Dynamic Shape Analysis via Degree Metrics *

Maria Jump and Kathryn S. McKinley
Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, USA
{mjump,mckinley}@cs.utexas.edu

Abstract

Applications continue to increase in size and complexity which makes debugging and program understanding more challenging. Programs written in managed languages, such as Java, C#, and Ruby, further exacerbate this challenge because they tend to encode much of their state in the heap. This paper introduces *dynamic shape analysis* which seeks to characterize data structures in the heap by dynamically summarizing the object pointer relationships and detecting dynamic degree metrics based on class. The analysis identifies recursive data structures, automatically discovers dynamic degree metrics, and reports errors when degree metrics are violated. Uses of dynamic shape analysis include helping programmers find data structure errors during development, generating assertions for verification with static or dynamic analysis, and detecting subtle errors in deployment. We implement dynamic shape analysis in a Java Virtual Machine (JVM). Using SPECjvm and DaCapo benchmarks, we show that most objects in the heap are part of recursive data structures that maintain strong dynamic degree metrics. We show that once dynamic shape analysis establishes degree metrics from correct executions, it can find automatically inserted errors on subsequent executions in microbenchmarks. These suggests it can be used in deployment for improving software reliability.

Categories and Subject Descriptors D. Software [*D.2. SOFTWARE ENGINEERING*]: D.2.5. Testing and Debugging

General Terms debugging aids, testing tools

Keywords dynamic shape analysis, degree metrics, dynamic invariants

1. Introduction

Object-oriented languages encode program state in objects. With a growing number of objects allocated in the heap, it is unsurprising that many semantic, data structure, and concurrency bugs manifest themselves in heap allocated data structures. Heap analysis therefore has the potential to help developers detect errors and specify their programs correctly.

To manage objects, programmers use regular data structures, such as arrays and recursive data structures. A *recursive data structure* (*RDS*) forms a regular pattern of nodes and references (pointers), such that removing references between any two nodes results in two instances of the same data structure. For example, a singly-linked list with n nodes is a simple recursive data structure and can be divided into two smaller singly-linked lists: one of size x and another of size $n - x$. For a long time, researchers have used *static shape analysis* to characterize recursive data structures in the heap based on the code that creates and manipulates them using [9, 10, 16, 28, 29]. By analyzing program statements, static shape analysis detects recursive data structures and their invariants. For example, it can detect that a singly-linked list has two invariants: $n-1$ nodes have exactly one incoming pointer as well as one outgoing pointer. Despite its recent advances [9], static shape analysis is not used extensively because it requires flow- and context-sensitivity, which makes it very expensive and necessarily conservative.

In this paper, we introduce *dynamic shape analysis*, which dynamically detects recursive data structures and degree invariants that hold during a particular program execution by piggybacking on periodic garbage collections. Our analysis computes a *class-field summary graph* (*CFSG*) which summarizes the dynamic object graph based on class definitions. The *CFSG* records the number of objects and their recursive degree metrics as in- and out-degree invariants. When a specific number of nodes of a data structure exhibit a particular degree, we call it a *fixed metric* and track the number of objects that exhibit the fixed metric. For example, in a singly-linked list with n nodes, exactly $n - 1$ nodes have an out-degree equal to one and the last node has an out-degree equal to zero. For any degree invariant, that is not fixed, the *CFSG* records a *range metric* as the fraction of objects with a given property along with its variance.

A fully accurate dynamic shape analysis requires analysis of the heap after every pointer mutation which is extremely costly. To make the costs more tractable, we piggyback on garbage collection in a tool called ShapeUp that we add to a Java Virtual Machine (JVM). Since garbage collection is periodic and relatively infrequent, dynamic shape analysis can be made efficient. For example, ShapeUp adds an average 4 to 8% to total runtime and less than 1% to space overheads in our system. Performing dynamic shape analysis less frequently trades accuracy for efficiency. ShapeUp loses accuracy since the program can violate degree metrics between collections. Our results, however, indicate that performing dynamic shape analysis relatively infrequently is sufficient to produce information accurate enough to find many errors.

We evaluate ShapeUp by first identifying a variety of both library and custom recursive data structures in SPECjvm and DaCapo Java benchmarks. We demonstrate that the vast majority of objects in the heap are part of recursive data structures and that these data structures maintain dynamic degree metrics for their entire execution. While individual data structures maintain degree

* This work is supported by NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, DARPA NBCH30390004, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

metrics (e.g., an in-degree of one), the heap as a whole does not because of its transient nature. Using microbenchmarks, we show a unique application of dynamic shape analysis in which ShapeUp uses correct executions to develop invariants and then finds errors in incorrect executions. We automate the generation of incorrect executions and find that some errors trigger anomalies in the degree metrics that ShapeUp detects and reports. These results suggest that ShapeUp may be useful for hard-to-find errors that make it into deployment.

In summary, dynamic heap analysis effectively summarizes the objects in the heap by class, finds dynamic invariants, and finds violations by mining much of the heap's regular structure from the object graph. This paper begins the study of dynamic shape analysis by degree metrics by showing how degree metrics can be used to detect and report errors in microbenchmarks. It leaves as future work the study of more precise summaries that can handle multiple instances of a single data structure as well as application to real programs. We begin in the next section by discussing potential uses for dynamic shape analysis in more detail and explain how it differs from its static counterpart.

2. Motivation

Dynamic and static shape analysis differ in their use cases.

Static shape analysis is a flow and context-sensitive analysis that proves that a program correctly constructs and manipulates a recursive data structure. Static analysis strives to detect that a data structure will never violate some invariant, such as the nodes in a binary tree have an in-degree of at most one. Unfortunately there are points in the program (i.e., when a data structure is manipulated) that invariants are temporarily violated. To aid static shape analysis, programmers specify the invariants and the points at which they expect the invariants to hold. Even with this help, static analysis, which is necessarily conservative, cannot always prove some properties even if they do hold. Furthermore, static shape analysis is still intractable for all but modestly-sized programs.

Dynamic shape analysis, instead, samples data structures in the heap for a given program run. It analyzes the current shape of the data structure and, given correct invariants, can determine if the current dynamic shape violates these invariants. It cannot guarantee a data structure will never violate its invariants as invariants can be violated between samples. However, dynamic shape analysis can help developers find bugs that persist early in development, during testing, and after deployment. Data structure reports during development can help programmers find obvious errors. For example, consider the case when a developer intends to create a doubly-linked list, but forgets to set the back pointer and creates a singly-linked list instead. The ShapeUp invariant report would clearly indicate that $n-1$ nodes showed an in-degree of one rather than the expected value of two.

