for Storing the Properties .....	15
Known Data structures.....	16
Interesting Properties .....	18
Linear .....	18
Array .....	18
N-ary .....	18
Multiple structures .....	18
Cyclicity .....	18
Left Tree Order in Array.....	18
Right Tree Order in Array.....	19
Ascending array .....	19
Descending array .....	19
Max heap order in Array.....	19
Min heap order in Array .....	19
Binary max heap .....	19

Binary min heap.....	20
Binary left order.....	20
Binary right order.....	20
Completeness.....	20
Array Completeness.....	20
Max Height.....	21
Max Balance.....	21
Number of Nodes.....	21
Root Null.....	21
Alternating n-ary tree structures.....	21
First structure is last.....	21
Breadth ascending order.....	22
Breadth descending order.....	22
Breadth Array ascending order.....	22
Breadth Array descending order.....	22
Dynamic Analysis.....	22
REPOK Generation.....	24
Automated Testing.....	25
<b>Chapter 5: Problem with Fewer Samples.....</b>	<b>26</b>
<b>Chapter 6: Experiments.....</b>	<b>28</b>
Intentional Naming Tree.....	28
Heap Array.....	30
Binary Search Tree (Complete).....	32
Sorted Linked List.....	34
Max Heap (max balance is 1).....	35
<b>Chapter 7: Discussion.....</b>	<b>38</b>
Limitations.....	38
Optimization of Repeated Traversals.....	38
Integration with other Software Analysis Framework.....	39
Static Analysis for Optimizing Generation.....	39

<b>Chapter 8: Related Work</b> .....	40
Static Analysis .....	40
Combined Dynamic/Static Analysis .....	41
<b>Chapter 9: Conclusion</b> .....	42
<b>References</b> .....	<b>43</b>
<b>Appendix</b> .....	<b>46</b>
Instructions to Run Code .....	46
API	
Class	
Methods.....	46
DetectionEngine ().....	47
getProperties ().....	47
writePropertiesToFile ().....	47
getRepOK ().....	48
writeRepOKToFile ().....	48
Code Website.....	48
SOME CONFIGURATION XMLFILES (samples) .....	49
KnownModels.....	49
KnownRules.....	50
BackBone.....	52
DynamicAnalysis.....	52

**Vita**

## List of Figures

Figure 1:	Overall design of Deryaft Implementation. ....	4
Figure 2:	Algorithm to compute a core set. ....	8
Figure 3:	Example of a structure with a wrapper class ....	9
Figure 4:	Core algorithm for maintaining the queue ....	14
Figure 5:	Basic data structure to store properties. ....	16
Figure 6:	Example of the known data-structures file ....	17
Figure 7:	An example of a small tree. ....	26
Figure 8:	Intentional Naming Tree. ....	28
Figure 9:	Array representing a heap ....	30
Figure 10:	Complete Binary Search Tree ....	32
Figure 11:	Sorted Linked List. ....	34
Figure 12:	Max Heap with maximum balance of 1 ....	35

## Chapter 1: Introduction

Checking programs that manipulate dynamically-allocated, structurally complex data is notoriously hard. Existing dynamic and static analyses [19, 4, 8, 20, 2, 10] that check non-trivial properties of such programs impose a substantial burden on the users, e.g., by requiring the users to provide invariants, such as loop or representation invariants, or to provide complete executable implementations as well as specifications. Previously Deryaft, a novel framework for generating representation invariants of structurally complex data given a (small) set of structures was presented. This report presented an implementation of the framework in java.

The generated invariants serve various purposes. Foremost, they formally characterize properties of the given structures. More importantly, they facilitate the use of various analyses. To illustrate, consider test generation using a constraint solver, such as Korat [4], which requires the user to provide detailed invariants. Deryaft enables using just a handful of small structures to allow these solvers to efficiently enumerate a large number of tests and to systematically test code. The generated invariants can similarly be used directly in other tools, such as ESC/Java [8], that are based on the Java Modeling Language [17], which uses Java expressions, or simply be used as assertions for runtime checking, e.g., to check if a public method establishes the class invariant. The invariants even enable non-conventional assertion-based analyses, such as repair of structurally complex data, e.g., using the Juzi framework [15].

## Chapter 2: Technologies Used

### JRE

This Deryaft java based implementation is currently supported on Sun's java. More specifically JRE System Library (jre1.6.0-02) was used. Various reference libraries are being used. The major ones are listed below.

1. **import** java.lang.reflect.Field;
2. **import** java.util.Hashtable;
3. **import** java.util.Vector;
4. **import** java.util.ArrayList;

### ECLIPSE

Eclipse was the IDE choice to implement in java. Eclipse is an open-source software framework written primarily in Java. In its default form it is a Java IDE, consisting of the *Java Development Tools* (JDT) and compiler (ECJ).

### REFLECTION API

To extract the basic properties from the structures, Reflection API is used. Reflection gives the code access to internal information for classes loaded into the JVM and allows writing code that works with classes selected during execution, not in the source code.

## Chapter 3: Design

The overall design conforms to the original Deryaft design proposal. The implementation has been structured in to separate components logically defined. The components interact with each other by exchanging intermediate data represented in standard structures that store the intermediate analysis data. The components that need external configuration do that by reading the configuration data through XML files. The XML files have to be in the same location as the other source files.

There are distinct stages of the implementation (figure 1). First preliminary analysis is performed on the initial data. Then the core algorithm runs, which maintains a priority queue of properties. After all the properties have been identified, dynamic analysis is performed on the properties. Once the dynamic module generates the final results, those results are stored in an output file. Also the results are used to generate repOK that contains code to test the fundamental properties of the structures. The next stage after generating the repOK is the automated testing of the repOK against the originally given list of objects.

All the modules have been explained in detail later along with the details of core algorithm and the data structures used to store the analysis results. Also the appendix gives the sample configuration xml files that the implementation uses.

