Design of Deryaft: A Novel Framework for Generating Representation Invariants of Structurally Complex Data

by

Muhammad Zubair Malik M.Sc. Computer Sc., BS Computer System Engineering

Thesis

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

Masters of Science

The University of Texas at Austin

November 2007

Design of Deryaft: A Novel Framework for Generating Representation Invariants of Structurally Complex Data

Dedication

To Software Verification and Validation Research Group

Acknowledgements

I would like to thank my adviser Dr. Khurshid for his original idea which has resulted in this thesis. Without his idea, encouragement and guidance this work would not have been possible. I feel extremely lucky to have work with him.

I would like to thank my friend and colleague Aman Pervaiz for his support in implementing this work for Java. Also I would like to thank Engin Uzuncaova for helping in Alloy implementation of my approach.

November 18, 2007

Abstract

Design of Deryaft: A Novel Framework for Generating Representation Invariants of Structurally Complex Data

Muhammad Zubair Malik, MS

The University of Texas at Austin, 2007

Supervisor: Sarfraz Khurshid

This dissertation presents a novel approach for generating likely structural invariants of complex data structures. Generating likely invariants using dynamic analyses is becoming an increasingly effective technique in software checking methodologies. Given a small set of concrete structures, our approach analyzes their key characteristics to formulate local and global properties that the structures exhibit. For effective formulation of structural invariants, this approach focuses on graph properties, including reachability, and views the program heap as an edge-labeled graph.

The Deryaft Tool implements this approach for Java. Deryaft outputs a Java predicate that represents the invariants; the predicate takes an input structure and returns true if and only if it satisfies the invariants. The invariants generated by Deryaft directly enable automation of various existing frameworks, such as the Korat test generation framework and the Juzi data structure repair framework, which otherwise require the user to provide the invariants. Experimental results with the Deryaft prototype show that it feasibly generates invariants for a range of subject structures, including libraries as well as a stand-alone application. The focus of this dissertation is design of Deryaft but we also provide details of aDeryaft which specializes our algorithm for Alloy constraint generation and facilitates frameworks such as TestEra for test generation.

Table of Contents

List of Figures

Chapter 1: Introduction

Software verification and validation are standard ways to provide assurance of software quality. The verification depends on specification of the software which are provided manually. This dissertation introduces a novel algorithm for automatically generating these specifications for programs. We focus on software components that have structurally complex inputs: the inputs are structural (e.g., represented with linked data structures) and must satisfy complex properties that relate parts of the structure (e.g., invariants for linked data structures). Our main contribution is a technique for the automatic generation of specifications for structurally complex inputs which are unique to structures understudy. We assume that the user provides a representative set of structure instances that is sufficiently discriminative from examples of other structures.

WRITING INVARIANTS IS BURDENSOME

Specification-based testing is widely recognized as an effective methodology for increasing the reliability of software, and recent approaches, such as TestEra [26] and Korat [4], have shown how to automate it. However, writing specifications or program invariants correctly itself is tricky, error-prone and time consuming. This is especially the case for programs that use advanced constructs of modern programming languages and manipulate structurally complex data. Moreover, for legacy code, it is usually not pragmatic to provide specifications.

The invariants are critical in verifying the correctness of a program. If a developer is not accurate in codifying the problem it is likely that his misunderstanding will reflect in the invariants he identifies. This is also possible that developer was accurate in codifying the problem but misidentifies the invariants and further still identifies correct invariants but miscodes them. All of these possible interactions and their ramifications make the writing of invariants a burdensome process. And provides a natural motivation to try and automate the invariant writing process.

DYNAMIC ANALYSIS FOR INFERRING INVARIANTS

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.

We present a novel framework for generating representation invariants of structurally complex data given a (small) set of structures. The Deryaft tool implements this framework for Java programming language. 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].

Given a set of structures, Deryaft inspects them to formulate a set of hypotheses on the underlying structural as well as data constraints that are likely to hold. Next, it checks which hypotheses actually hold for the structures. Finally, it translates the valid hypotheses into a Java predicate that represents the structural invariants of the given structures. The predicate takes an input structure, traverses it, and returns true if and only if the input satisfies the invariants.

Deryaft views the program heap as an edge-labeled graph whose nodes represent objects and whose edges represent fields [14] and focuses on generating graphs properties, which include reachability. To make invariant generation feasible, Deryaft incorporates a number of heuristics, which allow it to hone on relevant properties. For non-linear structures, Deryaft also conjectures properties about lengths of paths from the root, and completeness of acyclic structures. Thus, it conjectures local as well as global properties. In addition to properties of structure, Deryaft also conjectures properties among data values in the structures. For example, it conjectures whether the key in a node is larger than all the keys in the node's left sub-tree, or whether the value of a field represents a function of the number of nodes in the structure.

The undecidability of the problem that Deryaft addresses necessitates that its constraint generation, in general, cannot be sound and complete [7]. The generated constraints are sound with respect to the set of given structures. Of course, unseen structures may or may not satisfy them. Deryaft's generation is not complete: it may not generate all possible constraints that hold for the given set of structures. We provide a simple API for allowing users to systematically extend the pool of invariants Deryaft hypothesizes.