More interestingly, perhaps, is the case of detecting invariant violations during or after deployment. This paper demonstrates that dynamic shape analysis can develop invariants by observing correct executions and then can find anomalous ones, such as when a well-tested program exhibits a runtime error as a result of a race. If the errors persist over time, periodic analysis of the heap detects them. Furthermore, the information from dynamic shape analysis augments static analysis and verification. For example, users can provide dynamically detected invariants as input to static or dynamic verification. Dynamic shape analysis therefore has the potential to help programmers at many stages of development and deployment.

3. Related Work

Related work includes dynamic invariants based on program-counter locations, static shape analysis, error detection and correction using invariant specifications, and C heap analysis.

For a long time, static shape analysis has sought to understand heap structure by analyzing code to identify recursive data structures [9, 10, 16, 28, 29]. Unfortunately, it is not widely used because it requires flow and context-sensitivity which makes it very expensive and necessarily conservative. Our analysis efficiently gives the same information, but is specific to one or more program executions since it observes the current state of the heap rather than analyzing all possible heap states. Dynamic shape analysis, like static shape analysis, can be used to generate specifications and tests.

More recently, dynamic analyses have discovered likely invariants by mining dynamic program behavior, correlating it with program locations, and then identifying anomalous executions [15, 17, 18, 22, 21, 23, 25, 31]. For example, Hangal and Lam showed that crashes are often preceded by anomalous behavior, i.e., the program violates one or more dynamic invariants that were established either on previous executions or earlier in the current execution. They show that recording variable and condition values while reporting unseen values aids debugging. We show that this hypothesis applies to the heap as well, i.e., the heap object graph encodes semantics and unusual heap relationships reveal software flaws.

Recent work has also shown how to detect and correct data structure errors using programmer-specified invariants [1, 4, 11] or user-defined predicate routines [8, 14]. This approach requires a programmer to specify the nature of the data structure and then uses model checking and partial evaluation to detect and fix errors as they occur in the wild. User-defined predicates can encode valuable information, such as which value encodes the number of nodes that should be in the data structure, which ShapeUp will not discover. The advantage of ShapeUp is that it is fully automated and does not require a predicate routine. It detects similar errors by automatically discovering many of the same properties that user predicates contain. Developers can use the results of our analysis to write predicate routines for complex recursive data structures.

HeapMD examines simple heap properties in C programs [12]. Specifically, it shows that many C heaps contain a stable fraction of objects with an in- or out-degree of zero, one, or two. While HeapMD provides inspiration, our work shows that the more transient nature of the Java heap and more complex relationships in the object graph rarely provide stable whole-heap invariants. Differentiating the heap by class and connectivity, however, reveals recursive data structures that do have many stable degree invariants.

Pheng and Verbrugge visualize dynamic data structure evolution from program traces, showing how memory usage and drag varies over time [26]. They analyze complete program traces, whereas we show how to efficiently compute summaries by piggybacking on the garbage collector. Their analysis identifies lists, trees, and directed acyclic graphs, whereas ShapeUp develops invariants.

Jump and McKinley introduce the *class points-from graph* which summarizes the entire heap by user classes and the pointer relationships between them [20]. This heap representation helped developers find memory leaks by identifying growing parts of the graph. Our work is orthogonal and complimentary. This paper extends the class points-from graph by adding field edges, creating a *class-field summary graph*. Furthermore, we use the resulting graph to characterize data structures in the SPECjvm and DaCapo Java programs, to identify recursive degree invariants in correct program executions, and to identify malformed data structures quickly and precisely.

4. Data Structure Analysis

To manage large amounts of data, programs written in modern languages use recursive data structures. Developers implicitly and explicitly maintain invariants over data structures and in the code that allocates and manipulates them. A *recursive data structure*

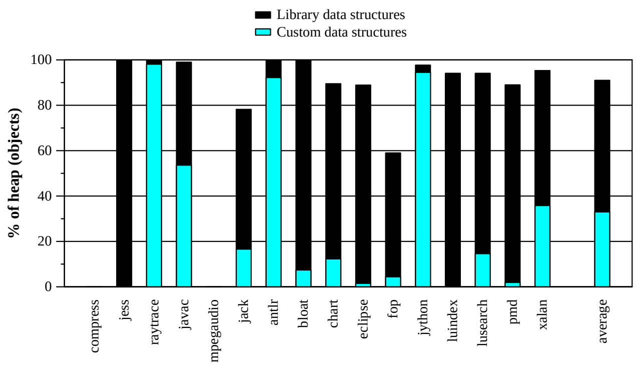


Figure 1. Recursive data structure in the heap

(*RDS*) is a set of objects linked by references (pointers) in a regular pattern such that any part is composed of a smaller or simpler instance of the same data structure. For example, a subset of a singly-linked list is also a singly-linked list. While the definition of the data structure is unbounded, the size of any particular *RDS* in the heap is bounded.

We examine the composition of the heap in terms of the recursive data structures for SPECjvm and DaCapo benchmarks and present them in Figure 1. We separate recursive data structures based on where they are implemented. A *custom* data structure is one that is specifically implemented by the application. Other data structures are implemented in libraries and are simply used by the application. Figure 1 shows that recursive data structures are used ubiquitously in the benchmarks analyzed. While compress and mpegaudio rely strictly on arrays for handling their data, in other benchmarks, 91% of all objects are part of a *RDS*, 33% of which are contained in custom data structures.

In Java and other object-oriented languages, a recursive data structure is implemented separately from the data that the data structure contains. Thus we refine the definition of *RDS* to include object class and differentiate objects of the class that implement the recursive backbone of the data structure. The *recursive backbone* defines the shape of the *RDS* and consists of objects of a single class that reference other object(s) of the same class. For instance, a tree is composed of smaller trees (sub-trees) where the smallest tree is a single node. A given class definition of a tree contains a class `Node` with some number of references to other `Node`s, as well as references to one or more data objects. Table 1 details the microbenchmarks that we implemented to evaluate ShapeUp including singly-linked lists, doubly-linked lists, binary trees, binary trees with parent pointer, and a simplified hashmap. For each data structure, the first column illustrates example instantiations. The second column presents the definition of the `Node` class. The third column shows the *class-field summary graph* (*CFSG*, explained in the next Section), which summarizes the recursive backbone and reflects the shape of the *RDS*. It is this shape that both static and dynamic shape analysis seek to characterize. We next describe how ShapeUp piggybacks on the garbage collector to create this summary.

