Deryaft: A Tool for Generating Representation Invariants of Structurally Complex Data

Muhammad Zubair Malik Aman Pervaiz Engin Uzuncaova Sarfraz Khurshid
The University of Texas at Austin
{mzmalik,pervaiz,uzuncaov,khurshid}@ece.utexas.edu

ABSTRACT

Deryaft is a tool 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.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification D.2.5 [**Software Engineering**]: Testing and Debugging;

General Terms: Reliability, Verification

Keywords: Invariant Detection, Representation Invariants

1. INTRODUCTION

Specifications enable a range of powerful software analyses, such as model checking, software testing, runtime verification, and data structure repair. However, manually writing a specification typically represents a significant burden on the developer, especially when specifying properties of programs that manipulate structurally complex data. For such programs, a common form of specifications is representation invariants, which are also termed class invariants in object-oriented languages. These invariants define properties of object graphs that are considered valid for the corresponding class. The object-oriented paradigm requires these invariants to be maintained in all publicly visible states. To check whether the methods maintain the invariants correctly, their description is necessary. For complex structures that have intricate structural properties, writing the invariants correctly is hard.

We recently designed a framework [7, 9] to automate generation of likely representation invariants of structurally complex data. This paper describes the Deryaft tool that implements our framework. Instead of writing a detailed invariant by hand, the user constructs a few small concrete structures that satisfy the desired properties. Deryaft then fully automatically generates an executable Java predicate that represents the invariants of the given structures. The predicate takes an input object graph, traverses it, and returns true if and only if the input satisfies the invariants. Following Liskov [8], we term these predicates `repOk` methods.

Deryaft views the program heap as an edge-labeled graph whose nodes represent objects and whose edges represent fields [6] and focuses on generating graphs properties, which include reachability. 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 the predicate, which it outputs.

To make invariant generation feasible, Deryaft incorporates a number of heuristics, which allow it to hone on relevant properties. For example, it identifies a *core* set of fields such that for any given structure, all the objects that are reachable from the root along any field are also reachable from the root along the core fields only. Deryaft conjectures reachability properties, such as acyclicity, only for the fields in the core set. Deryaft treats the fields not in the core set as *derived* and conjectures relations between derived and core fields, e.g., whether a derived field is the transpose of a core field. 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.

Deryaft draws inspiration from the Daikon invariant generator [5]. A key difference is Deryaft's focus on generating representation invariants of structurally complex data. This focus allows Deryaft to generate such invariants more accurately than Daikon [9].

Constraint generation using a small number of structures (or more generally program executions) cannot be sound and complete in general [5]. The generated constraints are sound with respect to the set of given structures. 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.

Experimental results with Deryaft show that it accurately generates invariants for a range of subject structures, including those from standard libraries as well as a stand-alone application, using only a handful of manually constructed representative structures of small sizes. The invariants generated by Deryaft directly enable automation of various existing software reliability frameworks, such as the Korat test generation framework [2] and the Juzi data structure repair framework [4], which otherwise require the user to provide detailed representation invariants.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

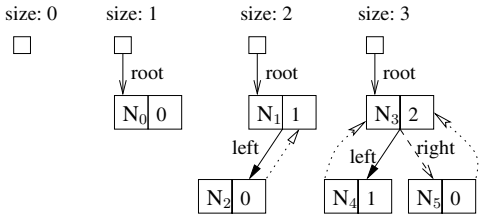

Figure 1: Four heaps represented using binary trees, one each containing zero, one, two and three nodes.

2. EXAMPLE

We illustrate Deryaft's invariant generation using the heap data structure, which is also called a priority queue [3]. We consider a binary tree representation of heaps. The following class declares a binary tree with parent pointers:

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

    private static class Node {
        Node left, right, parent;
        int key; // data element in the node
    }
}
```

The representation invariants of the heap are: acyclicity along `left` and `right`; correctness of `parent` and `size`; heap property, i.e., the key of a node is greater than any key in a left or right child; and complete binary tree, i.e., all levels of the tree except for the last level have the maximum possible number of nodes and the last level is filled from left to right.

Deryaft requires the user to provide a set of valid heaps. Figure 1 represents an example input set. Small squares represent the `BinaryTree` objects. The labeled arrows represent the fields `root`, `left`, `right`. The dotted arrows with hollow heads represent `parent` fields. $\mathtt{N}_0, \ldots, \mathtt{N}_5$ represent the identities of node objects. The nodes also contain the integer keys, which for the heaps shown range over the set $\{0, 1, 2\}$.

Figure 2 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 `sizeOk` checks the number of nodes reachable from `n` equals `s`. The method `complete` checks that the tree rooted at the given `root` is complete. 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 `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`.

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 Deryaft generates. Finally, we describe its algorithm, illustrate it and describe its customization.

3.1 Program heap as an edge-labeled graph

We take a *relational view* [6] of the program heap: we view the heap of a Java program as an edge-labeled directed graph whose

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

...
```

Figure 2: Code snippet of heap invariant generated by Deryaft.