Even though Deryaft requires a small set of structures to be given, if a method that constructs structures is given instead, Deryaft can use the method in place of the structures. For example, consider a method that adds an element to a binary search tree. Exhaustive enumeration of small sequences of additions of say up to three arbitrarily selected elements, starting with an empty tree, automatically provides a set of valid binary search trees (assuming the implementation of add is correct) that Deryaft requires.

Deryaft's approach has the potential to change how programmers work. Test-first programming [3] already advocates writing tests before implementations. Having written a small test suite, the user can rely on Deryaft to generate an invariant that represents a whole class of valid structures; Korat [4] can use this invariant to enumerate a high quality test suite; Juzi can use the same invariant to provide data structure repair. Thus, Deryaft facilitates both systematic testing at compile-time as well as error recovery at runtime.

CONTRIBUTIONS

This thesis report makes the following contributions:

- It gives a novel algorithm for generating representation invariants of a structurally complex data from a given small set of structures
- It introduces rooted edge-labeled graph view of the program heap for generating Java predicates
- It introduces the partition of relations into core and derived sets for generating specifications in Java
- It presents Deryaft which generates invariants as Java predicates that can directly be used in other applications e.g., for test generation and error recovery
- It describes aDeryaft which generates constraints for Alloy
- It shows with experiments the feasibility of generating invariants for a variety of data structures
- It presents a fully automated testing framework

Chapter 2: Examples

This chapter presents two examples that illustrate how programmers can use Deryaft to generate invariants for their programs. The first example uses a linked list data structure to illustrate testing a method that manipulates a linked data structure. The second example uses a heap data structure to illustrate testing a method that manipulates an array-based data structure. The term heap here refers to the data structure, also known as priority queues, and not to the dynamically-allocated memory.

Linked List

We first present an example to illustrate Deryaft's generation of the representation invariant of acyclic singly-linked lists. Consider the following class declaration:

```java
public class SinglyLinkedList {
      private Node header; // first list node
      private int size; // number of nodes in the list
      private static class Node {
            int elem;
            Node next;
      }

}
```

A list has a header node, which represents the first node of the list, and caches the number of nodes it contains in the size field. Each node has an integer element elem and a next field, which points to the next node in the list.

Assume that the class SinglyLinkedList implements acyclic lists, i.e., there are no directed cycles in the graph reachable from the header node of any valid list. Figure 1 shows a set of three lists, one each with zero, one and three nodes, which are all acyclic.

Given a set of these lists, i.e., a reference to a HashSet containing the three list objects shown, Deryaft generates the representation invariant shown in Figure 2.

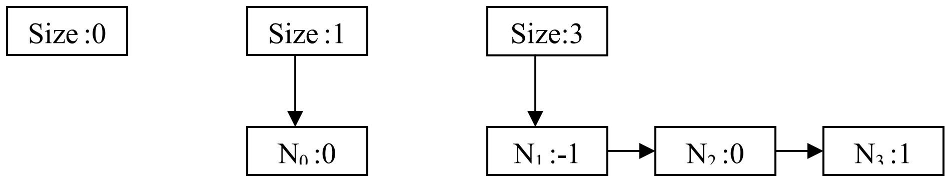


Figure 1: Three acyclic singly-linked lists, one each containing zero, one, and three nodes, as indicated by the value of the size field. Small hollow squares represent the list objects. The labeled arrows represent the fields header and next. N0, N1, N2, and N3 represent the identities of node objects. The nodes also contain the integer elements, which for the given three lists range over the set {-1, 0, 1}.

The method repOk performs two traversals over the structure represented by this. First, repOk checks that the structure is acyclic along the next field. Second, it checks that the structure has the correct value for the size field. The acyclicity checks that there is a unique path from header to every reachable node, while the check for size simply computes the total number of reachable nodes and verifies that number.

To illustrate how Deryaft automates existing analyses, consider enumeration of test inputs using the Korat [4] framework, which requires the user to provide a repOk and a bound on input size. To illustrate, given the repOk generated by Deryaft, and a bound of 5 nodes with integer elements ranging from1 to 5, Korat takes 1.9 seconds to generate all 3905 nonisomorphic lists with up to 5 nodes. Using the inputs that Korat enumerates, any given implementation of the list methods can be tested systematically.

```
public boolean repOk() {
      if (!acyclicCore(header)) return false;
      if (!sizeOk(size, header)) return false;
      return true;
}

private boolean acyclicCore(Node n) {
      Set<Node> visited = new HashSet<Node>();
      LinkedList<Node> worklist = new LinkedList<Node>();
      if (n != null) {
            worklist.addFirst(n);
            visited.add(n);
      }
      while (!worklist.isEmpty()) {
            Node current = worklist.removeFirst();
            if (current.next != null) {
                  if (!visited.add(current.next)) {
            //re-visiting a previously visited node
                        return false;
                  }
                  worklist.addFirst(current.next);
            }
      }
      return true;
}

private boolean sizeOk(int s, Node n) {
      Set<Node> visited = new HashSet<Node>();
      LinkedList<Node> worklist = new LinkedList<Node>();
      if (n != null) {
            worklist.addFirst(n);
            visited.add(n);
      }
      while (!worklist.isEmpty()) {
            Node current = worklist.removeFirst();
            if (current.next != null) {
                  if (visited.add(current.next)) {
                        worklist.addFirst(current.next);
                  }
            }
      }
      return (s == visited.size());
}
```

Figure 2: Invariant generated by Deryaft. The method repOk represents the structural invariants of the given set of list structures. The method acyclicCore uses a standard work-list based graph traversal algorithm to visit all nodes reachable from n via the field next and returns true if and only if the structure is free of cycles. The method sizeOk performs a similar traversal to checks that the number of nodes reachable from n equals s.