4.1 Summarizing Data Structures for *RDS* Analysis

Dynamic shape analysis examines the heap at each garbage collection. The state of a program's heap can be expressed as a directed graph $G = \{V, E\}$, where V is the set of all heap-allocated objects and E is the set of references between objects in the heap. That is, if an object o with field f that refers to object p, then the edge $(o.f, p)$ exists in the graph G. The in-degree of an object o is the number of other objects in the heap that reference o. The out-degree of an object o is the number of objects to which o actually refers (*not* the potential number specified by the object's class). The

```
void scanObject(ObjectRef obj) {
  MMType objClass = getObjectType(obj);
  summaryGraph.incNodeCount(objClass);

  int out = 0;
  for(int field = 0 ; field < ob.refs ; field++) {
    ObjectRef childRef = class.getChildRef(obj,field);
    MMType childClass = getObjectType(child);
    summaryGraph.incEdge(objClass, field, childClass);
    if (objClass == childClass) {
      out++;
      // incrementally accumulate in-degree histogram
      int in = childRef.getIndegree();
      // decrement old histogram bin
      summaryGraph.decInHistogram(childRef, in);
      in++;
      // increment new histogram bin
      summaryGraph.incInHistogram(childRef, in);
      childRef.setIndegree(in);
    }
  }
  // directly increment out-degree histogram
  summaryGraph.incOutHistogram(out);
}
```

Figure 2. Pseudocode for accumulating in- and out-degree metrics during object scan.

roots of the heap-graph are references stored in the statics (global variables), stacks, and registers. A tracing garbage collector starts at these roots and detects live objects by performing a transitive closure through all the live object references in the heap. ShapeUp piggybacks on this scan and summarizes the structure of the heap in a *class-field summary graph* (*CFSG*), which describes the dynamic shape of objects per class.

The *CFSG* summarizes the entire heap by *class nodes* and *field edges*. The class summary nodes record the total number of objects of this class in the heap. For each reachable (live) object o_{c_1} discovered during the garbage collection's trace of the heap, ShapeUp determines the object's class c_1 and increments the counter of the corresponding class node. The field edges represent each reference $(o.f, p)$ as a directed edge between o and p distinguished by field f. In order to capture the shape of a recursive data structure, ShapeUp uses degree metrics. A *degree metric* is defined as the in- and out-degree of an object instance. Since dynamic shape analysis tries to characterize the shape of the *RDS* as defined by the recursive backbone, ShapeUp tracks only those degree metrics corresponding to the objects and edges that define the recursive backbone of the *RDS*. ShapeUp tracks these edges in the *CFSG* class node in a distribution histogram.

By definition, the backbone edges of a *RDS* are defined as $(o_{c_1}.f, p_{c_2})$ such that $c_1 = c_2$. In this case, ShapeUp increments and accumulates the out-degree of o_{c_1} and the in-degree of p_{c_2} in a histogram that tracks the number of objects with each in- and out-degree in the *CFSG* class node. Object scanning allows ShapeUp to compute the in- and out-degree of each individual object at low cost. When the collector scans an object, the out-degree is known and ShapeUp directly increments the out-degree histogram that corresponds to the number of non-null outgoing references. Since the in-degrees of object instances are not completely known until the collector completes its transitive closure over the heap, the in-degree histogram is computed incrementally. We detail this process both in pseudocode found in Figure 2 and in the following section.

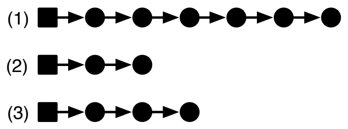
4.2 Step-by-Step Example

Figure 3 shows a step-by-step example of building the *CFSG* for the recursive backbone of a doubly-linked list. In the figure, instances of the same class have the same shape (i.e., the `DoublyLinkedList` object is a square and `Node`s are circles).

Table 1. Data structures in microbenchmarks showing example heap graphs, implementation, and corresponding *CFSG*.

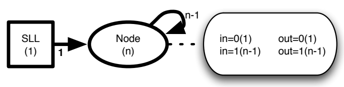
Examples	Implementation	General *CFSG*

Singly-Linked List

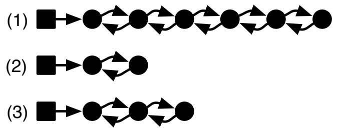


```
class SinglyLinkedList {
  Node head;

  static class Node {
    Object data;
    Node next;
  }
  ...
}
```

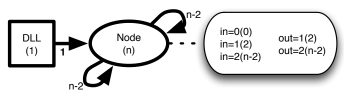


Doubly-Linked List

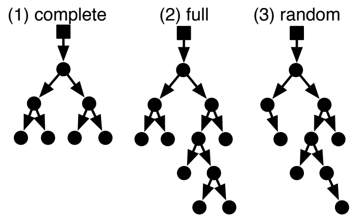


```
class DoublyLinkedList {
  Node head;
  Node tail;

  static class Node {
    Object data;
    Node next;
    Node prev;
  }
  ...
}
```

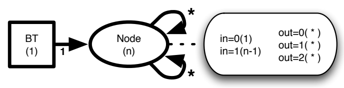


Binary Tree

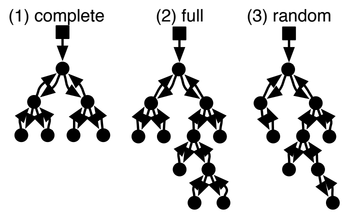


```
class BinaryTree {
  Node root;

  static class Node {
    Object data;
    Node left;
    Node right;
  }
  ...
}
```

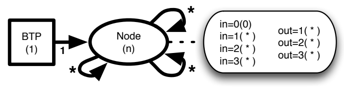


Binary Tree with Parent

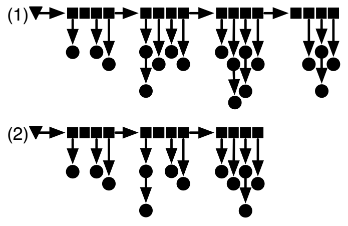


```
class BinaryTreeParent {
  Node root;

  static class Node {
    Object data;
    Node left;
    Node right;
    Node parent;
  }
  ...
}
```



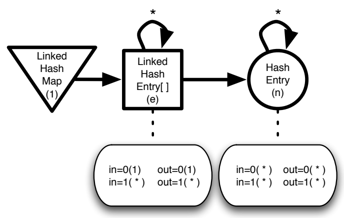
Linked HashMap (simplified)



```
class LinkedHashMap {
  LinkedHashEntry root;

  static class LinkedHashEntry {
    HashEntry[] entries;
    LinkedHashEntry nextGrp;
  }

  static class HashEntry {
    Object key;
    Object data;
    HashEntry next;
  }
  ...
}
```