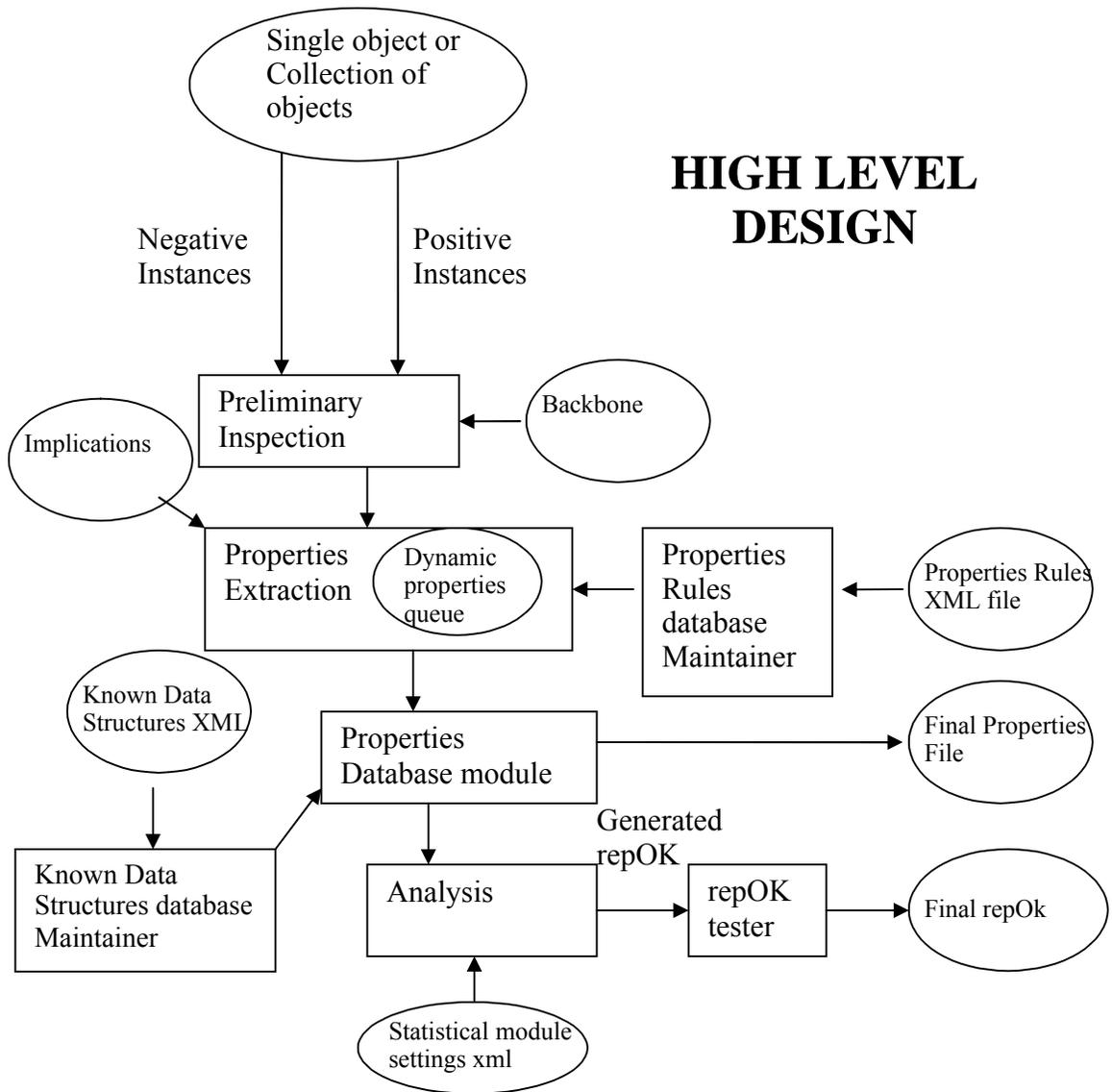


Figure 1: Overall design of Deryaft Implementation.

## Chapter 4: Implementation Details

The implementation accepts objects in a variety of formats. It then performs a preliminary analysis on the objects. The analysis can be guided by specifying core and derived fields manually using an XML file called the backbone. The results are then passed on to the “Properties extractor” module that maintains a queue of properties and adds and removes properties to be tested in the queue, based on the current property evaluation and the properties rules that are maintained in a rules database (extracted from the rules XML file). This module also eliminates redundant properties by consulting an XML file that has implication rules listed in it to determine the redundancies. The properties extractor module writes the results in a file if desired. It also passes the discovered properties database to the dynamic module that uses the statistical module settings to generate the preliminary repOk code. This code is then compiled and tested against the list of the objects initially given for the training. If the repOK passes upon all the objects, then it is presented as the final repOK.

### INSPECTION

Deryaft requires both negative and positive examples to infer properties of structures. In the given implementation these examples can be provided in multiple ways to facilitate different use cases.

1. The best approach to use Deryaft is by using the API as specified in the Appendix in detail.

2. The other way to give structures to the Deryaft implementation is by generating the required structures and then serializing the objects in a file. This file can then be provided by giving the “-objectFile” option to the Deryaft jar file. This technique works well, however it imposes the restriction that the structures being generated must implement the class “Serializable” and hence must have a unique static “serialVersionUID” for the respective class that implements the structure.
  
3. The second way is to call the “inspect” method in the main class (DetectionEngine). This method takes a vector of objects and assumes all the objects within the vector to be of the same kind of structures. Several overloaded functions are also provided that accept
  - a. A single object.
  - b. An array of objects.
  - c. A Collection of objects.

### **PRELIMINARY ANALYSIS**

The first thing that follows the receiving of objects is that the fields are derived from the given objects. The fields can be primitive or non-primitive. A simple match against known primitive fields segregate the fields set into the required primitive and non-primitive fields.

For the non-primitive fields, the fields can be either core or derived. This segregation can be done manually by providing a backbone XML file. If the backbone is provided, then Deryaft does not attempt to segregate the fields but rather inspects only

the core and derived fields specified in the XML file. An example of a backbone file is given below.

```

<BackBone>
  <Reference>
    <Core>
      <field>Node.left</field>
      <field>Node.right</field>
    </Core>
    <Derived>
      <field>Node.parent</field>
    </Derived>
  </Reference>
</BackBone>

```

However if the files is not given as input, then Deryaft automatically segregates the fields. For a given set,  $S$ , of structures, let  $F$  be the set of all reference fields. The following algorithm as suggested previously in the design of Deryaft is given below.

```

Set coreFields(Set ss) {
  // post: result is a set of core fields with respect to the
  // structures in ss
  Set cs = allClasses(ss);
  Set fs = allReferenceFields(cs);
  foreach (Field f in fs)
    Set fs' = fs - f;
    boolean isCore = false;
    foreach (Structure s in ss) {
      if (reachable(s, fs') != reachable(s, fs)) {
        isCore = true;
        break;
      }
    }
    if (!isCore) fs = fs';
  }
  return fs;
}

```

Figure 2: Algorithm to compute a core set. The method `allClasses` returns the set of all classes of objects in structures in `ss`. The method `allReferenceFields` returns the set of all reference fields declared in classes in `cs`. The method `reachable` returns a set of objects reachable from the root of `s` via traversals only along the fields in the given set.

**Definition 1.** A subset  $C \subseteq F$  is a core set with respect to  $S$  if for all structures  $s \in S$ , the set of nodes reachable from the root  $r$  of  $s$  along the fields in  $C$  is the same as the set of nodes reachable from  $r$  along the fields in  $F$ .

In other words, core set preserves reachability in terms of the set of nodes. Indeed, the set of all fields is itself a core set. We aim to identify a *minimal* core set, i.e., a core set with the least number of fields. To illustrate, the set containing both the reference fields `header` and `next` in the example from Section 2 is a minimal core set with respect to the given set of lists.

**Definition 2.** For a core set  $C$ , the set of fields  $F - C$  is a derived set.

Since `elem` in Section 2 is a field of a primitive type, the `SinglyLinkedList` example has no fields that are derived. Our partitioning of reference fields is inspired by the notion of a *back-bone* in certain data structures [19].

**Algorithm.** The set of core fields can be computed by taking each reference field in turn and checking whether removing all the edges corresponding to the field from the graph changes the set of nodes reachable from root. Figure 3 gives the pseudo-code of an algorithm to compute core fields.

Once the core and derived fields are detected, the structures are segregated in to various basic types by doing a simple inspection using the Reflection API. The only complication that is faced over here is that sometimes structures are in fact embedded in a main class. For example, a linked list can have a wrapper class that contains a root pointer and an integer field denoting the number of nodes in the linked list. An example is given below.

```
Class LinkedListWrapper  
{  
    LinkedList root;  
    int nodesCount;  
}
```

Figure 3: Example of a structure with a wrapper class

The implementation does check this and tries to detect where the actual structure lies. It can detect the data-structure up to three levels of wrapper classes. It also tries to make sense of the other fields in the wrapper class if they are present. For example the implementation can detect the integer field in the above example and will try to relate the value in the field to the number of links in the case of a linked list or to balance, height and number of nodes in case of trees.

The basic types in which the structures are segregated are as follows.

1. Linear (e.g. Singly linked list)
2. Binary Structures (e.g Binary Trees)

3. N-ary Trees
4. Array based
5. Multiple structures based (e.g. Intentional Naming Trees)

The linear property is present if there is only one self-referencing field. (A self-referencing field is the field whose data-type is the same as the primary structure being detected-excluding the wrapper classes.)

Binary structures have two self-references. From there on, the segregation between a doubly linked list and binary trees is easy to make. It is handled later though.

N-ary trees are supported too. However the children of a node in an n-ary tree can only be detected if they are implemented as a Collection, Array, Vector, ArrayList or simply List. Support for other implementations can be trivially added. The implementation given does cover most of the implementations though.

Array based structures are also detected and an effort is made to tell if they are serialized binary trees or some heap implementations.

Finally the tricky part is to try to detect complex structures like Intentional Naming trees. They have two different alternating structures with more complex properties. They are supported natively too and would be discussed later with examples.

## **DETECTION RULES FILE**

At this point the implementation needs two files to proceed. The first is a rule file and the other is an XML file that tells what properties are enabled or disabled. The rule file has a single rule for each property. For every property, there are corresponding positive and negative properties that are arranged in the order of priority. The first positive property has the most priority and the last positive rule has the least property. Same cases apply to the negative rules.

