Dynamically Discovering Program Invariants Involving Collections

Michael D. Ernst[†], **William G. Griswold**[‡], **Yoshio Kataoka**[†], and **David Notkin**[†]

[†]Dept. of Computer Science & Engineering
University of Washington
Box 352350, Seattle WA 98195-2350 USA
{mernst,kataoka,notkin}@cs.washington.edu

[‡]Dept. of Computer Science & Engineering
University of California San Diego, 0114
La Jolla, CA 92093-0114 USA
wgg@cs.ucsd.edu

Abstract

Explicitly stated program invariants can help programmers by characterizing aspects of program execution and identifying program properties that must be preserved when modifying code; invariants can also be of assistance to automated tools. Unfortunately, these invariants are usually absent from code. Previous work showed how to dynamically detect invariants by looking for patterns in and relationships among variable values captured in program traces. A prototype implementation, Daikon, recovered invariants from formally-specified programs, and the invariants it detected assisted programmers in a software evolution task. However, it was limited to finding invariants over scalars and arrays. This paper presents two techniques that enable discovery of invariants over richer data structures, in particular collections of data represented by recursive data structures, by indirect links through tables, etc. The first technique is to traverse these collections and record them as arrays in the program traces; then the basic Daikon invariant detector can infer invariants over these new trace elements. The second technique enables discovery of conditional invariants, which are necessary for reporting invariants over recursive data structures and are also useful in their own right. These techniques permit detection of invariants such as "$\mathsf{p.value} > \mathsf{limit}$ or $\mathsf{p.left} \in \mathsf{mytree}$", The techniques are validated by successful application to two sets of programs: simple textbook data structures and student solutions to a weighted digraph problem.

1 Introduction

Previous research demonstrated the feasibility of dynamically detecting likely program invariants by analyzing traces of variable values [ECGN], and showed how to improve the speed of invariant detection and the usefulness of its output [ECGN00]. A prototype implementation, Daikon, was both accurate — it reported explicitly stated invariants in formally specified textbook programs — and useful — it discovered, in an undocumented C program, invariants that programmers found helpful in modifying the program. Daikon discovered invariants over scalars and arrays, but could not infer invariants involving richer collections of data. This paper extends the previous techniques to discover such invariants. For example, Daikon can now discover properties over collections represented using linked lists.

Our approach discovers invariants from program executions by instrumenting the source program to trace the variables of interest, running the instrumented program over a test suite, and checking properties over both the instrumented variables and derived variables not manifest in the program. This paper describes two specific extensions that allow the inference of invariants over collections:

- The instrumenter explicitly records in the trace file collections that are implicit (e.g., pointer-based). It does so by traversing the collection and writing out the visited objects as an array. We call this process *linearization*. Representing the fields of collections as arrays permits inference of collection-wide properties (such as membership tests or summing a field over the data structure) without further modifying Daikon.
- Daikon computes *conditional invariants* (invariants that are not universally true) by splitting data traces into parts based on predicates it chooses, then detecting invariants on each part. Conditional invariants are essential for recursive data structures, which behave differently in their base and recursive cases. (Conditional invariants are also useful beyond recursive data structures.) We discuss several policies for splitting data traces and experimentally evaluate a simple one that extracts boolean conditions from the program source; in practice, it works well.

We have implemented these techniques and applied them to two sets of programs: the first is taken from a data structures textbook, and the second is student solutions to a weighted digraph problem. Daikon finds most of the relevant invariants and misses few of the relevant invariants. In addition, the invariants have helped us identify several oversights in the design and implementation of some of the data abstractions. See Section 5 for details.

As a motivating example, consider (part of) Daikon's output for a program that uses a linked list as a priority queue:

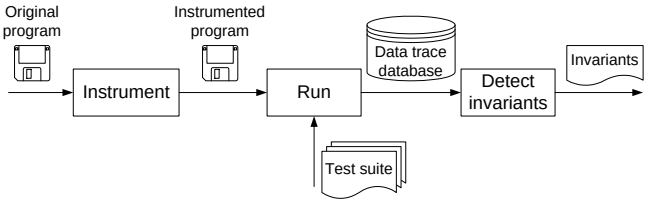

Figure 1: An overview of dynamic invariant inference as implemented by the Daikon tool.

```
PriorityQueue:::CLASS
  prio.closure(next).rank sorted by <=
void PriorityQueue.insert():::EXIT
  size(prio.closure(next)) =
       size(orig(prio.closure(next))) + 1
Object PriorityQueue.remove():::EXIT
  return = orig(prio.next.element)
  prio.next = orig(prio.next.next)
```

This output states that all elements reachable via `next` pointers from the root `prio` are sorted by their `rank` fields, that insertions increase the size of the priority queue, and that removals always occur at the beginning of the queue.