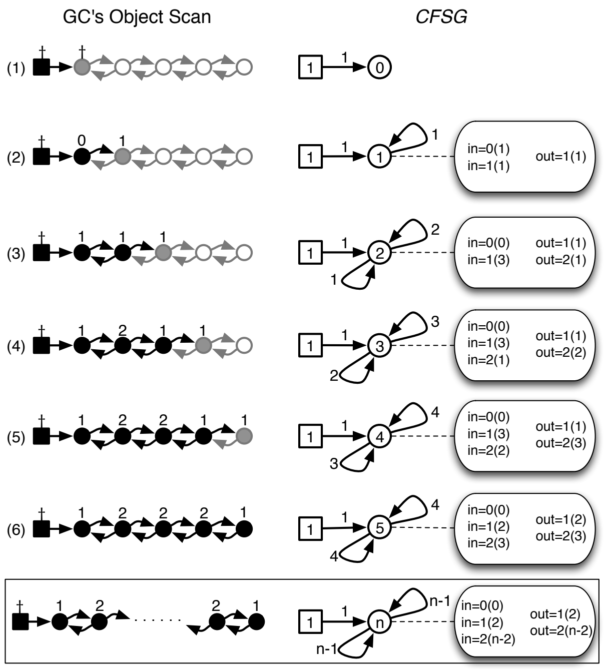


Figure 3. Step-by-step example of building the *CFSG*. Histogram bins are represented by $in = k(\#)$ where $\#$ is the number of objects in that bin.

The left side of the figure shows the heap graph during the garbage collector's object scan. Following the three-color abstraction [13], black objects have already been scanned by the collector, gray objects have been enqueued for scanning, and white object have not yet been seen. The right side shows the corresponding *CFSG*. We walk through this figure step-by-step:

Step 1: We start after the `DoublyLinkedList` object has been scanned (black square). Above each object, the figure shows the in-degree of that object instance. In this case, both objects are marked with † indicating that neither object has been identified as a recursive-backbone object. In the *CFSG*, the node for the `DoublyLinkedList` class shows one instance with one edge to the `Node` class. Notice that the number of `Node` objects in the *CFSG* is zero since the `Node` object has only been enqueued but not scanned (gray). No recursive-backbone object has been identified.

Step 2: The collector scans the first `Node` object that defines the recursive backbone. The *CFSG* captures and identifies the recursive-backbone objects with a *self-edge* representing the *next* field in the `Node` and increments the in-degree histogram for this root object ($in = 0(1)$). We track in- and out-degree with separate histograms in the *CFSG* node. Since the out-degree is known, the corresponding bar in the histogram is incremented ($out = 1(1)$). The in-degree, which must be computed incrementally, requires a byte in the object header to count in-degree for each instance. As the collector scans each object, the children are examined and the in-degree in the object header is incremented. For each gray child, which has already been enqueued for scanning, ShapeUp decrements the histogram bar corresponding to the child's previous in-degree and increments the bar corresponding to the child's new in-degree. For each white child, ShapeUp only increments the bar corresponding to the $in = 1$ ($out = 1(1)$).

Step 3: The scan of the second `Node` object adds a second self-edge in the *CFSG* representing the *prev* pointer in the `Node` and $out = 2$ is added to the *CFSG* node. The in-degree is computed incrementally for each child ($in = 0(0)$, $in = 1(3)$).

Steps 4-5: The scan continues to process each `Node` of the doubly-linked list.

Step 6: At the *tail* of the data structure, we process the final `Node` of the data structure. At this point, ShapeUp has captured the shape of the entire data structure in the *CFSG*.

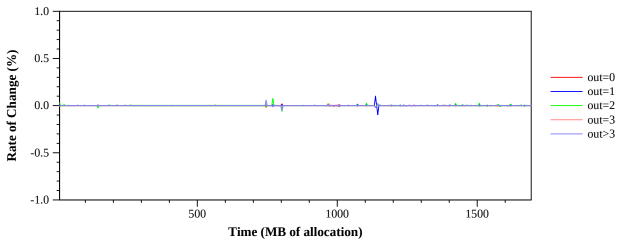
The bottom of Figure 3 shows the most general form of the *CFSG* for a correct doubly-linked list. In the summary, it shows that $n - 2$ objects have in-degree and out-degree equal to two, while 2 objects (the head and the tail) have in-degree and out-degree equal to one. We call these *fixed metrics* of the data structure. Column three of Table 1 shows the general *CFSG* for several different data structures. At the end of the collection, the *CFSG* completely summarizes the number of objects of each class and the number of objects with each in- and out-degree that are live at the time of the collection.

Finally, we add a phase to the end of garbage collection during which we add the current *CFSG* in to a cumulative *CFSG* for the program. We aggregate the average percent of objects with each degree metric (e.g., $in = 0$, $in = 1$, $in = 2$, etc.). For single data structures, we record a *fixed metric* if 0, 1, 2, n, $n - 1$, or $n - 2$ objects exhibit the degree metric. Otherwise, we record a *range metric* as a percentage of objects observed by ShapeUp during one or more executions. We find that fixed metrics are very sensitive to violation when anomalies are introduced while range metrics are more tolerant. In the next section, we evaluate the *CFSG* and show how it can be used to detect errors in recursive data structures.

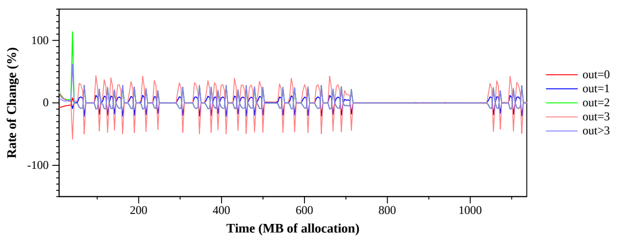
5. Experimental Methodology

We implement ShapeUp in MMTk, a memory management toolkit in Jikes RVM version 2.9.1 [2, 3]. MMTk implements a number of high-performance collectors [5, 6]. We measure performance on SPECjvm [30] and DaCapo v.06-10-MR2 [7] benchmark suites using full-heap mark-sweep and generational mark-sweep collectors. We use configurations that precompile as much as possible, including key libraries and the optimizing compiler (the *Fast* build-time configuration), and turn off assertion checking. Additionally, we remove the nondeterministic behavior of the adaptive compilation system by applying replay compilation [19].