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.

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 sets

Deryaft partitions the set of reference fields of the 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.

Definition. 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 a core set. We aim to identify a *minimal* core set in terms of its size.

To illustrate, the set containing the reference fields `root`, `left`, and `right` in the example from Section 2 is a minimal core set with respect to the given set of heaps.

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

Since `size` and `key` in Section 2 is are primitive fields, `parent` is the only derived field in the heap example.

```
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 3: Algorithm to compute a core set.

Our partitioning of reference fields is inspired by the notion of a *back-bone* in certain data structures [10].

3.2.1 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. 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.

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.

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.

Global: path lengths For non-linear structures, e.g., trees, we consider properties that relate lengths of different paths from root. For example, *balanced* represents that no simple path from the root differs in length from another simple path by more than one.

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*.

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 its successors along reference fields.

```
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, core, derived, classes);
    Set propertiesThatHold =
            checkProperties(relevantGlobal, structs);
    propertiesThatHold.addAll(
            checkProperties(relevantLocal, structs));
    simplify(propertiesThatHold);
    return generateInvariants(propertiesThatHold);
}
```

Figure 4: The Deryaft algorithm.

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. 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 properties that are checked on the given structures, `checkProperties` does not check a property p if a property q that either contradicts or implies p is already known to be true.

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 five key steps:

- 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 Java code that represents properties.

3.5 Illustration

We illustrate the key steps of the algorithm using the heap example from Section 2.

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`}).$cardinality$())

- $equals(\texttt{size} + x,$ $reachable(\texttt{root}, \{\texttt{left}, \texttt{right}\}).cardinality())$
- $height\text{-}difference(\texttt{root}, \{\texttt{left}, \texttt{right}\}, x)$
- $complete(\texttt{root}, \{\texttt{left}, \texttt{right}\})$

The formulation of relevant local properties gives:

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

The computation of properties that actually hold gives:

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

Removal of redundant properties gives:

- $acyclic(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
- $equals(\texttt{size}, reachable(\texttt{root}, \{\texttt{left}, \texttt{right}\}).cardinality())$
- $complete(\texttt{root}, \{\texttt{left}, \texttt{right}\})$
- $two\text{-}cycle(\texttt{root}, \texttt{parent}, \texttt{left})$
- $two\text{-}cycle(\texttt{root}, \texttt{parent}, \texttt{right})$
- $is\text{-}null(\texttt{root}, \texttt{parent})$
- $>(\texttt{root}, \{\texttt{left}\})$
- $>(\texttt{root}, \{\texttt{right}\})$

Deryaft's code generation takes these resulting properties and generates Java code, which performs appropriate traversals to check the properties (Figure 2).

3.6 Customization

To improve the accuracy of invariants generated by Deryaft, the users may provide a configuration file in XML. A user can:

- define core and derived sets;
- select which properties should (not) be considered; and
- define contradictions and implications among properties.

The following code snippet shows an example configuration file for the heap example:

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

<Properties>
  <Global>
    <Property>
      <Name> height-difference </Name>
      <Status> enabled </Status>
    </Property>
  </Global>
  <Local>
    <Property>
      <Name> two-cycle </Name>
      <Status> enabled </Status>
    </Property>
  </Local>
</Properties>

<Implications>
  <Implication>
    <Property>
      <Name>acyclic</Name>
      <Value>true</Value>
    </Property>
    <ImpliedProperty>
      <Name>directed-acyclic</Name>
      <Value>true</Value>
    </ImpliedProperty>
  </Implication>
</Implications>

...
```

Indeed, a configuration file may be empty, in which case Deryaft performs its default analysis.

4. EVALUATION

Experiments [7, 9] show that Deryaft can generate structurally complex invariants for a variety of data structures, including textbook structures, such as singly-linked acyclic lists, sorted lists, doubly-linked circular lists, binary search trees, AVL trees, heaps (including array-based implementations) [3], as well as intentional names [1]. For these subjects, Deryaft accurately identified the invariants using only a handful of given concrete structures of small sizes.

Acknowledgments

This work was funded in part by grants #CCF-0702680 and #IIS-0438967 awarded by the National Science Foundation.

5. REFERENCES

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th International Conference on Verification, Model Checking and Abstract Interpretation (SOSP)*, Kiawah Island, December 1999.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[4] B. Elkarablieh, I. Garcia, Y. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the 22th IEEE International Conference (ASE)*, November 2007.

[5] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[6] D. Jackson and A. Fekete. Lightweight analysis of object interactions. In *Proceedings of the Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

[7] S. Khurshid, M. Z. Malik, and E. Uzuncaova. Generating Alloy specifications from instances. In *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Paphos, Cyprus, November 2006.

[8] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[9] M. Z. Malik, A. Pervaiz, and S. Khurshid. Generating representation invariants of structurally complex data. In *Proceedings of the 13th International Symposium on Theoretical Aspects of Computer Software (TACAS)*, Braga, Portugal, 2007.

[10] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.