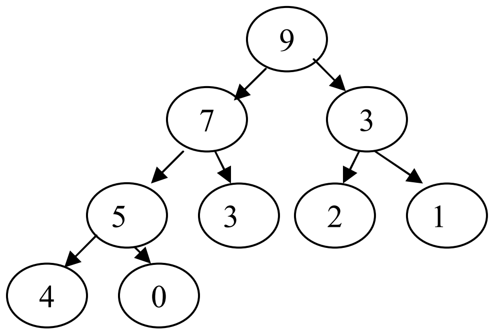
Notice that neither the generation of repOk nor the enumeration of test inputs required an a priori implementation of any method of the class SinglyLinkedList. Indeed once such methods are written, they can be checked using a variety of frameworks that make use of the representation invariants, which traditionally have been provided by the user but can now be generated using Deryaft.

In case a partial implementation of the class SinglyLinkedList is available, Deryaft is able to utilize that. For example, assume that we have an implementation of the instance method add:

```
void add(int i) { ... }
```

which adds the given integer i at the head of the list this. Given add, it is trivial to automatically synthesize a driver program that repeatedly invokes add to enumerate all lists within a small bound, e.g., with up to 3 nodes, using the integer elements {-1, 0, 1}. These lists then serve as the set of input structures for Deryaft.

```java
public class HeapArray {
      private int[] data;
      private int size;
}
```



9	7	3	5	3	2	1	4	0

Figure 3: Displays the declaration of a heap array and an example heap structure with its physical array and abstract diagrammatic representation.

Heap Array

Heap arrays provide an array-based implementation of the binary heap data structure that is also commonly known as a priority queue. This example illustrates how Deryaft can generate invariants for array based implementations. A heap has a capacity that is the length of the underlying array and a size that is the number of elements currently in the heap. For a heap element at index i, its left child is at index $2\times i + 1$ and the right child is at index $2\times i + 2$. The figure below gives the declaration and an example array based representation of a max-heap.

The Integrity constraints are size ≤ capacity and the heap satisfies the max-heap (respectively min-heap) property: an element is larger (respectively smaller) than both its children. If examples of heap structures are constructed to provide Deryaft with a representative set of heap structure in array it is able to hypothesis and generate invariants capturing integrity constraints of heap. The output of Deryaft analysis illustrated by the figure below, Notice agian that neither the generation of repOk nor the enumeration of test inputs required an a priori implementation of any method of the class HeapArray.

```
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;
}
```

Figure 4: Invariants generated by Deryaft for Array based Heap implementation.

Chapter 3: Related Work

There is a large body of research on detecting invariants in a program. We are not aware, however, of any work prior to Deryaft that allowed detection of structural invariants of complex data structures. In this chapter, we discuss how Deryaft relates to other projects on invariant generation, static and dynamic analysis (and in particular Daikon).

DYNAMIC ANALYSES

Comparison with Daikon

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.

We have conducted some intial experiments to compare the output of Daikon with Deryaft. Daikon does not seem to generate rich data structure invariants for the subjects we have presented in this paper. For example, for the SinglyLinkedList class (Section 2), using the lists shown in Figure 1, Daikon generates the following class invariant

for SinglyLinkedList:

```
/*@ invariant this.header.next.next != null; */
/*@ invariant this.header.next.elem == -1; */
/*@ invariant this.header.elem == 0 || this.header.elem == 1; */
/*@ invariant this.size == 0; */
```

and the following for Node:

```
/*@ invariant this.next == null; */
/*@ invariant this.elem == -1 || this.elem == 0 || this.elem == 1; */
```

Even using a larger test suite with 100 randomly generated lists using the API methods of SinglyLinkedList, we were not able to generate more precise invariants with Daikon. We believe that Daikon experts can set its parameters so that it generates a richer class of invariants.

Comparison with aDeryaft:

In previous work [16], we developed aDeryaft, a tool for assisting Alloy [13] users build their Alloy specifications. aDeryaft generates first-order logic formulas that represent structural invariants of a given set of Alloy instances. This paper extends both the design and implementation of aDeryaft by (1) supporting all of Java data-types (including arrays), which significantly differ from Alloy's relational basis, (2) extending the class of invariants supported and (3) evaluating using a wide class of subject structures, including those from a stand-alone application.

STATIC ANALYSES

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 interpretationwith widening, matrix theory for Petri nets, constraint-based techniques etc. Most of these analyses are limited to relations between primitive variables.

SHAPE ANALYSES

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 ANALYSES

Some recent approaches combine static and dynamic analyses for inferring API level specifications [22, 25].

MODEL CHECKERS

Invariant generation has also been used in the context of model checkers to explain the absence of counterexamples, while focusing on integer variables [24].