Here is an example for clarification. The following rule is present in the xml file corresponding to the property “linear”.

```
<Rule linear>
  <Positive>
    <posistiveRule>cyclicity</posistiveRule>
    <posistiveRule> ascendingOrder </posistiveRule>
    <posistiveRule> descendingOrder </posistiveRule>
  </Positive>
</Rule>
```

This rule means that if the head of the priority queue has linear property (a linked list like structure) and it evaluates to be positive, then when we look up the rule for the “linear” property it states that we must add the property of “acyclicity”, “ascendingOrder” and “descendingOrder”. Cyclicity means that the next property that should be evaluated is whether the structure has a cycle. The next properties with lower priorities to be checked are “ascendingOrder” (whether linear structure is sorted in an ascending order) and “descendingOrder” (whether the linear structure is sorted in a descending order).

Once the properties are added to the queue, the cyclicity is at the front of the queue. Now the cyclicity property is evaluated as the next step. Depending on the outcome of the “cyclicity” property, there exists a corresponding rule in the xml file.

```
<Rule cyclicity>
  <Negative>
    <negativeRule> ascendingOrder </negativeRule>
    <negativeRule> descendingOrder </negativeRule>
  </Negative>
  <Positive>
    <negativeRule> cyclicAscendingOrder </negativeRule>
    <negativeRule> cyclicDescendingOrder </negativeRule>
  </Positive>
</Rule>
```

So if for example cyclicity property is found to be true, this rule is executed. It can be seen that the rule has negative and positive properties. What it means is that the properties listed under negative must be eliminated from the queue if already present. So the previously added properties “ascendingOrder” and “descendingOrder” would now be taken off from the queue. However two new properties will be added which are ascending order in a cyclic list and descending order in a cyclic list.

These rules can be added/deleted/modified any time in the xml file.

The second file needed is an XML file that tells what properties are enabled or disabled at that point. An example is given below.

```
<DynamicAnalysis>
<Global>
<Property>
<Name> height-difference </Name>
<Status> enabled </Status>
</Property>
<Property>
<Name> height-difference </Name>
<Status> enabled </Status>
</Property>
</Global>
<Local>
<Property>
<Name> unary </Name>
<Status> enabled </Status>
</Property>
<Property>
<Name> two-cycle </Name>
<Status> enabled </Status>
</Property>
</Local>
</DynamicAnalysis>
```

There is also support for specifying an Implication XML file that helps in removing redundant properties. The implications exist in the XML file as pairs. An example is shown below.

```
<Implications>
<Implication>
<Property>
<Name>acyclic</Name>
<Value>true</Value>
</Property>
<ImpliedProperty>
<Name>directed-acyclic</Name>
<Value>true</Value>
</ImpliedProperty>
</Implication>
<Implication>
<Property>
<Name>nearly-complete</Name>
<Value>true</Value>
</Property>
<ImpliedProperty>
<Name>height-difference</Name>
<Value>1</Value>
</ImpliedProperty>
</Implication>
</Implications>
```

This XML file suggests that if a structure is acyclic, then it is also directed acyclic. Also if the structure is nearly complete, then the “height difference of 1” property is redundant.

## **PRIORITY QUEUE BASED ALGORITHM**

To detect the properties, a runtime queue is maintained that contains the properties that are still to be tested. This queue interacts with the rules as well. The rules that are stored in an xml file are read in an internal data structure. The algorithm removes one property from the head of the queue, evaluates it and then based on its outcome

consults the rule database. Depending on the particular rules for that property, some more properties may be added in the queue or may be removed. More specifically within each rule, every property has negative and positive properties associated with it. Negative properties imply rules to be removed from the dynamic queue while the positive properties mean new properties to be added on the tail of the queue.

The algorithm is discussed in detail next .

```

while(!dynamicPropertiesQueue.empty())
{
    Property property = dynamicPropertiesQueue.getHead(); //remove property from the head
    boolean result = PropertiesEvaluator.evaluate(object,property);
    if(result) //if the property holds
    {
        discoveredProperties.addPositiveProperty(property);//add as a positive property
        if(knownRules.isPropertyPresent(property))//is there a rule for the property
        {
            if(knownRules.negativeRulesPresent(property))//are there any negative rules
            {
                Vector<Property> negativeProperties = knownRules.getNegativeProperties(property);
                for(int i = 0;i<negativeProperties.size();i++)
                {
                    if(dynamicPropertiesQueue.isPropertyPresent(negativeProperties.get(i)))
                    {
                        //remove property if present
                        dynamicPropertiesQueue.removeProperty(negativeProperties.get(i));
                    }
                }
            }
        }
        if(knownRules.positiveRulesPresent(property))//are there any positive rules
        {
            Vector<Property> positiveProperties = knownRules.getPositiveProperties(property);
            for(int i = 0;i<positiveProperties.size();i++)
            {
                if(!dynamicPropertiesQueue.isPropertyPresent(positiveProperties.get(i)))
                {
                    //add property if not present
                    dynamicPropertiesQueue.addProperty(positiveProperties.get(i));
                }
            }
        }
    }
    else
    {
        //property does not hold so add as a negative instance
        discoveredProperties.addNegativeProperty(property);
    }
}

```

Figure 4: Core algorithm for maintaining the queue

## **DATA STRUCTURE FOR STORING THE PROPERTIES**

All the properties are stored as a vector of properties. Properties are of two types. The first type is the set of properties that are either true or false for example cyclicity or order in a binary tree. The second type is those properties that have a numeric value like that maximum height of a tree or the maximum balance or the length of a list.

For properties that are either present or absent, the data structure keeps tracks of the number of total samples received, positive samples, negative samples, positive samples percentage and negative samples percentage. Also there are fields to hold the property name and a Boolean field to tell if that property is applicable to the current data structure being evaluated. For example for a linear structure, binary heap max property would be irrelevant.

For the properties that have a value, there is vector that resides within that property that holds all the samples results. The property also keeps track of the average value of such property, maximum value received so far, the minimum value so far and also whether all the results received are equal. For example for the maximum height property, we can retrieve average height, maximum height, minimum height etc. The meticulous storage of these properties enable for detailed analysis later by the Statistics module that may set significance to these properties (For example for an AVL tree, the maximum balance is important and its absolute value must be less than or equal to one.)

Some properties are related to a particular field of the structure. For example a tree structure may hold several data types in its node and the structure may be a binary search tree for one of the primitive fields and not for others. Properties like Max heap property, min heap property, tree order etc. are evaluated for all the primitive fields that reside in the structure. Hence a structure may have a certain property for a specific

primitive field. Therefore in the storage of properties, we also store whether that property holds for a certain “Field”.

```
public class DSProperties
{
    Vector <DSProperty>dataStructureProperties;
}
public class DSProperty
{
    String PropertyName;
    Vector<Integer> valueVector; //could be balance, number of nodes or height etc

    int propertyType; //0 means that the property can have just two values
    int numSamplesEncountered;
    int positiveSamples;
    int negativeSamples;
    double positivePercentage;
    double negativePercentage;
    double averageValue;
    double minimumValue;
    double maximumValue;
    double areAllValueFieldsEqual;
    boolean Applicable;
    Field relevantField; //null means that property does not relate to a particular field
}
```

Figure 5: Basic data structure to store properties.

## **KNOWN DATA STRUCTURES**

Once all the objects are evaluated, the implementation reads another xml file that contains the names of the common structures and their relevant properties and stores it in

a known structures data base. These structures can be added/deleted and modified as desired.

Below are some sample entries of the xml file.

```
<Model Binary Search Tree>
<property binary> 1 </property>
<property cyclicity> 0 </property>
<property LeftOrder> 1 </property>
</Model>
<Model MinHeap>
<property binary> 1 </property>
<property cyclicity> 0 </property>
<property Completeness> 1 </property>
<property MinHeapProperty> 1 </property>
</Model>
<Model MaxHeap>
<property binary> 1 </property>
<property cyclicity> 0 </property>
<property Completeness> 1 </property>
<property MaxHeapProperty> 1 </property>
</Model>
<Model AVL Tree>
<property binary> 1 </property>
<property cyclicity> 0 </property>
<property maxBalance> 1 </property>
</Model>
<Model Unordered Linked List>
<property linear> 1 </property>
<property cyclicity> 0 </property>
</Model>
```

Figure 6: Example of the known data-structures file

The implementation after having evaluated all the objects compares the discovered properties against the database and attempts to discover if they are well known data structures.

## **INTERESTING PROPERTIES**

The implementation attempts to discover a variety of structural properties. These are listed below.