Section 2 provides background on dynamic invariant detection. Section 3 discusses linearization for manipulating pointer-accessed collections. Section 4 describes how to create and test conditional invariants, and evaluates one such strategy. Section 5 experimentally assesses the effectiveness of these techniques. Section 6 presents ongoing work in performing inference online, in conjunction with the instrumented program. Section 7 discusses related work, and Section 8 concludes.

2 Background

Dynamic invariant detection [ECGN] discovers likely invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values (Figure 1). The inference step tests a set of possible invariants against the values captured from the instrumented variables; those invariants that are tested to a sufficient degree without falsification are reported to the programmer. As with other dynamic approaches such as profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. The Daikon inference engine is language independent, currently supporting instrumenters for C, Java, and Lisp.

Daikon detects invariants at specific program points such as loop heads and procedure entries and exits; each program point is treated independently. The invariant detector is provided with a variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. Each of a set of possible invariants is tested against various combinations of traced variables. The following lists some classes of invariants Daikon computes, where x, y, and z are variables, and a, b, and c are computed constants:

- invariants over any variable, such as being constant ($x = a$), taking its values from a small set ($x \in \{a, b, c\}$), etc.
- invariants over a single numeric variable, such as being in a range ($a \leq x \leq b$), non-zero, modulus ($x \equiv a \pmod b$), etc.
- invariants over two numeric variables, such as a linear relationship ($y = ax + b$), an ordering relationship ($x \leq y$), functions ($x = fn(y)$) for built-in unary functions, combinations of invariants over a single numeric variable ($x + y \equiv a \pmod b$), etc.
- invariants over three numeric variables, such as a linear relationship ($z = ax + by + c$), functions, etc.
- invariants over a single sequence variable, such as minimum and maximum sequence values, lexicographical ordering, element ordering, invariants holding for all elements in the sequence, etc.
- invariants over two sequence variables, such as an elementwise linear relationship, lexicographic comparison, subsequence relationship, etc.
- invariants over a sequence and a numeric variable, in particular membership ($x \in y$).

For each variable or tuple of variables, each potential invariant is tested. Each potential unary invariant is checked for all variables, each potential binary invariant is checked over all pairs of variables, and so forth. A potential invariant is checked by examining each sample (i.e., tuple of values for the variables being tested) in turn. As soon as a sample not satisfying the invariant is encountered, that invariant is known not to hold and is not checked for any subsequent samples. Because false invariants tend to be falsified quickly, the cost of computing invariants tends to be proportional to the number of invariants discovered. All the invariants are inexpensive to test and do not require full-fledged theorem-proving.

To enable reporting of invariants regarding components or properties of aggregates, Daikon represents such entities as additional derived variables available for inference. For instance, if array `a` and integer `lasti` are both in scope, then properties over `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program text. Derived variables are treated just like other variables by the invariant detector, permitting the engine to infer invariants that are not hardcoded into its list. For instance, if `size(A)` is derived from sequence `A`, then the system can report the invariant $i < size(A)$ without hardcoding a less-than comparison check for the case of a scalar and the length of a sequence. For performance reasons, derived variables are introduced only when known to be sensible. For instance, for sequence `A`, the derived variable `size(A)` is introduced and invariants are computed over it before `A[i]` is introduced, to ensure that `i` is in the range of `A`.

An invariant is reported only if there is adequate evidence of its plausibility. In particular, if there are an inadequate number of samples of a particular variable, patterns observed over it may be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input. The property is reported only if its probability is smaller than a user-defined confidence parameter.

3 Invariants over collections

The Daikon invariant detector computes invariants only over scalars and arrays. For programs that use richer data structures, reporting such invariants is useful but limiting. This section extends the previous techniques to allow Daikon to discover invariants over collections of data represented using structures other than arrays. Our approach is to extend the instrumenter to find collections that are implicit in the program, linearize them, and record them explicitly in the trace file as arrays.