Chapter 4: JAVA Predicates from Instances

This chapter formally presents generation of imperative predicates from instances and gives the details of our algorithm with reference to its Java implementation: Deryaft. An imperative predicate is a piece of code that takes an input, which we call structure, and returns a Boolean. The most general predicates are totalities, one that always returns true accepting all structures or one that always returns false rejecting all structures. A good predicate only returns true if the structure is of desired type and rejects all other structures. This chapter presents in detail how Deryaft generates the most specific likely imperative predicates for the instances of the structures presented to it as training examples. Here we give theoretical background and details of our algorithm. We first describe an abstract view of the program heap. Next, we define core and derived sets. Then, we characterize the invariants that Deryaft can generate. Finally, we describe how its algorithm works and illustrate it.

Program Heap as an Edge-Labeled Graph

We take a relational view [14] of the program heap: we view the heap of a Java program as an edge-labeled directed graph whose nodes represent objects and whose edges represent fields. The presence of an edge labeled f from node o to v says that the f field of the object o points to the object v (or is null) or has the primitive value v. Mathematically, we treat this graph as a set (the set of nodes) and a collection of relations, one for each field. We partition the set of nodes according to the declared classes and partition the set of edges according to the declared fields; we represent null as a special node. A particular program state is represented by an assignment of values to these sets and relations. Since we model the heap at the concrete level, there is a straightforward isomorphism between program states and assignments of values to the underlying sets and relations.

To illustrate, recall the class declaration for SinglyLinkedList from chapter 2. The basic model of heap for this example consists of three sets, each corresponding to a declared class or primitive type:

```
SinglyLinkedList
Node
Int
```

and four relations, each corresponding to a declared field:

```
header: SinglyLinkedList x Node
size: SinglyLinkedList x int
elem: Node x int
next: Node x Node
```

The “size: 3” list from Figure 1 can be represented using the following assignment of values to these sets and relations:

```
SinglyLinkedList = { L0 }
Node = { N1, N2, N3 }
int = { -1, 0, 1 }

header = { <L0, N0> }
size = { <L0, 3> }
elem = { <N1, 1>, <N2, -1>, <N3, 0> }
next = { <N1, N2>, <N2, N3>, <N3, null> }
```

Deryaft assumes (without loss of generality) that each structure in the given set has a unique root pointer. Thus, the abstract view of a structure is a rooted edge-labeled directed graph, and Deryaft focuses on generating properties of such graphs, including properties that involve reachability, e.g., acyclicity.

Core and Derived Fields

Deryaft partitions the set of reference fields declared in the classes of objects in the given structures into two sets: core and derived. For a given set, S, of structures, let F be the set of all reference fields.

```
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 5: 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, a 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.

Properties of Interest

We consider global as well as local properties of rooted edge-labeled directed graphs, which are likely representatives of structurally complex data. The properties are divided into various categories as follows.

Global: reachability

Reachability properties include the shape of the structure reachable from root along some set of reference fields. The shapes can be acyclic (i.e., there is a unique path from the root to every node), directed-acyclic (i.e., there are no directed cycles in the graph), circular (i.e., all the graph nodes of a certain type are linked in a cycle), or arbitrary. Note that any acyclic graph is also directed-acyclic.

To illustrate, the property `acyclic(header, {next})`, i.e, the structure reachable from header along the field next is acyclic, holds for all the lists shown in Figure 1.

Global: primitive fields

In reasoning about graphs, the notion of a cardinality of a set of nodes occurs naturally. We consider properties relating values of integer fields and cardinalities of sets of reachable objects. For example, the property `equals(size, reachable(header, next).cardinality())` checks whether size is the cardinality of the set of objects reachable from header following zero or more traversals of next.

Global: path lengths

For non-linear structures, such as trees, we consider properties that relate lengths of different paths from root. For example, the property balanced represents that no simple path from the root differs in length from another simple path by more than one. For binary trees, this property represents a *height-balanced tree*.

Local: reference fields

In edge-labeled graphs that are not acyclic (along the set of all fields), local properties that relate different types of edges are likely. To illustrate, consider a graph where if an edge connects a node n of type N to a node m of type M, there is a corresponding edge that connectsmto n.We term such properties two-cycles. For a doubly-linked list, next and previous form a two-cycle.

Another local property on reference fields is whether a particular node always has an edge of a particular type from it to null.

Local: primitive fields

Another category of local properties pertains to primitive values. For example, in a binary tree, the value in a node might be greater than the values in the node's children. We consider local properties that relate a node's value to it's successors along reference fields.

Algorithm

Given a set of structures, Deryaft traverses the structures to formulate a set of hypotheses. Next, it checks which of the hypotheses actually hold for the given structures. Finally, it translates the valid hypotheses into a Java predicate that represents the structural invariants of the 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 invariants.

To make invariant generation feasible, a key heuristic that Deryaft incorporates to focus on relevant properties is: hypothesize properties about reachability, such as acyclicity or circularity, only for the fields in the core set; and hypothesize local properties that relate derived fields and core fields, e.g., whether a derived field forms two-cycles with some core fields.

Figure 6 presents theDeryaft algorithmusing Java-like pseudo-code.Tominimize the number of properties that are checked on the given structures, the checkProperties does not check a property p if a property q that contradicts p is already known to be true, e.g., if acyclic holds then circular (for the same set of fields) is not checked.

To minimize the number of checks in the generated repOk, the simplify method removes redundant properties from set of properties that actually hold, e.g., if a graph is acyclic, there is no need to generate a check for directed-acyclic.