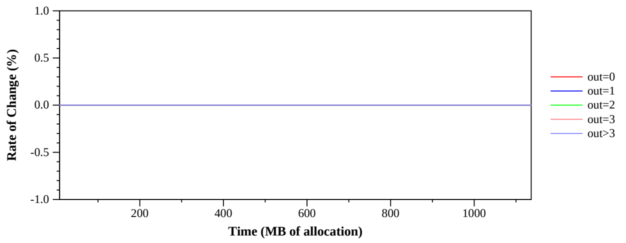
Overhead. We configure ShapeUp to perform dynamic shape analysis together with garbage collection at the natural garbage collection points that are triggered when the heap is full. We experiment with a range of heap sizes that are proportional to the live memory size for a given benchmark, following best practices [7]. We select heap sizes that range from the minimum in which the program can execute to six times the minimum. Common practice for production systems is to select heap sizes of around two times the minimum. The current implementation of *CFSG* adds an average of 4.6% to the total time of all the benchmarks when using a full-heap collector (maximum of 39% for pmd) and an average of 8.4% when using a generational collector (maximum of 92% for hsqldb), while adding <1% to the space requirements of the program. We omit these results for brevity. We believe that this overhead can be reduced further with performance tuning. If users can tolerate more overhead and would like more samples for debugging, such as during testing or development, ShapeUp can analyze the heap more frequently. Users can easily obtain samples at specific program points of interest during debugging by inserting calls to `System.gc()` in their source code.



(a) Out-degree metrics for jython



(b) Out-degree metrics for pmd



(c) Out-degree metrics for `HashMap$HashEntry` in pmd

Figure 4. Rate of change of degree metrics.

6. Degree Metrics Detection

In this section, we analyze the data structures in the SPECjvm and DaCapo benchmarks, classifying them as either library or custom implementations. We find that whole-heap degree metrics are *not* stable while class degree metrics are, including the degree metrics for classes that define recursive backbones. We then evaluate ShapeUp with microbenchmarks that implement singly-linked and doubly-linked lists, binary trees, and linked hashmaps.

6.1 SPECjvm and DaCapo Benchmarks

Recall from Figure 1, programs ubiquitously use *RDSs* to manage the large number of objects in the heap. In our benchmarks, 91% of all objects are part of a *RDS*. Of these objects, 67% come from library implementations and 33% from custom data structure implementations. We gain inspiration from previous work that showed that measuring whole-heap degree metrics was sufficient to discover data structure bugs in C programs [12]. Using the same measure of stability, we find that whole-heap measurements are too granular to predict errors in data structures for Java.

Figure 4 presents two sample whole-heap degree metrics. Figure 4(a) shows the rate of change of out-degree metrics for the entire heap as a function of megabytes of allocation for jython. Figure 4(b) shows the same whole-heap metrics for pmd, whereas (c) shows out-degree metrics for the dominant *RDS* in pmd. Notice that the y-axis scale in (b) is two orders of magnitude larger than in (a) and (c). Figure 4 shows that out-degree metrics for jython are relatively constant, whereas the out-degree metric for the whole heap varies wildly for pmd. The results for pmd are representative of most of our benchmarks. Of 18 benchmarks from SPECjvm and DaCapo, 11 do not have any stable whole-heap degree metrics: jess, raytrace, db, javac, mtrt, jbb2000, bloat, eclipse, fop, lusearch, and pmd. Two programs have many stable degree metrics: 7 degree metrics for compress and 5 for jython. Four programs have 1 or 2 stable degree metrics: mpegaudio (2), luindex (2), jack (1), and antlr (1). Simply monitoring whole-heap degree metrics is not sufficient to understand most program behavior in Java.

Although degree metrics were unstable across the whole heap, the degree metrics of classes are stable. Figure 4(c) illustrates this stability by plotting the out-degree metrics for `HashMap$HashEntry` in pmd, its biggest data structure. Table 2 reports the dominant recursive data structures for each benchmark in column two. Column three indicates whether it is implemented in a library (L) or the benchmark has a custom (C) implementation. If the dominant recursive data structure is library-implemented, the table also includes the most dominant custom recursive data structure. One exception is luindex which does not implement any custom data structures. Column four indicates the average fraction of the heap the dominant data structure occupies. Notice that only two programs contain one very dominate *RDS* that occupies more than 90% of the heap, e.g., jython and luindex. The remaining nine programs have one data structure that occupies 40 to 80% of the heap and many, but not all, are defined in the libraries. For example, raytrace uses a single custom data structure which consumes 78% of the heap on average. Hashmap and its variants are common in the benchmarks and thus we include this more complex data structure in our microbenchmarks.

Columns five through nine enumerate in- and out-degree metrics of zero, one, two, and three, respectively. The top row of these columns reports in-degree metrics and the bottom row reports out-degree metrics. Since these metrics are with respect to the *RDS*, an object can have an in-degree of zero in the *CFSG* when the object has only an incoming pointer from an object of another class.

Each entry is either a fixed or a range metric. For example, all instances (n) of `OctNode` with $out = 0$ in raytrace. This information can be useful during development because even a single node with $out = 0$ would indicate an error. We indicate a range metric with square brackets and show the minimum and maximum fraction of objects observed during the entire execution of the benchmark. If they are the same, we report one number.

Two key trends emerge: (1) Many range metrics are tight, some having zero variance. For example, 72.91% of LinkedHashMap$LinkedHashEntry nodes have in-degree equal to two in jack and luindex. (2) Each *RDS* exhibit fixed degree metrics. Given an expected fixed metric value by the programmer or derived from previous executions, the system can easily automate and report fixed metric violations. For example, if the programmer asserts no object should have an in-degree greater than or equal to two (also called an ownership type [24]), the system can report an error if this metric is violated. The next two sections explore this potential with microbenchmarks and automated error insertion.

6.2 Microbenchmarks

We implement the following microbenchmarks to study them in more detail: singly-linked lists, doubly-linked lists, binary trees, binary trees with parent pointers, and simplified linked hashmaps. We include hashmap because hashmaps account for more than 50% of *RDSs* in 8 of the 16 benchmarks we tested. Recall from Section 4 that Table 1 shows the microbenchmark implementations, sample heap graphs, and corresponding *CFSG*.