This approach requires no change to Daikon's inference engine. This design decision stabilized the most complex aspect of Daikon, the invariant detector; in turn, this has increased the robustness of and our confidence in Daikon's implementation. Furthermore, as we show in Section 5, this approach works quite well for the programs we have tried to date. (Linearizing collections adds many variables to the data trace; Section 6 discusses strategies to reduce this cost.)

To linearize an arbitrary-sized collection as an array requires selecting a root, determining a method for traversing the collection, and selecting the field or fields in the collection's objects to be written into the trace file. Roots are selected from variables that are explicit in the program. The linearization process is driven by the objects found when a root is explored. If a field is found that leads from an object to another object of the same type (e.g., `element.next`), the instrumenter outputs the two objects as successive elements in the same array. If there are multiple fields with this property (e.g., a `prev` field in addition to the `next` field), then one linearization is done for each field and multiple arrays are written into the trace file.

When an element is written to the data trace file, its fields with non-recursive types are also written out. This process continues to a user-specified depth. The resulting new variables are named according to the expressions that produce them at runtime, such as `person.birthdate`.

The linearized arrays written to the trace file are given new names. If `header` contains a recursive `next` field, then `header.closure(next)` names the collection of all elements reachable by following `next` fields — that is, the linked list rooted at `header`.

Traversal to the next object in a collection can be handled similarly for other representations. As one example, a program may create a pool of objects using an array, using the indices of the array as links within the pool (as is frequently done in Fortran). Following the indices through the array identifies the elements to be written to the trace file. Similar structures can be defined using hash tables.

As it is linearizing collections, the instrumenter writes to the trace file additional information about the structures, such as whether they are cyclic, a dag, or a tree, which is computed during the data structure traversal but is not evident from the linearized form. This information is written as scalars that can be handled directly by the Daikon invariant detection engine.

Invariants over collections can be classified as either local invariants or global invariants. Local invariants relate a small number of objects, as in $\mathsf{emp.dept} = \mathsf{emp.manager.dept}$. By contrast, global invariants involve an arbitrary-size part of the collection, as in $\mathsf{x} \in \mathsf{mytree}$. Examples of global invariants include $\mathsf{num\_used} < \mathsf{size}(\mathsf{mylist})$ and mytree is sorted. Local invariants do not always imply global ones; for instance, knowing $\mathsf{a}[\mathsf{i}-1].\mathsf{rank} < \mathsf{a}[\mathsf{i}].\mathsf{rank}$ (for unconstrained `i`) implies that array a is sorted, but knowing $\mathsf{p.left.rank} < \mathsf{p.rank} < \mathsf{p.right.rank}$ (for all p, when the specified elements exist) does not imply that the tree containing p is sorted.

Therefore, global invariants must be checked explicitly. This is simple, however, given the explicit linearized collections in the trace file. (Previously described techniques could already detect local invariants.)

4 Conditional invariants

Many important program properties are not universally true. For instance, the local invariant over a sorted binary tree, $\mathsf{p.left.value} < \mathsf{p.right.value}$, is true only if p, `p.left`, and `p.right` are non-null. Other examples include a deletion routine with the postcondition $\mathsf{if}\ \mathsf{x} \in \mathsf{orig}(\mathsf{list})\ \mathsf{then}\ \mathsf{size}(\mathsf{list}) = \mathsf{size}(\mathsf{orig}(\mathsf{list})) - 1$, where `orig(list)` is the value of `list` on entry to the routine, and an absolute value routine with postcondition $\mathsf{if}\ \mathsf{arg} < 0\ \mathsf{then}\ \mathsf{result} = -\mathsf{arg}\ \mathsf{else}\ \mathsf{result} = \mathsf{arg}$. Conditional invariants are particularly important for programs that manipulate recursive data structures, because different properties typically hold in the base case and in the recursive case.

The mechanism for detecting conditional invariants is to split data traces into parts based on some predicate, perform invariant inference on each part, and report those subinvariants, contingent on the splitting predicate. An example is $\mathsf{if}\ \mathsf{p} \neq \mathsf{null}\ \mathsf{then}\ \mathsf{p.value} > \mathsf{MINVAL}$.