```
String deryaft(Set structs) {
//    post: result is a string representation of a Alloy
//    constraints
//    that represent structural invariants of given structures
      Set classes = allClasses(structs);
      Set fields = allFields(structs);
      Set core = coreFields(fields);
      Set derived = derivedFields(fields, core);
      Set relevantGlobal =
            globalProperties(structs, core, classes);
      Set relevantLocal =
            localProperties(structs, derived, classes);
      Set propertiesThatHold =
            checkProperties(relevantGlobal, structs);
      propertiesThatHold.addAll(
                  checkProperties(relevantLocal, structs));
      simplify(propertiesThatHold);
      return generateInvariants(propertiesThatHold);
}
```

Figure 6: The Deryaft algorithm. The methods allClasses and allFields respectively return a set of all classes and a set of all fields from the given set of structures. The method coreFields (derivedFields) returns the set of core (derived) fields. The methods globalProperties (localProperties) compute sets of global (local) properties relevant to the given structures. The method checkProperties returns a subset of given properties, which hold for all given structures. The method simplify removes redundant constraints. The method generateInvariants translates given properties to Java predicates.

In summary, the algorithm performs the following five key steps:

- Identification of core and derived fields;
- Formulation of global and local properties that are relevant;
- Computation of properties that actually hold;
- Minimization of properties; and
- Generation of Java code that represents properties.

Chapter 5: Alloy Specifications from Instances

Alloy is a first-order relational logic with transitive closure, and it is particularly suitable for specifying structural properties of software. Alloy has steadily been gaining popularity due to the rapid feedback that its SAT-based analyzer provides fully automatically. Alloy users however, still have to manually write specifications in a declarative language and use a paradigm that is different from the commonly used imperative programming paradigm. In this chapter we discuss aDeryaft which is an implementation of our approach for generating Alloy specifications. We discuss how the algorithm developed for Deryaft is also useful for aDeryaft and further we elaborate the subtle differences between generating specification for imperative language versus a declarative language.

Similar to Deryaft, the user constructs by hand a few small concrete instances that represent the constraints of the software structure they want to specify. aDeryaft then fully automatically generates an executable Alloy specification, which represents the constraints that summarize the given structures. To efficiently generate Alloy specifications, aDeryaft exploits the relational basis of Alloy and formulates graph properties that are likely to hold for the given instances. It then checks the properties for these instances and translates the valid properties into Alloy constraints, which it outputs as an Alloy specification.

aDeryaft vs Deryaft

The most obvious difference from user perspective is that aDeryft outputs Alloy formulas instead of Java predicates. Interestingly aDeryaft accepts Java programs as inputs and is able to translate them into Alloy specifications. The user never has to write code in Alloy. And this step also brings out the most important internal difference between aDeryaft and Deryaft. Since aDeryaft can only translate linked structures into Alloy code it can only generate constraints on linked structures and is unable to handle array based structure such a Heap Array discussed in chapter 2.

REPRESENTING AN ALLOY INSTANCE USING JAVA

To illustrate how aDeryaft can be used to construct an Alloy instance using Java, we take a constructive approach and demonstrate by example Java code snippet that builds a set containing the three lists described in Figure 1 of chapter 1.

```
java.util.Set instances = new java.util.HashSet();
List l0 = new List();
l0.header = null;
l0.size = 0;
instances.add(l0);
List l1 = new List();
Entry e0 = new Entry();
l1.header = e0;
l1.size = 1;
e0.elem = 0;
e0.next = null;
instances.add(l1);
List l2 = new List();
Entry e1 = new Entry();
Entry e2 = new Entry();
Entry e3 = new Entry();
l2.header = e1;
l2.size = 3;
e1.elem = 1;
e2.elem = -1;
e3.elem = 0;
e1.next = e2;
e2.next = e3;
instances.add(l2);
```

where the classes List and Entry are defined as expected:

```
class List {
      Entry header;
      int size;
}
```

Using the construction above our size 3 list of figure 1 in Alloy will be given by following instances.