### **Linear**

This property tells whether a property has one self reference or multiple self references.

### **Array**

This property tells whether the major structure is an array. The treatment of an array like structure is different from a self-reference based.

### **N-ary**

This property tells whether the major structure is an n-ary tree.

### **Multiple structures**

This property tells whether the structure has different objects linked with each other like in an Intentional Name Tree.

### **Cyclicity**

This property tells whether the structure whether binary, linear or n-ary etc. has a cycle in it.

### **Left Tree Order in Array**

If the structure is an array, this property is true if when attempting to deal with the array as a binary tree implementation, the binary tree has left child less than the current and the right child greater than the current.

**Right Tree Order in Array**

If the structure is an array, this property is true if when attempting to deal with the array as a binary tree implementation, the binary tree has left child greater than the current and the right child less than the current.

**Ascending array**

If the structure is an array, this property is true if that array is sorted in an ascending order.

**Descending array**

If the structure is an array, this property is true if that array is sorted in a descending order.

**Max heap order in Array**

If the structure is an array, this property is true if when attempting to treat that array as a binary tree the max heap property is discovered.

**Min heap order in Array**

If the structure is an array, this property is true if when attempting to treat that array as a binary tree the min heap property is discovered.

**Binary max heap**

This property tells whether the structure is a binary tree that has the max heap property (parent is always greater than the children). This property is for a particular field in the structure.

**Binary min heap**

This property tells whether the structure is a binary tree that has the min heap property (parent is always smaller than the children). This property is for a particular field in the structure.

**Binary left order**

This property tells whether the structure is a binary tree that has the binary left order property (parent is greater than the left child but smaller than the right). This property is for a particular field in the structure.

**Binary right order**

This property tells whether the structure is a binary tree that has the binary right order property (parent is smaller than the left child but greater than the right). This property is for a particular field in the structure.

**Completeness**

This property tells whether the binary or an n-ary tree is complete.

**Array Completeness**

This property tells whether the array when interpreted as a binary tree is complete or not.

**Max Height**

This property denotes the maximum height in a binary or an n-ary tree.

**Max Balance**

This property denotes the maximum balance in a binary tree.

**Number of Nodes**

This property denotes the number of nodes in a binary or an n-ary tree.

**Root Null**

This property tells whether the value field of the root in an n-ary or a binary tree is null.

**Alternating n-ary tree structures**

This property tells whether in an n-ary tree there are two structures that are alternating. (e.g. Intentional Naming Tree)

**First structure is last**

This property tells whether in an n-ary tree with two alternating structures, the structure at the leaves is the same as the root structure.

**Breadth ascending order**

This property tells whether when parsing a binary or an n-ary tree breadth first, for some value field, the ascending order holds.

**Breadth descending order**

This property tells whether when parsing a binary or an n-ary tree breadth first, for some value field, the descending order holds.

**Breadth Array ascending order**

This property tells whether when interpreting an array as a binary tree and traversing it breadth first, for some value field, the ascending order holds.

**Breadth Array descending order**

This property tells whether when interpreting an array as a binary tree and traversing it breadth first, for some value field, the descending order holds.

**DYNAMIC ANALYSIS**

Once all the properties are discovered for a set of structures the next step is the analysis so that a proper repOk can be generated. The analysis module also reads the xml file as input for predefined settings. These settings are required because some properties are ambiguous and therefore guidance is required in those cases.

For example consider the maximum balance property. This property is obviously important for example in an AVL tree where the absolute value of the maximum value at any node can be at most one. However when generalizing this property it becomes ambiguous. Because if we receive a set of objects and we find a maximum balance value “k” such that the maximum balances of the set of objects is less than or equal to “k”, then we can conclude the property of the structures having a maximum balance of “k”. However that property may not even be there and we may be just hunting for patterns that are not even there. This problem cannot be solved by giving more samples, because for any number “n” of samples, we can always find an integer “k” such that all the node balances in the “n” objects are less than “k”.

Generally when generating a repOk, this module would only look for properties that are held in all the structures and not some in the set, meaning the properties that have a positivePercentage or negativePercentage of 100 %. For such properties respective codes for repOk would be generated.

However there are exceptions when the statistics module would need the settings to determine whether to generate some repOk properties codes. The obvious examples being the maximum height and balances for which the external opinion is needed. Depending on the settings these ambiguous properties are either deemed important or not. If they are deemed important, then their cut-off values are also defined in the xml file. For example the xml file currently being used denotes the maximum height property as unimportant and the maximum balance property as important with the cut-off value of 1 (to try to detect an AVL tree).

```
<StatisticsSettings>
  <property maximumBalance>
    <relevance> 1 </relevance>
    <cuttOff> 1 </ cuttOff >
  </property>
  <property maximumHeight>
```

```
<relevance> 0 </relevance>
<cuttOff> -1 </ cuttOff >
<property>
<property numberNodes>
<relevance> 0 </relevance>
<cuttOff> -1 </ cuttOff >
<property>
<repOkClass>
TestClass.java
</repOkClass>
</StatisticsSettings>
```

The statistical data of all the properties is stored and maintained. This module can be enhanced to add statistical modeling and the settings for the statistical modeling can be read from the XML file.

Current implementation supports very crude statistical analysis. We assume that if the property is present in all the samples (value of 1) then we generate code for the repOK. But if the presence is  $0 \leq p < 1$ , then we ignore that property.

## **REPOK GENERATION**

The statistics module then passes the results to the repOk generating module that generates codes that check the specific structural properties. There are appropriate checking codes for every one of the properties discussed earlier. The codes can be generated as reflection based or non-reflection based. Right now the implementation only supports the non-reflection based codes. However the reflection based codes are trivial to generate since they are already written and are being used in the detection process. The only thing to do is to export them which will be implemented in the future. The generated code is a function that calls sub-functions each representing a specific property. This code is written in a java file whose class name is specified in the xml settings of the statistics module.

## **AUTOMATED TESTING**

Once the repOk is generated and written to a file, the testing module compiles the file and attempts to check the repOk against the objects given to train. The repOk must pass for all the objects to be successfully generated. This ensures proper generation of repOk codes.

## Chapter 5: Problem with Fewer Samples

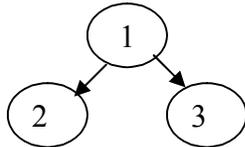


Figure 7: An example of a small tree.

Owing to the nature of the training, the fewer number of modules result in discovering of structural properties that are not even there. Let us take a concrete example for this. Assume that only one object is given for training. Let that object be a binary tree with one parent and two children. The tree has an integer field whose value is 1 in the parent, 2 in the left child and 3 in the right child.

The implementation would attempt to find properties and would discover the following properties true.

1. Binary  $\rightarrow$  true
2. Cyclicity  $\rightarrow$  false
3. maxBalance  $\rightarrow$  1
4. Binary min heap  $\rightarrow$  true
5. Completeness  $\rightarrow$  true
6. Maximum Height  $\rightarrow$  1

It may be noticed, that many of these properties may not be true if given more samples. This sample would pass when tested for the repOk generated but it is perhaps too specific to be of general use.



## Chapter 6: Experiments

Presented next are some of the experiments that were conducted and also the codes that were generated.