Table 3 lists the microbenchmark variations that we tested. We performed 100 trials on correct executions consisting of a random number between 100 and 100,000 of nodes in the *RDS* to collect dynamic degree metrics. At each garbage collection measurement point, ShapeUp calculates the degree metric and merges it to previous correct runs. If the degree metric is fixed, ShapeUp indicates the fixed value for future comparison. If the metric is not fixed,

Table 2. Dominant *RDSs* for selected SPECjvm and DaCapo benchmarks.

Benchmark	*RDS* Node	(L)ibrary or (C)ustom	% heap	metric	= 0	= 1	= 2	= 3
raytrace	OctNode	C	78.0	in	n	0	0	0
				out	n	0	0	0
jack	LinkedHashMap$	L	44.2	in	0	2	[72.91,72.91]	[27.07,27.07]
	LinkedHashEntry			out	0	2	[72.91,72.91]	[27.07,27.07]
	RuntimeNfaState	C	9.4	in	[91.30,91.30]	[8.69,8.69]	0	0
				out	[91.30,91.30]	[8.69,8.69]	0	0
bloat	HashMap$	L	56.6	in	[74.76,87.89]	[12.10,25.23]	0	0
	HashEntry			out	[74.76,87.89]	[8.69,8.69]	0	0
	CallMethodExpr	C	3.7	in	[61.40,90.57]	[7.17,30.71]	[1.89,7.78]	0
				out	[68.50,90.57]	[4.53,15.74]	[3.77,15.75]	0
eclipse	LinkedHashMap$	L	59.0	in	[0.01,0.04]	[72.91,72.97]	[26.99,27.06]	0
	LinkedHashEntry			out	[0.01,0.04]	[72.91,72.98]	[26.98,27.06]	0
	AND_AND_Expression	C	0.4	in	[80.00,100.00]	[0.00,20.00]	0	0
				out	[80.00,100.00]	[0.00,20.00]	0	0
fop	HashMap$	L	51.4	in	[74.73,76.46]	[23.53,25.26]	0	0
	HashEntry			out	[74.73,76.46]	[23.53,25.26]	0	0
	PropertyList	C	4.4	in	[47.77,49.92]	[35.12,37.55]	[11.62,13.38]	[0.78,1.90]
				out	1	n	0	0
jython	PyFrame	C	94.6	in	1	n-1	0	0
				out	1	n-1	0	0
luindex	LinkedHashMap$	L	99.3	in	0	[0.01,0.02]	[72.91,72.92]	[27.05,27.06]
	LinkedHashEntry			out	[0.01,0.02]	[72.92,72.92]	[27.05,27.05]	0
lusearch	WeakHashMap$	L	47.5	in	[43.64,75.62]	[24.37,45.85]	[0,1.04]	[0,0.05]
	WeakBucket			out	[43.54,75.62]	[23.37,46.00]	[0,1.04]	0
	HitDoc	C	2.0	in	[0,100]	[0,100]	[0,66.66]	0
				out	[0,100]	[0,100]	[0,66.66]	0
pmd	HashMap$	L	51.4	in	[73.51,87.50	[12.49,26.48]	0	0
	HashEntry			out	[73.51,87.50]	[12.49,26.48]	0	0
	PackageNode	C	2.0	in	1	[0,80.00]	0	0
				out	1	[0,80.00]	0	0
xalan	ChildIterator	C	34.6	in	0	n	0	0
				out	0	n	0	0

Table 3. Degree metrics discovered by ShapeUp on correct data structures with between 100 and 100,000 nodes.

Data Structure		= 0	= 1	= 2	= 3	= 4, = 5, = 6, =7, >7
Singly-linked list	in	1	n-1	0	0	0
	out	1	n-1	0	0	0
Doubly-linked list	in	0	2	n-2	0	0
	out	0	2	n-2	0	0
Complete binary tree	in	1	n-1	0	0	0
	out	[50.00, 50.16]	1	[49.84, 50.00]	0	0
Full binary tree	in	1	n-1	0	0	0
	out	[50.00, 50.13]	0	[49.87, 50.00]	0	0
Random binary tree	in	1	n-1	0	0	0
	out	[33.66, 35.70]	[28.69, 35.08]	[32.26, 35.61]	0	0
Complete binary tree w/PP	in	0	[50.00, 50.01]	2	[49.92, 50.00]	0
	out	0	[50.00, 50.01]	2	[49.92, 50.00]	0
Full binary tree w/PP	in	0	[50.00, 50.05]	1	[49.86, 50.00]	0
	out	0	[50.00, 50.05]	1	[49.86, 50.00]	0
Random binary tree w/PP	in	0	[33.79, 35.21]	[30.02, 32.61]	[33.61, 34.76]	0
	out	0	[33.79, 35.21]	[30.02, 32.61]	[33.61, 34.76]	0
Linked Hashmap (HashEntry)	in	[31.96, 51.76]	[48.24, 68.04]	0	0	0
	out	[31.96, 51.76]	[48.24, 68.04]	0	0	0

ShapeUp determines the percentage of objects with that metric and determines the range metric from the correct runs.

Table 3 uses the same format for presenting fixed and range metrics as Table 2, which is explained in the previous section. In isolation, data structures show a larger number of fixed metrics. For example, the singly-linked list has 1 node with $out = 0$ which represents the tail of the data structure. Furthermore, the start (or root) of a *RDS* has one node with $in = 0$. In the singly-linked list, out-degree equals one ($out = 1$) for $n - 1$ nodes.

For the less regular data structures, we see fewer fixed metrics. Section 6.3 shows how ShapeUp uses its automatically discovered invariants from correct executions in our microbenchmarks (Table 3) to detect errors when we insert errors.

Table 4. Errors introduced into microbenchmarks

Error	Description	Violation Type	Runs
Singly-Linked List			
cyclic	creates cycle from `tail` to `head`	single	10
cycle	creates random cycle from `tail` to a random object	single	100
Doubly-Linked List			
cyclic	creates cycle from `tail` to `head`	single	10
cycle	creates a random cycle between two objects	multiple	100
disconnect	disconnects random link	multiple	100
skip	creates a skip in the `next` or `prev` pointers	multiple	100
random	randomly insert errors (cycle, disconnect, or skip)	multiple	100
Binary Tree			
linkerror	creates a connection from a `null` pointer to a random object	multiple	100
Binary Tree with Parent Pointer			
disconnect	delete a random reference	multiple	100
linkerror	creates a connection from a `null` pointer to a random object	multiple	100
random	randomly insert errors (disconnect or linkerror)	multiple	100
Linked Hashmap			
bucketlink	randomly connects two buckets in the hashmap	multiple	100

```
if (metric is constant) {
  if (thisMeasure != constantMeasure) {
    // fixed metric violation
  }
} else {
  thisPercent = thisMeasure/numberOfObjects
  if (thisPercent < minPercent ||
      maxPercent < thisPercent) {
    // range metric violation
  }
}
```

Figure 5. Pseudocode for discovering dynamic invariant violations.