```
List = { L2 }
Entry = { E1, E2, E3 }
Int = { -1, 0, 1, 3 }
header = { <L2, E1> }
size = { <L2, 3> }
elem = { <E1, 1>, <E2, -1>, <E3, 0> }
next = { <E1, E2>, <E2, E3> }
```

As can be seen clearly that aDeryaft is able to save the programmer from trouble of translating the code into Alloy instances. The aDeryfat API is prototypical and its ability to build arbitrary Alloy instances is limited. Nevertheless the approach can be easily generalized for any recursively declared structure.

aDeryaft Algorithm

The algorithm for aDeryaft is very similar to Deryaft but careful reader will note the subtle differences and their ramifications as mentioned in previous section. Given a set of structures, aDeryaft traverses the structures to formulate a set of hypotheses. Next, it checks which of the hypotheses actually hold for the given structures. Finally, it translates the valid hypotheses into Alloy constraints that represent the structural invariants of the given structures, i.e., it generates a predicate that takes an input structure and evaluates to true if and only if the input satisfies the invariants. To make invariant generation feasible, a key heuristic that aDeryaft incorporates to focus on relevant properties is: hypothesize properties about reachability, such as acyclicity or circularity, only for the fields in the core set; and hypothesize local properties that relate derived fields and core fields, e.g., whether a derived field forms two-cycles with some core fields.

To minimize the number of properties that are checked on the given structures, the checkProperties does not check a property p if a property q that contradicts p is already known to be true, e.g., if acyclic holds then circular (for the same setof fields) is not checked. To minimize the number of constraints in the generated repOk, the simplify method removes redundant properties from set of properties that actually hold, e.g., if a graph is acyclic, there is no need to generate a check for directed acyclic. We are considering the use of a decision procedure for simplification. Indeed we may use the Alloy Analyzer to simplify.

In summary, the algorithm performs the following six key steps with the differences bold italicized:

- ***Identification of Alloy signatures and fields;***
- Identification of core and derived fields;
- Formulation of relevant global and local properties;
- Computation of properties that actually hold;
- Minimization of properties; and
- ***Generation of Alloy code that (1) declares signatures and fields and (2) defines constraints on them based on the discovered properties.***

Chapter 6: A Fully Automated Testing Framework

Testing is critical to the production of high-quality code as the dominant method for finding software errors. Currently test inputs are manually generated in a labor intensive fashion. To find possible errors and successfully testing any software system, good test inputs are a key requirement. But due to labor-intensive nature typical inputs only exercise a subset of the functionality of the software. The alternative is to automate generation of test inputs such as proposed by Korat [4]. Unfortunately it shifts the burden of manually supplying exact specification of software system. In this chapter we discuss our vision of a fully automated testing paradigm, which generates bounded exhaustive tests even before writing the code. First we introduce Korat [4] and its non-isomorphic test case generation. Then we describe how to use Korat in conjunction with Deryaft to fully automate the testing.

Korat

Korat is a tool that can generate structurally complex test inputs for Java programs provided specifications of the program. Given a Java predicate and a bound on the size of input structures, Korat automatically generates all nonisomorphic structures for which the predicate always returns true, and therefore are valid structures. Korat uses a finitization (described below) to bound the size of the input structures to the predicate. To enable efficient search for valid structures, Korat encodes each structure with a state, and each finitization bounds the state space. Korat uses backtracking to systematically search this state space to find all valid structures. The figure below gives a high level view of Korat .