### INTENTIONAL NAMING TREE

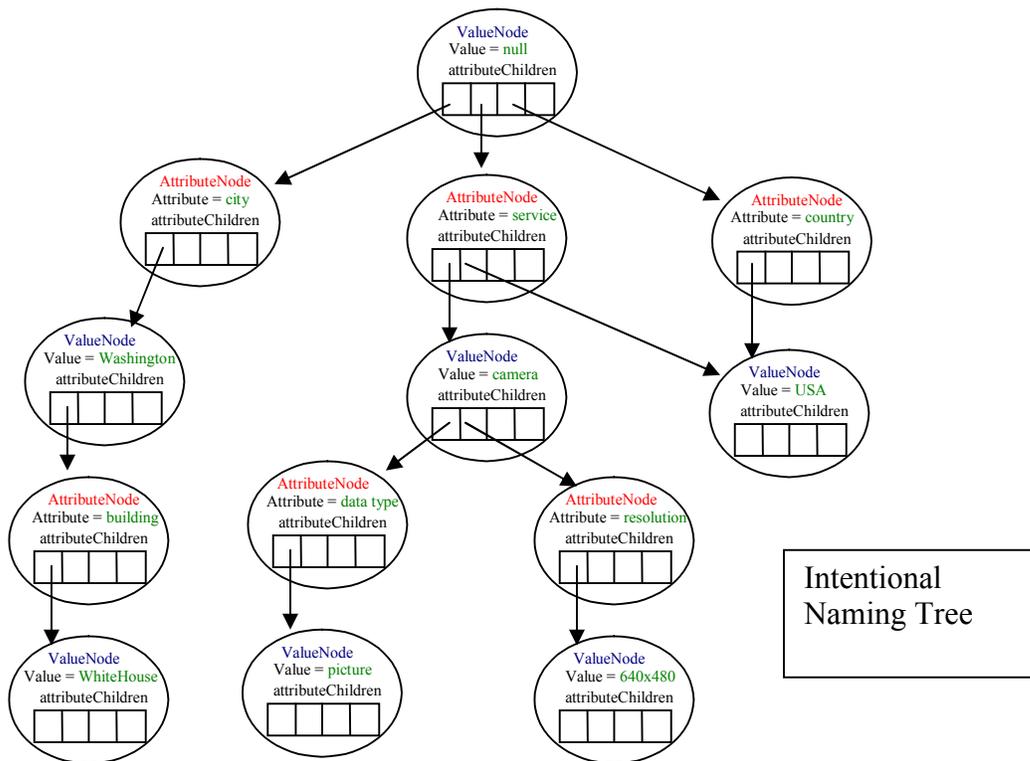


Figure 8: Intentional Naming Tree.

```
public class TestClass
{
    public static boolean repOK(ValueNode head)
    {
        if(isINTRootValueNull(head)==false) return false;
        if(isLastNodeFirstHelper(head,null)==false) return false;
        if(isIntentionalNameTreeCyclic(head,null,new Hashtable())==false);
        return true;
    }
}
```

```

public static boolean isINTRootValueNull(ValueNode valueNode)
{
    if(valueNode.attributeChildren == null) return true;
    else return false;
}

public static boolean isLastNodeFirstHelper(ValueNode valueNode,AttributeNode
attributeNode)
{
    if(valueNode != null)
    {
        if(valueNode.attributeChildren != null)
        {
            for(int i=0;i<valueNode.attributeChildren.size();i++)
            {
                AttributeNode attributeChild = valueNode.attributeChildren.get(i);
                if (isLastNodeFirstHelper(null,attributeChild) == false) return false;
            }
        }
    }
    else if(attributeNode != null)
    {
        if(attributeNode.valueChildren != null)
        {
            if(attributeNode.valueChildren.size() == 0) return false;
            for(int i=0;i<attributeNode.valueChildren.size();i++)
            {
                ValueNode valueChild = attributeNode.valueChildren.get(i);
                if (isLastNodeFirstHelper(valueChild,null)== false) return false;
            }
        }
        else
        {
            return false;
        }
    }
    return true;
}

public static boolean isIntentionalNameTreeCyclic(ValueNode valueNode,AttributeNode
attributeNode, Hashtable hashTable)
{
    if(valueNode != null)
    {
        hashTable.put(valueNode.hashCode(), valueNode);
        if(valueNode.attributeChildren != null)
        {
            for(int i=0;i<valueNode.attributeChildren.size();i++)
            {
                AttributeNode attributeChild = valueNode.attributeChildren.get(i);
                if(hashTable.containsKey(attributeChild.hashCode())) return true;
                else if (isIntentionalNameTreeCyclic(null,attributeChild,hashTable)) return true;
            }
        }
        return false;
    }
    else if(attributeNode != null)
    {
        hashTable.put(attributeNode.hashCode(), attributeNode);
        if(attributeNode.valueChildren != null)
        {
            for(int i=0;i<attributeNode.valueChildren.size();i++)
            {
                ValueNode valueChild = attributeNode.valueChildren.get(i);

```

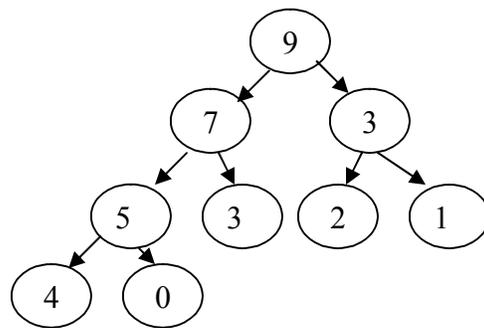
```

    if(hashTable.containsKey(valueChild.hashCode())) return true;
    else if (isIntentionalNameTreeCyclic(valueChild,null,hashTable)) return true;
  }
  return false;
}
return false;
}
}

```

## HEAP ARRAY

9	7	3	5	3	2	1	4	0
---	---	---	---	---	---	---	---	---



Array  
Representing  
a Heap

Figure 9: Array representing a heap

```

import java.util.Hashtable;
import java.util.Vector;
import java.util.ArrayList;

public class TestClass
{
  public static boolean repOK(HeapArray head)
  {
    ArrayList arrayList = head.null;
    if(isArrayLooped(arrayList)==false) return false;
    if(isArrayComplete(arrayList)==false) return false;

```

```

    if(isMaxArrayHeap(arrayList, 1)==false) return false;
    return true;
}

public static boolean isArrayLooped(ArrayList heap )
{
    Hashtable hashtable = new Hashtable();
    for(int i=0;i<heap.size();i++)
    {
        if(heap.get(i) != null)
        {
            if(hashtable.containsKey(heap.get(i).hashCode())) return true;
            hashtable.put( heap.get(i).hashCode(), heap.get(i));
        }
    }
    return false;
}

public static boolean isArrayComplete(ArrayList heap )
{
    for(int i=0;i<heap.size();i++)
    {
        if(heap.get(i) == null)
        {
            while(i<heap.size()) if(heap.get(i++) != null) return false;
        }
    }
    return true;
}

public static boolean isMaxArrayHeap(ArrayList heap, int index)
{
    Integer currentNode = (Integer)heap.get(index - 1);
    if(currentNode == null) return true;
    if((2*index - 1) < heap.size())
    {
        Integer leftChild = (Integer)heap.get(2*index - 1);
        if(leftChild!= null)
        {
            if (currentNode < leftChild ) return false;
            if(isMaxArrayHeap(heap, 2*index) == false) return false;
        }
    }
    if((2*index) < heap.size())
    {
        Integer rightChild = (Integer)heap.get(2*index);
        if(rightChild!= null)
        {
            if (currentNode < rightChild ) return false;
            if(isMaxArrayHeap(heap, 2*index + 1) == false) return false;
        }
    }
    return true;
}
}
}

