Generating Representation Invariants of Structurally Complex Data

Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid

The University of Texas at Austin, 1 University Station C5000, Austin, TX 78712
{mzmalik,pervaiz,khurshid}@ece.utexas.edu

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

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.

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.

We present Deryaft, a novel framework for generating representation invariants of structurally complex data given a (small) set of structures. 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].

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 34–49, 2007.

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 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.

We make the following contributions:

- **Algorithm.** Deryaft is a novel algorithm for generating representation invariants of structurally complex data from a given small set of structures;
- **Java predicates.** Deryaft generates invariants as Java predicates that can directly be used in other applications, e.g., for test generation and error recovery;
- **Experiments.** We present experiments using our prototype to show the feasibility of generating invariants for a variety of data structures, including libraries as well as a stand-alone application.

2 Example

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

```
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.

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 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 from 1 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.

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

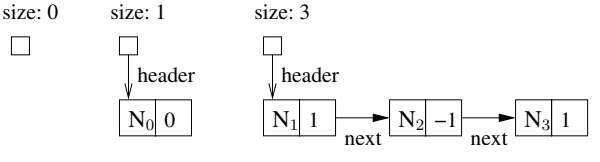

Fig. 1. Three acyclic singly-linked lists, one each containing zero, one, and three nodes, as indicated by the value of the `size` field. Small hollow squares represent the list objects. The labeled arrows represent the fields `header` and `next`. N_0, N_1, N_2, and N_3 represent the identities of node objects. The nodes also contain the integer elements, which for the given three lists range over the set {-1, 0, 1}.

```
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());
}
```

Fig. 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`.

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.

3 Deryaft

This section describes Deryaft. 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.

3.1 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 Section 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.

3.2 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;
}
```

Fig. 3. 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.

3.3 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 connects m to 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.

3.4 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 4 presents the Deryaft algorithm using Java-like pseudo-code. 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 set of fields) is not checked.

```
String deryaft(Set structs) {
    // post: result is a string representation of a Java method
    //       that represents the structural invariants of the
    //       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);
}
```

Fig. 4. 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 relevant global (local) properties. The method `checkProperties` returns a subset of given properties, which hold for all given structures. The method `simplify` removes redundant constraints. The method `generateInvariants` generates a Java predicate that corresponds to the given properties.

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.

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.

3.5 Illustration: Binary Tree Representation of Heaps

To illustrate the variety of invariants that Deryaft can generate, we next present a case study on generating invariants of the heap data structure, which is also called a priority queue [5]. We consider a binary tree representation of heaps.

The following class declares a binary tree with parent pointers:

```
public class BinaryTree {
    Node root;  // first node in the tree
    int size;   // number of nodes in the tree

    private static class Node {
        Node left;
        Node right;
        Node parent;
        int key;
    }
}
```

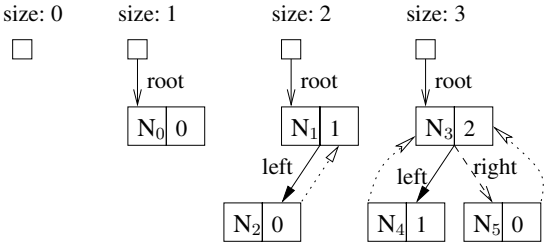

Fig. 5. Four heaps represented using binary trees, one each containing zero, one, two and three nodes, as indicated by the value of the `size` field. Small hollow squares represent the `BinaryTree` objects. The labeled arrows represent the fields `root`, `left`, `right`. The dotted arrows with hollow heads represent `parent` fields, which have not been labeled for clarity. $\mathtt{N}_0$, ..., $\mathtt{N}_5$ represent the identities of node objects. The nodes also contain the integer keys, which for the given four heaps range over the set $\{0, 1, 2\}$.

Consider a binary tree representation of heap, which requires: acyclicity along `left` and `right`; correctness of `parent` and `size`; heap property: the key of a node is greater than any key in a left or right child; and nearly complete binary tree.

Consider the heaps represented in Figure 5. As an example execution of the algorithm for computing the core fields (Figure 3), consider computing the set with respect to these structures. The algorithm initially sets `fs` to $\{$`left`, `right`, `parent`$\}$, i.e., the set that contains all the fields that represent homogeneous relations. Removing `left` from the set changes reachability, e.g., in the case of the structure with three nodes and therefore `left` is core; similarly `right` is core; however, removing `parent` does not effect the reachability in any of the given structures and therefore `parent` is not core.

As an example execution of the `deryaft` algorithm (Figure 4), consider computing the representation invariants for the given structures. The formulation of relevant global properties gives:

- $acyclic$(`root`, $\{$`left`, `right`$\}$)
- $directed\text{-}acyclic$(`root`, $\{$`left`, `right`$\}$)
- $circular$(`root`, $\{$`left`, `right`$\}$)
- $equals$(`size`, $reachable$(`root`, $\{$`left`, `right`$\}$).$cardinality$())
- $equals$(`size` + 1, $reachable$(`root`, $\{$`left`, `right`$\}$).$cardinality$())
- $height\text{-}difference$(`root`, $\{$`left`, `right`$\}$, x)
- $nearly\text{-}complete$(`root`, $\{$`left`, `right`$\}$)

The formulation of relevant local properties gives:

- $two\text{-}cycle$(`root`, `parent`, `left`)
- $two\text{-}cycle$(`root`, `parent`, `right`)
- $is\text{-}null$(`root`, `parent`)
- $\{<, \leq, >, \geq\}$(`root`, $\{$`left`$\}$)
- $\{<, \leq, >, \geq\}$(`root`, $\{$`right`$\}$)

```
public boolean repOk() {
    if (!acyclicCore(root)) return false;
    if (!sizeOk(size, root)) return false;
    if (!nearlyComplete(root)) return false;
    if (!parentNull(root)) return false;
    if (!parentTwoCycleLeft(root)) return false;
    if (!parentTwoCycleRight(root)) return false;
    if (!greaterThanLeft(root)) return false;
    if (!greaterThanRight(root)) return false;
    return true;
}

private boolean parentNull(Node n) {
    return (n.parent == null);
}

private boolean parentTwoCycleLeft(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.left != null) {
            if (current.left.parent != current) return false;
            if (visited.add(current.left)) {
                worklist.addFirst(current.left);
            }
        }
        if (current.right != null) {
            if (visited.add(current.right)) {
                worklist.addFirst(current.right);
            }
        }
    }
    return true;
}
```

Fig. 6. Code snippet of heap invariant generated by Deryaft

The computation of properties that actually hold gives:

– $acyclic(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
– $directed\text{-}acyclic(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
– $equals(\texttt{size}, reachable(\texttt{root}, \{\texttt{left}, \texttt{right}\}).cardinality())$
– $height\text{-}difference(\texttt{root}, \{\texttt{left},\texttt{right}\}, 1)$
– $nearly\text{-}complete(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
– $two\text{-}cycle(\texttt{root}, \texttt{parent}, \texttt{left})$
– $two\text{-}cycle(\texttt{root}, \texttt{parent}, \texttt{right})$
– $is\text{-}null(\texttt{root}, \texttt{parent})$
– $\{>, \geq\}(\texttt{root}, \{\texttt{left}\})$
– $\{>, \geq\}(\texttt{root}, \{\texttt{right}\})$

Removal of redundant properties gives:

– $acyclic(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
– $equals(\texttt{size}, reachable(\texttt{root}, \{\texttt{left}, \texttt{right}\}).cardinality())$
– $nearly\text{-}complete(\texttt{root}, \{\texttt{left}, \texttt{right}\})$

– *two-cycle*(root, parent, left)
– *two-cycle*(root, parent, right)
– *is-null*(root, parent)
– *greater-than*(root, {left})
– *greater-than*(root, {right})

Deryaft's code generation takes these resulting properties and generates Java code, which performs appropriate traversals to check the properties. Figure 6 gives a code snippet of Deryaft's output. The method repOk represents the structural invariants of the given heaps. It invokes several helper methods to perform several traversals on the input structure to determine the structure's validity. The method acyclicCore returns true if and only if the input structure is free of cycles along the fields left and right. The method parentNull checks that the parent of n is null. The method parentTwoCycleLeft checks that for each node n, if n has a left child m, m's parent is n, i.e., parent and left form a two-cycle; parentTwoCycleRight checks that for each node n, if n has a right child m, m's parent is n. The method sizeOk checks the number of nodes reachable from n equals s. The method greaterThanLeft checks that for any node n, if n has a left child m, n's key is greater than m's key; the method greaterThanRight checks that for any node n, if n has a right child m, n's key is greater than m's key.

4 Experiments

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.

5 Discussion

This section discusses current limitations of Deryaft and future work.

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 explore more complex relations among primitive as well as reference fields.

Our Deryaft implementation is under construction. The prototype at this stage 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.

6 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.

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.

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

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

7 Conclusions

Dynamically detecting likely invariants, as pioneered by Daikon, is becoming immensely popular. In this paper, 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.

Acknowledgments

We thank the anonymous reviewers and Darko Marinov for useful comments. This work was funded in part by the Fulbright Program and the NSF Science of Design Program (award #0438967).

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation*, Paris, France, 2005.
3. Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
4. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.
5. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
6. P. Cousot and N. Halbwachs. 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, 1978.
7. Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, August 2000.
8. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.
9. Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. Software Eng.*, 1(1), 1975.
10. Rakesh Ghiya and Laurie J. Hendren. 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*, 1996.
11. Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *Proc. 18th Conference on Automated Software Engineering (ASE)*, San Diego, CA, October 2003.
12. Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002.
13. Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.
14. Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.
15. Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Proc. 12th SPIN Workshop on Software Model Checking*, San Francisco, CA, 2005.
16. Sarfraz Khurshid, Muhammad Zubair Malik, and Engin Uzuncaova. An automated approach for writing Alloy specifications using instances. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, 2006.

17. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
18. Darko Marinov, Sarfraz Khurshid, Suhabe Bugrara, Lintao Zhang, and Martin Rinard. Optimizations for compiling declarative models into boolean formulas. In *8th Intl. Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2005.
19. Anders Moeller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.
20. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.
21. Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using groebner bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004.
22. Mana Taghdiri. Inferring specifications to detect errors in code. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, Washington, DC, 2004.
23. Ashish Tiwari, Harald Rue, Hassen Saidi, and Natarajan Shankar. A technique for invariant generation. In *Proc. 7th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, London, UK, 2001.
24. M. Vaziri and G. Holzmann. Automatic detection of invariants in spin. In *Proc. SPIN Workshop on Software Model Checking*, November 1998.
25. 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.