```
void koratSearch(Predicate pred, Finitization fin) {
      initialize(fin);
      while (hasNextCandidate()) {
            Object candidate = nextCandidate();
            try {
                  if (pred(candidate))
                        output(candidate);
            } catch (Throwable t) {}
            backtrack();
      }
}
```

Figure 7: Overview of Korat

Nonisomorphic Test Case Generation

Given a Java predicate and a finitization for its input, Korat automatically generates all nonisomorphic inputs for which the predicate returns true. Here we briefly discuss how Korat achieves this, more interested readers are referred to Korat[4].

To generate a finite state space for predicate's structures Korat uses a set of bounds that limits the size of the structures. This is called finiatization and each finitization provides the limits on the number of objects of each class and the values for each field of those objects. As expected of any reasonably complicated structure, it can consist of objects from several classes, and the finitization specifies the number of objects for each of those classes. A set of objects from one class forms a *class domain*. The finitization also specifies a set of values for each field; this set forms a *field domain*, which is a union of several *class domains* and constants. The constants include *null* for pointers and all primitive values for fields of primitive types.

Using the finitization Korat constructs a state space of predicate structures. Korat first allocates the objects that appear in the finitization and then generates a vector of possible values for each field of those objects. Each state assigns a particular value to each of the fields, and the whole state space consists of the Cartesian product of possible values for each field. Korat first creates the state space and gets the root object of all candidates and structures. It then initializes result, the set of (valid) structures found so far in the search, to the empty set. During the search, Korat maintains a stack of fields that the executions of the predicate access; this stack is initially empty.

The search starts with the state set to all zeros. For each state, Korat first creates the corresponding candidate structure by setting the fields according to the indexing of the state into the state space. Korat then executes the predicate to check the validity of the candidate. During the execution, Korat monitors the fields that the predicate accesses and updates the stack in the following way: whenever the predicate accesses a field that is not already on the stack, Korat pushes the field on the stack. This ensures that the fields in the stack are ordered by the first time the predicate accesses them, i.e., the predicate has accessed (for the first time) the fields near the top of the stack after it had accessed (for the first time) the fields near the bottom of the stack. The predicate may access the same field several times, but the order is with respect to the first access. Korat then uses backtracking to generate the next state based on the accessed fields.

Korat avoids generating multiple candidates that are isomorphic to one another. Isomorphism between candidates partitions the state space into isomorphism partitions. The lexicographic ordering is induced by the ordering on the values in the field domains and the field orderings are built by repOk executions. For each isomorphism partition, Korat generates only the lexicographically smallest candidate in that partition.

Fully Automating the Testing

Korat requires specification as a predicate function to generate non-isomorphic test cases. The output of Deryaft is designed to complement this task. Deryaft outputs specification as a predicate function composed of predicate property satisfaction functions. Now we can generate all test cases up to a given size using just a few example instances.

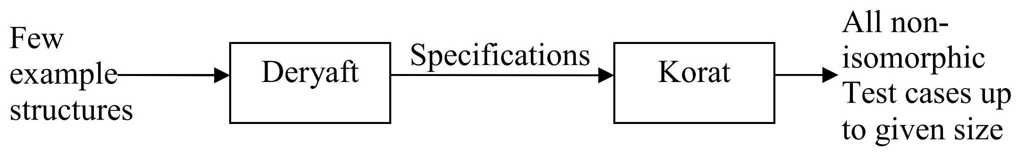


Figure 8: A fully automated testing framework

For more interesting software development processes where the system is evolving more interesting analysis model can be used. The one in which Korat is used to generate some structures which are operated by the program and fed back to Deryaft to learn new structural invariants as shown in the figure below.

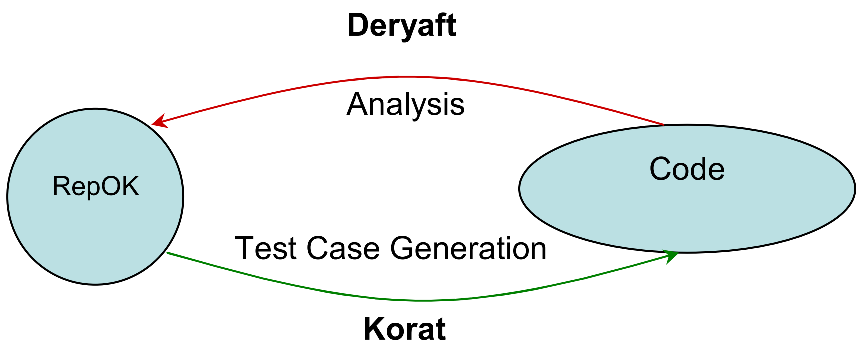


Figure 9: Automated testing framework with evolutionary development process

We can update the specification based on iterative analysis above or can just verify that it has not changed from the last time.

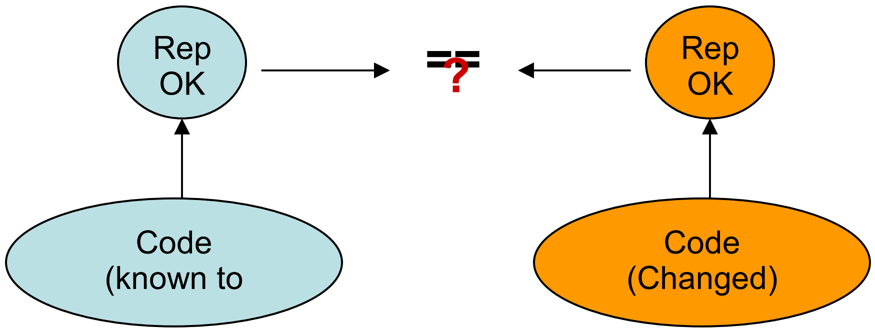


Figure 10: Using Deryft for Change verification

Chapter 7: Experiments and Discussion

In this chapter we first discuss some experiments with classic data structures or varying complexity to show Deryaft's ability to discover invariants. Then we discuss the current limitations of Deryaft and propose some improvements such as how to limit multiple program traversals to speedup Deryaft and how to append new invariants to Deryaft dictionary of rules for better and more accurate rule generation.

Experiments

A complete list of experiments with Deryaft can be found in [7]. This section describes Deryaft's generation of structural invariants for seven subjects, which include some structures library classes as well as a standalone application. For each subject, we constructed by hand five small representative structures and gave them as inputs to Deryaft. For all subjects, Deryaft correctly generated all the standard data structure invariants. The subjects were as follows.

Singly-linked acyclic list

A list object has a header node; each list node has a next field. Integrity constraint is acyclicity along next.

Ordered list

An ordered list is a singly-linked acyclic list, whose nodes have integer elements. Integrity constraints are acyclicity and an (ascending or descending) ordering on the elements.

Doubly-linked circular list

A list object has a header node; each list node has a next and a previous field. Integrity constraints are circularity along next and the transpose relation between next and previous. This subject is based on the library class java.util.LinkedList.

Binary search tree

A binary search tree object has a root node; each node has a left and a right child node, a parent, and an integer key. Integrity constraints are acyclicity along left and right, correctness of parent as well as correct ordering of keys: for each node, its key is larger than any of the keys in the left sub-tree and smaller than any of the keys in the right-sub tree.

AVL tree

An AVL tree [5] is a height-balanced binary search tree. Integrity constraints are the binary search tree constraints as well as the height-balance constraint.

Heap array

Heap arrays provide an array-based implementation of the binary heap data structure that is also commonly known as a priority queue. A heap has a capacity that is the length of the underlying array and a size that is the number of elements currently in the heap. For a heap element at index i, its left child is at index 2 □i+1 and the right child is at index 2 □ i + 2. Integrity constraints are size <= capacity and the heap satisfies the max-heap (respectively min-heap) property: an element is larger (respectively smaller) than both its children.

Intentional name