6.3 Automated Error Detection

This section explores using ShapeUp to automatically detect errors by using dynamically discovered invariants from correct runs to train ShapeUp. While we expect most errors to be discovered during testing, even well-tested programs have errors. For example, a data race that causes a cycle in a linked-list in a concurrent data structure may show up only in deployment.

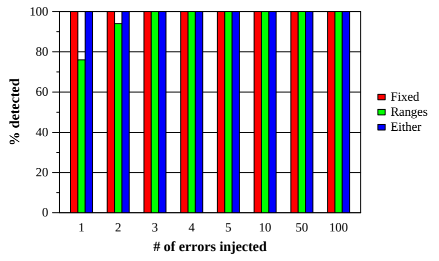
In training, we use correct program executions to dynamically discover degree metrics as shown in Table 3. In testing mode, we compare the current execution to the previously discovered degree metrics. On each full-heap garbage collection, ShapeUp analyzes the data structures and compares this sample to the stored dynamic degree metrics from correct executions. Figure 5 shows how ShapeUp compares metrics from this measurement with previous measurements to detect violations. If the metric is fixed and the current measurement is not the same, ShapeUp reports a violation. Otherwise, ShapeUp reports a potential violation if the fraction of objects with the metric falls below the minimum or above the maximum of a given range metric.

We insert random errors into each of the data structures in our microbenchmark as described in Table 4. For example, a *disconnect* error introduced into a doubly-linked list randomly chooses a node and than randomly chooses to disconnect one of the edges from that node. In the case of *random*, we randomly select one of the other error types to insert for each error. The insertion and deletion of edges in *RDSs* changes the shape of the recursive backbone and thus violate degree metrics. For each type of error, we perform 100 tests that insert 1, 2, 3, 4, 5, 10, 50, and 100 errors into *RDSs* that have 100,000 nodes. Table 5 illustrates the percentage of runs where ShapeUp successfully reports an error.

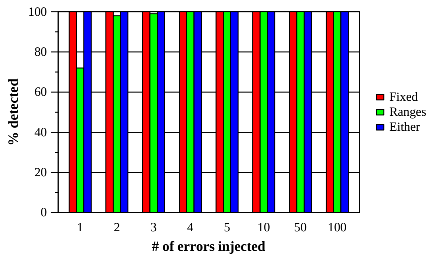
Table 5. Percentage of runs with detected errors.

	Number of errors injected							
Error	1	2	3	4	5	10	50	100
Singly-Linked List								
cyclic	100	n/a						
random	100	n/a						
Doubly-Linked List								
cyclic	100	n/a						
cycle	100	100	100	100	100	100	100	100
disconnect	100	100	100	100	100	100	100	99
skip	100	100	100	100	100	100	100	100
random	100	100	95	98	99	96	98	98
Complete Binary Tree								
linkerror	100	100	100	100	100	100	100	100
Full Binary Tree								
linkerror	100	100	100	100	100	100	100	100
Random Binary Tree								
linkerror	100	100	100	100	100	100	100	100
Complete Binary tree with Parent Pointer								
disconnect	53	92	93	99	100	100	100	100
linkerror	100	100	100	100	100	100	100	100
random	85	97	100	100	100	100	100	100
Full Binary Tree with Parent Pointer								
disconnect	72	83	92	97	100	99	100	100
linkerror	100	100	100	100	100	100	100	100
random	86	97	99	100	100	100	100	100
Random Binary Tree with Parent Pointer								
disconnect	14	22	27	35	41	69	100	100
linkerror	31	63	73	84	94	97	100	100
random	29	36	56	66	64	90	100	100
Linked Hashmap								
bucketlink	0	0	0	1	0	3	12	25

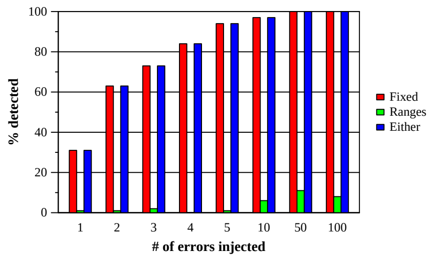
In Figure 6, we report the percentage of runs for which ShapeUp reported a fixed metric violation, a range metric violation, or either violation for the binary tree with parent pointer. Fixed metrics show a greater degree of sensitivity to shape violations. In fact, for every range metric violation, there was a corresponding fixed violation. Figure 6(a-c) show results for *linkerror*, which adds a link from a leaf node $node_l$ to a randomly selected node $node_r$ in the tree. Adding a link from $node_l$ to $node_r$ changes the in-degree of $node_r$ which is quickly detected except in the case of



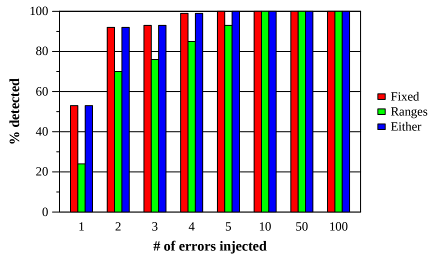
(a) Complete Binary Tree w/ PP (linkerror)



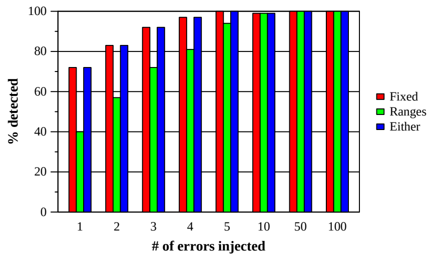
(b) Full Binary Tree w/ PP (linkerror)



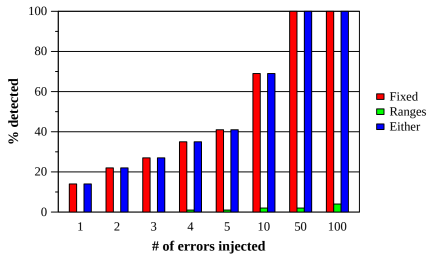
(c) Random Binary Tree w/ PP (linkerror)