```

## BINARY SEARCH TREE (COMPLETE)

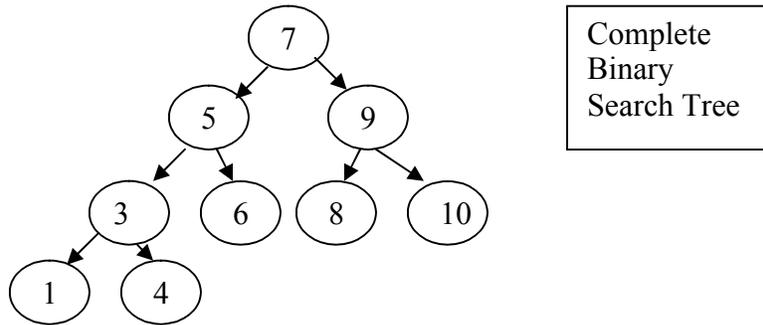


Figure 10: Complete Binary Search Tree

```
import java.util.Hashtable;
import java.util.Vector;
import java.util.ArrayList;

public class TestClass
{

public static boolean repOK(Node head)
{
if(detectCyclesRecursiveFunction(head, new Hashtable())==false) return false;
if(leftBinaryOrder(head)==false) return false;
if(isComplete(head)==false) return false;
if(evaluateMaxBalance(head,1)==false) return false;
return true;
}

public static boolean detectCyclesRecursiveFunction(Node _Node, Hashtable hashTable)
{
hashTable.put(_Node.hashCode(), _Node);
if(_Node.left != null)
{
if(hashTable.containsKey(_Node.left.hashCode())) return true;
else
if(detectCyclesRecursiveFunction(_Node.left, hashTable)) return true;
}
if(_Node.right != null)
{
if(hashTable.containsKey(_Node.right.hashCode())) return true;
else
if(detectCyclesRecursiveFunction(_Node.right, hashTable)) return true;
}
return false;
}
}
```

```

public static boolean detectLeftOrder(Node _Node)
{
    if( _Node.left != null)
    {
        if( _Node.value < _Node.left.value)    return false;
        if(detectMaxHeap( _Node.left) == false) return false;
    }
    if( _Node.right != null)
    {
        if( _Node.value > _Node.right.value)    return false;
        if(detectLeftOrder( _Node.right) == false) return false;
    }
    return true;
}

public static boolean isComplete(Node _Node)
{
    Vector<TreeElement> queue = new Vector<TreeElement>();
    queue.add(new TreeElement( _Node,0));
    TreeElement root = null;
    while(queue.size() != 0)
    {
        root = queue.remove(0);
        if(root.obj != null)
        {
            queue.add(new TreeElement(((Node)(root.obj)).left,root.level + 1));
            queue.add(new TreeElement(((Node)(root.obj)).right,root.level + 1));
        }
        else
        {
            for(int i=0;i<queue.size();i++)
                if(((Node)(queue.get(i).obj)) != null) return false;
        }
    }
    return true;
}

public static boolean evaluateMaxBalance(Node _Node,int maxBalance)
{
    BalanceInformation balanceInformation = findMaxOffsetBalance( _Node);
    if(balanceInformation.maxOffset <= maxBalance) return true;
    else return false;
}

public static BalanceInformation findMaxOffsetBalance(Node _Node)
{
    Vector<BalanceInformation> balances = new Vector<BalanceInformation>();
    if( _Node == null) return new BalanceInformation(0,0);
    if( _Node.left != null)balances.add(findMaxOffsetBalance( _Node.left ));
    if( _Node.right != null)balances.add(findMaxOffsetBalance( _Node.right ));
    return computeNewBalanceInfo(balances);
}
}

```

## SORTED LINKED LIST

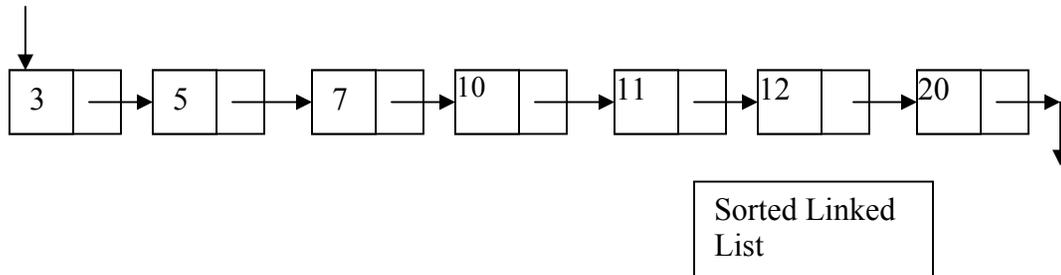


Figure 11: Sorted Linked List

```
import java.util.Hashtable;
import java.util.Vector;
import java.util.ArrayList;

public class TestClass
{

public static boolean repOK(Link head)
{
    if(detectCyclesRecursiveFunction(head, new Hashtable())==false) return false;
    if(unaryMaxOrder(head)==false) return false;
    return true;
}

public static boolean detectCyclesRecursiveFunction(Link _Link, Hashtable hashTable)
{
    hashTable.put( _Link.hashCode(), _Link);
    if( _Link.next != null)
    {
        if(hashTable.containsKey( _Link.next.hashCode())) return true;
        else
            if(detectCyclesRecursiveFunction( _Link.next, hashTable)) return true;
    }
    return false;
}

public static boolean unaryMaxOrder(Link head )
{
    if(head != null)
    {
        int previous = head.value;
        while(true)
        {
            System.out.println(previous);
            head = head.next;
            if(head==null) break;
            if(head.value < previous) return false;
            previous = head.value;
        }
    }
}
```

```

return true;
}
}

```

## MAX HEAP (MAX BALANCE IS 1)

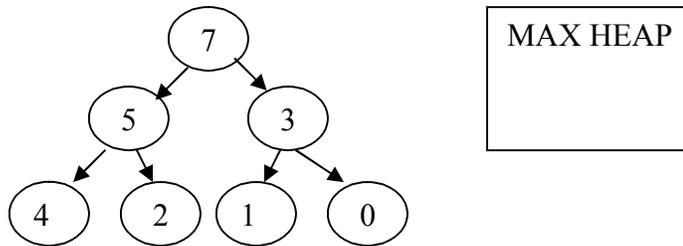


Figure 12: Max Heap with maximum balance of 1

```

import java.util.Hashtable;
import java.util.Vector;
import java.util.ArrayList;

public class TestClass
{

public static boolean repOK(Node head)
{
if(detectCyclesRecursiveFunction(head, new Hashtable())==false) return false;
if(detectMaxHeap(head)==false) return false;
if(MinBFSOrder(head)==false) return false;
if(isComplete(head)==false) return false;
if(evaluateMaxBalance(head,1)==false) return false;
return true;
}

public static boolean detectCyclesRecursiveFunction(Node _Node, Hashtable hashTable)
{
hashTable.put( _Node.hashCode(), _Node);
if( _Node.left != null)
{
if(hashTable.containsKey( _Node.left.hashCode())) return true;
else
if(detectCyclesRecursiveFunction( _Node.left, hashTable)) return true;
}
if( _Node.right != null)
{
if(hashTable.containsKey( _Node.right.hashCode())) return true;
else
if(detectCyclesRecursiveFunction( _Node.right, hashTable)) return true;
}
}
}

```

```

    return false;
}

public static boolean detectMaxHeap(Node _Node)
{
    if( _Node.left != null)
    {
        if( _Node.value < _Node.left.value)    return false;
        if(detectMaxHeap( _Node.left) == false) return false;
    }
    if( _Node.right != null)
    {
        if( _Node.value < _Node.right.value)    return false;
        if(detectMaxHeap( _Node.right) == false) return false;
    }
    return true;
}

public static boolean MinBFSOrder(Node _Node)
{
    Vector<Node> queue = new Vector<Node>();
    queue.add( _Node);
    Node root = null;
    int previous = queue.get(0).value;
    boolean firstIteration = true;
    while(queue.size() != 0)
    {
        root = queue.remove(0);
        if(!firstIteration && root.value > previous) return false;
        previous = root.value;
        firstIteration = false;
        if(root.left != null) queue.add(root.left);
        if(root.right != null) queue.add(root.right);
    }
    return true;
}

public static boolean isComplete(Node _Node)
{
    Vector<TreeElement> queue = new Vector<TreeElement>();
    queue.add(new TreeElement( _Node,0));
    TreeElement root = null;
    while(queue.size() != 0)
    {
        root = queue.remove(0);
        if(root.obj != null)
        {
            queue.add(new TreeElement(((Node)(root.obj)).left,root.level + 1));
            queue.add(new TreeElement(((Node)(root.obj)).right,root.level + 1));
        }
        else
        {
            for(int i=0;i<queue.size();i++)
                if(((Node)(queue.get(i).obj)) != null) return false;
        }
    }
    return true;
}

public static boolean evaluateMaxBalance(Node _Node,int maxBalance)
{
    BalanceInformation balanceInformation = findMaxOffsetBalance( _Node);

```

```
    if(balanceInformation.maxOffset <= maxBalance) return true;
else return false;
}

public static BalanceInformation findMaxOffsetBalance(Node _Node)
{
    Vector<BalanceInformation> balances = new Vector<BalanceInformation>();
    if( _Node == null) return new BalanceInformation(0,0);
    if( _Node.left != null)balances.add(findMaxOffsetBalance( _Node.left ));
    if( _Node.right != null)balances.add(findMaxOffsetBalance( _Node.right ));
    return computeNewBalanceInfo(balances);
}
}
```

## Chapter 7: Discussion

### LIMITATIONS

Constraint generation using a given set of structures has two limitations.

One, the set may not be representative of the class of desired structures. This can be partially solved by giving a large number of examples. Two, not all relevant properties can feasibly be identified, e.g., conjecturing all possible relations among integer fields is infeasible even using simple arithmetic operators. Deryaft's current generation implementation therefore, focuses on structural properties which involve reference fields, which can naturally be viewed as edges in a graph, and simple constraints on primitive data. A lot of primitive field's pattern recognition is endeavored; however a lot of room for improvement is possible.