The splitting policy determines the predicate used for splitting the data traces into parts. It is infeasible to try all possible splitting predicates or to automatically predict which splits are best for the user. We considered the following splitting policies, which could be used singly or in combination:

- A *static analysis* policy selects splitting conditions based on analysis of the program's source code.
- A *special values* policy compares a variable to preselected values chosen statically (such as null, zero, or literals in the source code) or dynamically (such as commonly-occurring values, minima, or maxima).
- A policy based on *exceptions to detected invariants* tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariant. If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant and an invariant over the exceptions.
- A *random* policy could perform exhaustive or stochastic splitting over a sample of the data, then re-use the most useful splits on the whole data.

- A *programmer-driven* policy allows a user to select splitting conditions *a priori*.

We implemented the static analysis policy of using boolean expressions in a method and side-effect-free zero-argument boolean member functions; the splitting conditions are automatically generated and applied. The results in Section 5 demonstrate that not only is this policy easy to implement, it also works well for the programs we considered.

One reason for the success of static splitting may be that the methods in the programs we tested (see Section 5) are relatively simple. For example, the `LinkedList` program has only a single conditional in its body and using this as a splitting condition led to the discovery of useful conditional predicates. If and when we find practical limitations of the static splitting policy, we will pursue the more complicated ones.

5 Assessment

To assess our techniques for inferring invariants over programs that use pointer-based collections, we analyzed the the quality of Daikon's output for two sets of programs. The first set consists of programs from a data structures textbook [Wei99]. Because these programs are small, described in the textbook, and implement well-understood data structures, we were able to determine a "gold standard" of invariants that should be reported by an ideal invariant detector. The second set of programs were written (to a single specification) by students in a software engineering course at MIT. These programs are larger (1500–5000 lines) and more realistic, and the students wrote down representation invariants; however, we found that frequently those specifications were incomplete or not satisfied by the code.

Our measure of quality for an invariant is *relevance* [ECGN00]. An invariant is relevant if it assists a programmer in a programming activity. Relevance is inherently contingent on the particular task, as well as the programmer's capabilities, working style, and knowledge of the code base. Because no automatic system can know this context, Daikon necessarily reports some invariants that the user does not find helpful and omits other invariants that the user might find helpful.

We manually classified reported invariants as potentially relevant or not relevant based on our own judgment, informed by careful examination of the program and the test cases. We judged an invariant to be potentially relevant if it expressed a property that was necessarily true of the program or expressed a salient property of its input, and if we believed that knowledge of that property would help a programmer to understand the code or perform a task. We made every effort to be fair, consistent, and objective in our assessments.

This section reports what percentage of the reported invariants are relevant (like precision in information retrieval) and what percentage of desired invariants are reported (like recall in information retrieval). We do not report the latter measurement for the student programs because of the spotty quantity and quality of their formal specifications, and because it would be less compelling for us to report detecting precisely the invariants that we had ourselves come up with. The section also illustrates some of the facts discovered by Daikon, as space permits.

class	relevant	implied	irrelevant	% relevant
`LinkedList`	317	11	1	96
`OrderedList`	201	5	5	95
`StackLi`	184	8	1	95
`StackAr`	159	0	0	100
`QueueAr`	500	0	0	100
`ListNode`	46	1	1	95
`LinkedListItr`	185	8	0	95

Figure 2: Invariants computed over data structures [Wei99]. `ListNode` and `LinkedListItr` are used internally by the first three data structures. The "relevant" column lists the number of reported invariants that were relevant. The "implied" column counts the number of invariants that were implied by other invariants (a simple test could eliminate these [ECGN00]). The "irrelevant" column counts the number of reported invariants that were irrelevant. The "% relevant" column is the ratio of the relevant invariants to the total number of reported invariants. These numbers do not include class invariants that were repeated at method entries and exits and tautological invariants that are necessarily true based on the subparts of variables being compared, all of which were removed by an automated postprocessing step.

5.1 Textbook data structures

We ran Daikon over the first five data structures in a data structures textbook [Wei99]: linked lists, ordered lists, stacks represented by lists, stacks represented by arrays, and queues represented by arrays. Before examining the reported invariants, we determined the desired output by reading the book and the programs.