The Intentional Naming System [1] (INS) is a service location system that allows client applications to specify what they are looking for without having to know where it may be situated in a dynamic network. A key data structure in INS is an intentional name—a hierarchical arrangement of attribute-value pairs that describe service properties. Clients use these names to locate services, while services use them as advertisements.

An intentional name can be implemented using the class AVPair that has two String fields attribute and value and a Vector<AVPair> field children. Structural integrity constraints for AVPair are: (1) attribute and value of the root are null; (2) the children of a node have unique attributes; and (3) the structure is acyclic along the children field.

DISCUSSION

In this section we first discuss the limitations of Deryaft and then propose a few improvements in its time complexity, extensibility and integration with other tools and approaches.

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. 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 algorithm 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. In future, we plan to exploremore complex relations among primitive as well as reference fields. Deryfat implementation described in [27] can handle a class of structures similar to the ones illustrated in this paper.

Optimization of Repeated Traversals

The repOk code that Deryaft outputs typically performs several traversals over a given structure. While an optimization of these traversals might not produce a noticeable speed-up in code generation due to the small size of given structures, optimizations may be quite important in the context of where the generated code is to be used. In fact, based on the usage context, very different optimizations may be necessary.

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. Thus, it is natural to extend Deryaft to incorporate information about the context to tune its generation to the intended use.

Introduction of New Invariants

It would be useful to build an extensible invariant generation system, where new invariants that involve new operators can be plugged into the invariant generator. This would enable not only focused generation on the particular domain of interest, but also generation of a wider class of invariants. Such extensibility requires a language for expressing invariants.

Integration with Other Software Analysis Frameworks

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: Conclusion

This chapter introduces the importance of generating likely invariants, summarizes my work on generating invariants of complex structures and concludes with directions for future work.

IMPORTANCE OF GENERATING INVARIANTS:

Software verification and validation are the critical component of any quality management system of software. The software verification is done against specifications but these are often missing, further even when available can be erroneous. Therefore learning specification will better enable software verification and improve software quality and reliability.

Software testing is very critical for software quality. But due to its labor intensive nature very often it is done inadequately. TestEra[26] and Korant[] have pioneered constraint based test case generation which makes the testing process automated and bounded exhaustive, but for these approaches someone has to supply specifications. By automating the generation of invariants we can further reduce the human involvement in the process.

DERYAFT GENERATION OF LIKELY INVARIANTS:

Dynamically detecting likely invariants, as pioneered by Daikon, is becoming immensely popular. In this thesis, we focused on generating representation invariants of structurally complex data, given a small set of concrete structures. We presented 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.

Deryaft 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 Deryaft 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. We illustrated how the Korat framework can use these invariants to enumerate inputs for Java programs and to check their correctness.

FUTURE WORK

The generation of specifications or structural invariants for complex structures is a very interesting and multidisciplinary field. It has been studied widely in chemistry to analyze complex molecules, closely related is biomedical engineering where it is used for understanding structures within DNA. In mathematics graph theorists and algebraists have made exciting discoveries about structures and their properties. We plan to investigate how the techniques developed in these other areas can help us learn structural properties of complex data.

In the design of Deryaft we have shown how a program can be viewed as an edge labeled graph. The spectra of graphs have been widely used in graph theory to characterize the properties of graphs and extract information from their structure. In a recent work we have performed spectral pattern matching to identify well known shape properties of the heap graphs, and used these properties as rules to index the dictionary of properties. This approach allows the generation of likely invariants but is computationally expensive.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. December 1999. The design and implementation of an intentional naming system: In Proc. 17th ACM Symposium on Operating Systems Principles (SOSP), Kiawah Island.

[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] Kent Beck and Erich Gamma. July 1998. Test infected: Programmers love writing tests: Java Report, 3(7).

[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 ACMSymposium 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 SIGPLANSIGACT 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´ıa, 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.

[24] M. Vaziri and G. Holzmann. November 1998. Automatic detection of invariants in spin: In Proc. SPIN Workshop on Software Model Checking.

[25] John Whaley, Michael C. Martin, and Monica S. Lam. July 2002. Automatic extraction of object oriented component interfaces: In Proc. International Symposium on Software Testing and Analysis (ISSTA).

[26] D. Marinov and S. Khurshid. Nov. 2001. TestEra: A novel framework for automated testing of Java programs: in Proc. 16th IEEE International Conference on Automated Software Engineering (ASE), San Diego, CA.

[27] A. Pervaiz. Nov. 2007. implementing deryaft for generating invariants of structurally complex data: MS Report, University of Texas at Austin. Austin, Texas.

Vita

Muhammad Zubair Malik was born on March 23rd 1978 in Lahore, Pakistan, to Nuzhat Kardar and Awais Malik. He went to Govt. Central Model School lower Mall Lahore from 1st to 10th grade and Later to Govt. College Lahore for his high School studies. He completed his BS in computer system engineering from GIK Institute Pakistan in 1999. After that he worked in the industry as a software engineer for one year with IBM Pakistan and later for one year with Xavor Corp. In 2001 he Joined FAST Lahore, from where he graduated with MS in Computer science in 2003. During his stay at FAST he worked as research officer at Multimedia Lab. He went back to work with his old employer Xavor Corp. as senior software engineer for two more years before joining University of Texas at Austin in 2006. Before this thesis he has one MS thesis and two research papers to his credit. He is a permanent resident of DHA, Lahore Pakistan.

Permanent address: 146/1 Sector U, DHA, Lahore, 54792, Pakistan

This dissertation was typed by Muhammad Zubair Malik.