### OPTIMIZATION OF REPEATED TRAVERSALS

Consider the case for structure enumeration using a constraint solver. It is well known that the performance of constraint solvers, such as propositional satisfiability (SAT) solvers, depends crucially on the formulation of given invariants—the same holds for Korat and the Alloy Analyzer [18]. In fact, repeated traversals which may seemingly be slow, may actually elicit faster generation.

The case for assertion evaluation is usually different: generated code that minimizes the number of traversals is likely to improve the time to check the assertion. The Deryaft's current implementation tries to generate codes that minimize the number of traversals to a certain extent. As soon as some property is found false, the control is immediately returned as false. The order of the properties checking also takes into

account the minimization of traversals. The future implementation will take this more into account and will attempt to discover multiple properties in fewer traversals.

#### **INTEGRATION WITH OTHER SOFTWARE ANALYSIS FRAMEWORK.**

We have given an example of how Korat can be used for input enumeration using invariants generated by Deryaft. We plan to fully integrate Deryaft's algorithm with various existing frameworks.

#### **STATIC ANALYSIS FOR OPTIMIZING GENERATION**

While in the presence of a partial implementation we may not require the user to provide a set of structures, we can use the implementation in a different way as well: a static analysis of the code, say the method that adds a node to a heap, can help formulate the likely invariants more accurately.

## Chapter 8: Related Work

Dynamic analyses Our work is inspired by the Daikon invariant detection engine [7], which pioneered the notion of dynamically detecting likely program invariants in the late 90s and has since been adapted by various other frameworks [12, 11]. Deryaft differs from Daikon in three key aspects. First, the model of data structures in Daikon uses arrays to represent object fields. While this representation allows detecting invariants of some data structures, it makes it awkward as to how to detect invariants that involve intricate global properties, such as relating lengths of paths. Deryaft’s view of the heap as an edge-labeled graph and focus on generic graph properties enables it to directly capture a whole range of structurally complex data. Second, Deryaft employs specific heuristics that optimize generation of invariants for data structures, e.g., the distinction between core and derived fields allows it to preemptively disallow hypothesizing relations among certain fields. We believe this distinction, if adopted, can optimize Daikon’s analysis too. Third, Deryaft generates invariants in Java, which can directly be plugged into a variety of tools, such as the Korat testing framework [4] and the Juzi [15] repair framework.

### STATIC ANALYSIS

Researchers have explored invariant generation using static analyses for over three decades. There is a wide body of research in the context of generating loop invariants [9, 6, 23, 21] using recurrence equations, abstract interpretation with widening, matrix theory for Petri nets, constraint-based techniques etc. Most of these analyses are limited to relations between primitive variables.

Shape analyses [10, 20, 19, 2] can handle structural constraints using abstract heap representations, predicate abstraction etc. However, shape analyses typically do not consider rich properties of data values in structures and mostly abstract away from the data. Moreover, none of the existing shape analyses can feasibly check or detect rich structural invariants, such as height-balance for binary search trees, which involve complex properties that relate paths.

### **COMBINED DYNAMIC/STATIC ANALYSIS**

Some recent approaches combine static and dynamic analyses for inferring API level specifications [22, 3].

Invariant generation has also been used in the context of model checkers to explain the absence of counterexamples, while focusing on integer variables [1].

## Chapter 9: Conclusion

An implementation is provided for the Deryaft, a novel invariant generation algorithm. Deryaft analyzes the key characteristics of the given structures to formulate local and global properties that the structures have in common. A key idea in Deryaft is to view the program heap as an edge-labeled graph, and hence to focus on properties of graphs, including reachability. Deryaft partitions the set of edges into core and derived sets and hypothesizes different classes of properties for each set, thereby minimizing the number of hypotheses it needs to validate.

Our implementation also generates a Java predicate that represents the properties of given structures, i.e., it generates a method that takes an input structure, traverses it, and returns true if and only if the input satisfies the properties. Even though our implementation does not require an implementation of any methods that manipulate the given structures, in the presence of such an implementation, it can generate the invariants without a priori requiring a given set of structures. The invariants generated by Deryaft enable automation of various software analyses.

## References

- [1] M. Vaziri and G. Holzmann. Automatic detection of invariants in spin. In Proc. SPIN Workshop on Software Model Checking, November 1998.
- [2] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. 2005. Shape analysis by predicate abstraction: In Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation, Paris, France.
- [3] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object oriented component interfaces. In Proc. International Symposium on Software Testing and Analysis (ISSTA), July 2002.
- [4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. July 2002. Korat: Automated testing based on Java predicates: In Proc. International Symposium on Software Testing and Analysis (ISSTA).
- [5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. Introduction to Algorithms: The MIT Press, Cambridge, MA.
- [6] P. Cousot and N. Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program: In Proc. 5th Annual ACM Symposium on the Principles of Programming Languages (POPL), Tucson, Arizona.
- [7] Michael D. Ernst. August 2000. Dynamically Discovering Likely Program Invariants: PhD thesis, University of Washington Department of Computer Science and Engineering.
- [8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java: In Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation.
- [9] Steven M. German and Ben Wegbreit. 1975. A synthesizer of inductive assertions: IEEE Trans. Software Eng., 1(1).
- [10] Rakesh Ghiya and Laurie J. Hendren. 1996. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C: In POPL '96: Proceedings of the 23rd ACM SIGPLAN SIGACT symposium on Principles of programming languages.

- [11] Neelam Gupta and Zachary V. Heidepriem. October 2003. A new structural coverage criterion for dynamic detection of program invariants: In Proc. 18th Conference on Automated Software Engineering (ASE), San Diego, CA.
- [12] Sudheendra Hangal and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection: In ICSE '02: Proceedings of the 24th International Conference on Software Engineering.
- [13] Daniel Jackson. 2006. Software Abstractions: Logic, Language and Analysis: The MIT Press, Cambridge, MA.
- [14] Daniel Jackson and Alan Fekete. October 2001. Lightweight analysis of object interactions: In Proc. Fourth International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan.
- [15] Sarfraz Khurshid, Iv'an Garc'ia, and Yuk Lai Suen. 2005. Repairing structurally complex data: In Proc. 12th SPIN Workshop on Software Model Checking, San Francisco, CA.
- [16] Sarfraz Khurshid, Muhammad ZubairMalik, and Engin Uzuncaova. 2006. An automated approach for writing Alloy specifications using instances. In 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos, Cyprus.
- [17] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. June 1998. Preliminary design of JML: A behavioral interface specification language for Java: Technical Report TR 98-06i, Department of Computer Science, Iowa State University.
- [18] Darko Marinov, Sarfraz Khurshid, Suhabe Bugrara, Lintao Zhang, and Martin Rinard. 2005. Optimizations for compiling declarative models into boolean formulas: In 8th Intl. Conference on Theory and Applications of Satisfiability Testing (SAT).
- [19] Anders Moeller and Michael I. Schwartzbach. June 2001. The pointer assertion logic engine: In Proc. SIGPLAN Conference on Programming Languages Design and Implementation, Snowbird, UT.
- [20] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. January 1998. Solving shape-analysis problems in languages with destructive updating. ACM Transactions on Programming Languages and Systems (TOPLAS).
- [21] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using groebner bases: In POPL '04: Proceedings of the 31st ACM SIGPLANSIGACT symposium on Principles of programming languages.

- [22] Mana Taghdiri. 2004. Inferring specifications to detect errors in code: In Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Washington, DC.
- [23] Ashish Tiwari, Harald Rue, Hassen Saidi, and Natarajan Shankar. 2001. A technique for invariant generation: In Proc. 7th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), London, UK.

## Appendix

### INSTRUCTIONS TO RUN CODE

The user can download binary distribution, [Deryaft-0.9.zip](#) that required java 1.5. To test if the jar works without using the API the following command line options can be used.

-g <nothing> → Generates a list of data structures and serializes them in a file

-d <filename> → (**required**) name of the file that is read. Then detection is performed on it

### API

API for Deryaft is very straight forward and is discussed below.

#### *Class*

After importing the jar file the only class that has the interfacing API is the DetectionEngine.java.