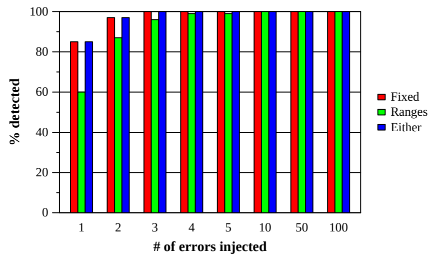
(d) Complete Binary Tree w/ PP (disconnect)



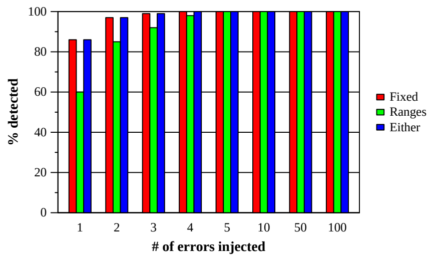
(e) Full Binary Tree w/ PP (disconnect)



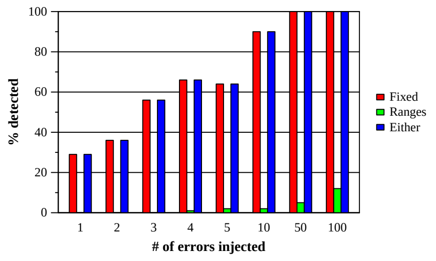
(f) Random Binary Tree w/ PP (disconnect)



(g) Complete Binary Tree w/ PP (random)



(h) Full Binary Tree w/ PP (random)



(i) Random Binary Tree w/ PP (random)

Figure 6. Errors detected in binary trees with parent pointer

the random binary tree with parent pointer. Examination of the degree invariants in Table 3 shows why this is true. For the complete and full versions of the binary tree with parent pointer, shifting the in-degree by one results in a fixed metric violation. For the random binary tree with parent pointer, ShapeUp has less success detecting errors in the corresponding random structure. Figure 6(d-f) show results for *disconnect* errors. Since our approach will not detect errors if the error makes objects unreachable, we find mixed success detecting disconnect errors. For example, disconnecting the edge connecting a leaf node makes the corresponding leaf node unreachable and thus the resulting data structure does not violate its degree invariants. Disconnect errors require semantic invariant detection which is orthogonal to this work. We believe these errors would be caught earlier in development, or could be detected by adding code that keeps track of the expected number of objects in the data structure. Finally, Figures 6(g-i) show results when errors are randomly inserted.

Since ShapeUp only samples the heap at garbage collection time, one issue is how quickly ShapeUp will detect an error, or if ShapeUp will miss an error that gets corrected before the next garbage collection. Our experience shows that data structure errors are likely to persist in the heap because programmers do not generally include code to correct these types of errors (or many others). Depending on the application, there are a variety of actions the system might take once an error is discovered. The system could halt the program or report the error to the developers. If developers include correction code, the system could execute this code, as proposed in prior work [8, 14].

7. Conclusions

Programmers are increasingly challenged by the size and complexity of the programs they create. As programs allocate an increasing number of objects in the heap, heap analysis becomes critical for program understanding and debugging. In this paper, we present *ShapeUp* a *dynamic shape analysis* tool that characterizes the shape of recursive data structures by summarizing the heap-graph in a *class-field summary graph* (*CFSG*) with very low overheads. The *CFSG* completely summarizes the number of objects of each class and the references between them by field. Degree metrics of recursive-backbone objects capture the shape of the recursive data structures in the form of dynamic invariants. We evaluate ShapeUp by characterizing recursive data structures in SPECjvm and DaCapo benchmarks. We demonstrate that the vast majority of objects in the heap are part of recursive data structures. While summarizing degree metrics across the entire heap in Java is not sufficient to understand most program behavior, we show that degree metrics for a single recursive data structure maintain invariants for their entire execution. We show how to use dynamically-discovered degree metrics to find errors in incorrect executions of microbenchmarks showing that for some data structures a single error is sufficient to trigger a violation that was reported by ShapeUp.

Future work should (1) explore more precise summaries for degree metrics including cases where multiple instances of a recursive data structure exist using lightweight techniques to separate *RDS* instances [27]; (2) consider data objects in addition to recursive types; and (3) evaluate ShapeUp in real applications. This paper shows that dynamic heap analysis can efficiently summarize the heap by class in such a way to find dynamic invariants and discover violations by mining degree metrics from the heap's regular structure.

References

[1] E. Aftandilian and S. Z. Guyer. GC assertions: Using the garbage collector to check heap properties. In *Workshop on Memory System Performance and Correctness*, pages 36–40, Seattle, Washington, March 2008.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, Colorado, November 1999.

[4] M. Arnold, M. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 143–162, Nashville, Tennesee, October 2008.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, New York, New York, June 2004.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *International Conference on Software Engineering*, pages 137–146, Scotland, United Kingdom, May 2004.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 2006.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, Rome, Italy, July 2002.

[9] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM Symposium on the Principles of Programming Languages*, pages 289–300, Savannah, Georgia, January 2009.

[10] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 1990.

[11] F. Chen and G. Rosu. Mop: An efficient and generic runtime verification framework. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 569–588, Montreal, Quebec, Canada, October 2007.

[12] T. M. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, 2006.

[13] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1978.

[14] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. STARC: Static analysis for efficient repair of complex data. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Montreal, Canada, October 2007.

[15] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 2000.

[16] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[17] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black box components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, San Diego, California, June 2007.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, Orlando, Florida, 2002.

[19] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, Vancouver, British Columbia, Canada, October 2004.

[20] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for Java. In *ACM Symposium on the Principles of Programming Languages*, pages 31–38, Nice, France, January 2007.

[21] Y. Kannan and K. Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *International Symposium on Software Testing and Analysis*, pages 283–294, Seattle, Washington, July 2008.

[22] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM Conference on Programming Language Design and Implementation*, pages 141–154, San Diego, California, 2003.

[23] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for backtrace of noncrashing bugs. In *SIAM International Conference on Data Mining*, pages 286–297, Newport Beach, California, April 2005.

[24] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming*, pages 74–98, Nantes, France, July 2006.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, California, 2007.

[26] S. Pheng and C. Verbrugge. Dynamic shape and data structure analysis in Java. Technical Report Sable TR 2005-3, McGill University School of Computer Science, October 2005.

[27] E. Raman and D. I. August. Recursive data structure profiling. In *Memory System Performance*, pages 5–14, Chicago, Illinois, June 2005.

[28] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[30] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[31] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *International Symposium on Microarchitecture*, pages 269–280, Portland, Oregon, December 2004.