#### *Methods*

The relevant methods to be used are as follows.

*DetectionEngine (Collection collection, String backBoneXML, String KnownRulesXML, String knownModelsXML, propertiesXML)*

- **Collection** is the collection of objects that need to be detected.
- **backBoneXML** is the path to the XML file in which the user can specify the primitive fields and the self-referenced fields that need to be investigated. Optionally this file can be empty, in which case Deryaft deems all the fields important.
- **knownRulesXML** is the path to the XML file that has the rules to be used when detecting properties. A sample XML file is given later.
- **knownModelsXML** is the path to the XML file that has the commonly known data-structures and their properties.
- **propertiesXML** is the path to the XML file that tells what detection rules are enabled or disabled.

*getProperties ()*

Returns an object of type DSProperties that has all the properties of type DSProperty listed as a vector in it. The fields in DSProperty have been discussed before.

*writePropertiesToFile (String propertiesOutputFile)*

Writes an XML file that has all the detected properties. If the file does not already exist, it is generated.

### ***getRepOK ()***

Returns the repOK method as a string that may also contain helper functions.

### ***writeRepOKToFile (String repOKClass, String repOKFilePath )***

Writes a class **repOKClass** that is written at the specified path.

## **CODE WEBSITE**

The link to website is <https://webspace.utexas.edu/ap3649/index.htm>.

The website has a reference manual and tutorials to get a user started. The jar file can be downloaded from the download section of the website. Also the report itself and the relevant publications can also be obtained.



**Deryaft - Home Page**

[Home](#)  
[Manual](#)  
[Tutorial](#)  
[Downloads](#)  
[Screenshots](#)  
[Publications](#)  
[About us](#)

### Welcome to Deryaft Home Page

Deryaft is a tool that discovers likely invariants of your structurally complex java programs. Structurally complex means that the inputs are structural (e.g., represented with linked data structures) and must satisfy complex constraints that relate parts of the structure e.g., Acyclicity for linked data structures).

Deryaft outputs an imperative predicate that specifies the desired structural constraints after observing dynamic execution of representative examples of the structure.

## SOME CONFIGURATION XMLFILES (SAMPLES)

### KnownModels

```
<KnownModels>

<Model Tree>
<property acyclicity> 1 </property>
<property binary> 1 </property>
</Model>

<Model Linked List>
<property unary> 1 </property>
<property acyclicity> 1 </property>
</Model>

<Model Max Linked List>
<property unary> 1 </property>
<property acyclicity> 1 </property>
<property maxOrder> 1 </property>
</Model>

<Model Min Linked List>
<property unary> 1 </property>
<property minOrder> 1 </property>
</Model>

<Model Binary Search Tree>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Order> 1 </property>
</Model>

<Model MinHeap>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Completeness> 1 </property>
<property MaxProperty> 1 </property>
</Model>

<Model MaxHeap>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Completeness> 1 </property>
<property MinProperty> 1 </property>
</Model>
```

```
<Model AVL Tree>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Balance> 1 </property>
</Model>

</KnownModels>
```

## KnownRules

```
<KnownModels>

<Model Tree>
<property acyclicity> 1 </property>
<property binary> 1 </property>
</Model>

<Model Linked List>
<property unary> 1 </property>
<property acyclicity> 1 </property>
</Model>

<Model Max Linked List>
<property unary> 1 </property>
<property acyclicity> 1 </property>
<property maxOrder> 1 </property>
</Model>

<Model Min Linked List>
<property unary> 1 </property>
<property minOrder> 1 </property>
</Model>

<Model Binary Search Tree>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Order> 1 </property>
</Model>

<Model MinHeap>
<property binary> 1 </property>
<property acyclicity> 1 </property>
<property Completeness> 1 </property>
<property MaxProperty> 1 </property>
</Model>
```

```

<Model MaxHeap> <KnownRules>

  <Rule unary>
    <Positive>
      <posistiveRule>acyclicity</posistiveRule>
      <posistiveRule>maxOrder</posistiveRule>
      <posistiveRule>minOrder</posistiveRule>
    </Positive>
  </Rule>

  <Rule linear>
    <Positive>
      <posistiveRule>arrayMaxBreadth</posistiveRule>
      <posistiveRule>arrayCycliclity</posistiveRule>

      <posistiveRule>arrayCompleteness</posistiveRule>
      <posistiveRule>arrayMaxHeap</posistiveRule>
      <posistiveRule>arrayBSTLeft</posistiveRule>
    </Positive>
  </Rule>

  <Rule intentionalName>
    <Positive>
      <posistiveRule>rootNull</posistiveRule>
      <posistiveRule>isFirstLast</posistiveRule>

      <posistiveRule>intentionalCyclicty</posistiveRule>
    </Positive>
  </Rule>

  <Rule acyclicity>
    <Negative>
      <negativeRule>maxOrder</negativeRule>
      <negativeRule>minOrder</negativeRule>
      <negativeRule>binary</negativeRule>
      <negativeRule>doubleList</negativeRule>
    </Negative>
  </Rule>

  <Rule binary>
    <Negative>
      <negativeRule>unary</negativeRule>
      <negativeRule>maxOrder</negativeRule>
      <negativeRule>minOrder</negativeRule>
    </Negative>

    <Positive>

```

```

<posistiveRule>leftBinaryOrder</posistiveRule>
<posistiveRule>rightBinaryOrder</posistiveRule>
<posistiveRule>maxHeapProperty</posistiveRule>
<posistiveRule>minHeapProperty</posistiveRule>
<posistiveRule>maxBFSOrder</posistiveRule>
<posistiveRule>minBFSOrder</posistiveRule>
<posistiveRule>completeness</posistiveRule>
<posistiveRule>maxBalance</posistiveRule>
<posistiveRule>maxHeight</posistiveRule>

</Positive>

</Rule>

</KnownRules>

```

## BackBone

```

<BackBone>
<ValueField>
value
<ValueField>
<RefernceField>
left
</RefernceField>
<RefernceField>
right
</RefernceField>

</BackBone>

```

## DynamicAnalysis

```

<DynamicAnalysis>

<Property acyclicity>
enabled
</Property>
<Property unary>
enabled
</Property>
<Property binary>

```

```

enabled
</Property>
<Property N-ary    >
enabled
</Property>
<Property linear>
enabled
</Property>
<Property intentionalName>
enabled
</Property>
<Property rootNull    >
enabled
</Property>
<Property isFirstLast>
enabled
</Property>
<Property intentionalCyclicty>
enabled
</Property>
<Property UnaryMaxOrder>
enabled
</Property>
<Property UnaryMinOrder>
enabled
</Property>
<Property leftBinaryOrder>
enabled
</Property>
<Property rightBinaryOrder>
enabled
</Property>
<Property maxHeapProperty>
enabled
</Property>
<Property minHeapProperty>
enabled
</Property>
<Property maxBFSOrder>
enabled
</Property>
<Property minBFSOrder>
enabled
</Property>
<Property completeness>
enabled
</Property>
<Property maxBalance>
2
</Property>
<Property numNodes>
10
</Property>

```

```
<Property maxHeight>
10
</Property>
<Property RootFake>
enabled
</Property>
<Property arrayMaxBreadth>
enabled
</Property>
<Property arrayCyclicity>
enabled
</Property>
<Property arrayCompleteness>
enabled
</Property>
<Property arrayMaxHeap>
enabled
</Property>
<Property arrayBSTLeft>
enabled
</Property>
```

```
</DynamicAnalysis>
```

## **Vita**

Aman Pervaiz was born on January 14<sup>rd</sup> 1981 in Lahore, Pakistan, to Pervaiz Akhtar and Roohi Pervaiz. He went to cathedral high school from 1<sup>st</sup> to 9<sup>th</sup>. He obtained his ordinary level (O-level) degree from Lahore Lyceum and his advanced degree (A-level) from Beaconhouse garden town branch. He completed his BS in computer engineering from LUMS Pakistan in 2005. During his stay at LUMS he worked as a research assistant and a teaching assistant for several courses. He joined UT Austin to obtain a MSE degree in computer engineering after graduating from LUMS. Before this report he has two research papers to his credit.

Permanent address: House No. 36, St No. 6, Lajpat Road, Shahdra Station  
Lahore Pakistan.

This report was typed by Aman Pervaiz.