The textbook's implementation comprises seven classes, ranging in size from a dozen to 65 lines of non-blank, non-comment code. (For comparison, the Sun JDK 1.2.2 `java.util.LinkedList` class is implemented in 673 lines, of which 255 are non-comment and non-blank. Its interface is richer but its functionality is essentially the same as the textbook's `LinkedList` class.) Because the provided test suites are minimal, we augmented them with additional test cases. Previous work describes how we detect and take advantage of polymorphic variables that have specific runtime types [ECGN00], so for simplicity we assume here that the polymorphic data structures are rewritten to contain Integer objects.

Figure 2 indicates the effectiveness of Daikon as applied to the five textbook data structures. For each of the classes in these programs for the selected test suite, Daikon reported at least 95% of the relevant invariants. For example, `LinkedList` reported 1 irrelevant invariant and 11 invariants that were implied by other reported invariants, meaning that 96% of the reported invariants were relevant. Daikon also never reported fewer than 98% of the expected invariants. For `LinkedList`, 3 manually identified relevant invariants were missed and 317 were reported, meaning that 99% of the manually identified invariants were found by Daikon.

Figure 3 shows the data from this process. The small

class	relevant	missing	% reported
LinkedList	317	3	99
OrderedList	201	5	98
StackLi	184	0	100
StackAr	159	0	100
QueueAr	500	10	98
ListNode	46	0	100
LinkedListItr	185	2	99

Figure 3: Missing invariants over data structures [Wei99]. The "relevant" column is repeated from Figure 2. The "missing" column counts desired invariants that Daikon failed to report. The "% reported" column is the ratio of relevant to the sum of relevant and missing.

numbers in the "relevant missing" column and the relatively larger numbers in the "relevant reported" column indicate that Daikon reports most relevant invariants and omits few relevant invariants. The qualitative analysis below will provide additional examples and details.

5.1.1 Qualitative analysis

The following subsections qualitatively assess the invariants detected on these five data structures, providing examples of the output of the system on the various test cases. For brevity, we discuss each invariant only once, even if it was detected at multiple program points.

We can classify each relevant detected invariant as a functional invariant or a usage property. Functional invariants depend only on the structure of the code; they are universally true of the data structure under test. Examples are traditional object or class invariants, or function preconditions, postconditions, or loop invariants. Usage properties result from a program's specific use of a data structure. Although these invariants do not provide information about the class per se, they can ease understanding of complex abstractions that are used in simpler ways or can demonstrate the adequacy or inadequacy of test suites, and we have seen concrete examples where they have helped programmers for just these reasons [ECGN].

Linked lists. Figure 4 illustrates some of the invariants detected over linked lists. Linked lists are implemented with a header node that is not part of the list proper; the class invariants indicate that there is always at least one `ListNode` reachable from `header`, which is another way of asserting that `header` is not null. Additionally, `header.element` is always set to 0. The reason is that there were only 32 different linked lists created by the tests, and so only 32 different values for `header`; that is not enough samples to justify the inequality. When we reran the experiment with a larger test suite, the invariant was reported.

Most of the `findPrevious` entry invariants are usage properties dependent on the particular program and test suite: elements are random integers between 0 and 31, and the maximum list size is 15. The first equality indicates that this program always inserts at the beginning of the list. The routine's exit invariants indicate that it does not change `x`, `header`, or objects accessible from `header` or their fields. In

```
LinkedList:::CLASS
  header != null
  size(header.closure(next)) >= 1
  header.element = 0
LinkedListItr
    LinkedList.findPrevious(Object x):::ENTER
  p.current = header
  x <= 31
  x >= 0
  size(header.next.closure(next)) <= 15
LinkedListItr
    LinkedList.findPrevious(Object x):::EXIT
  x = orig(x)
  header = orig(header)
  header.closure(next) =
      orig(header.closure(next))
  header.closure(next).element =
      orig(header.closure(next).element)
  return != null
  return.current != null
  x != return.current.element
  return.current.closure(next)
      is a subsequence of header.closure(next)
  MISSING: return.current.next.element = x
void LinkedList.insert(Object x,
                       LinkedListItr p):::EXIT
  x = header.next.element
  if (p != null && p.current != null)
    then size(orig(header.next.closure(next))) =
            size(header.next.closure(next)) - 1
    else header.closure(next) =
            orig(header.closure(next))
boolean LinkedList.isEmpty():::EXIT
  if (header.next == null)
    then return = true
    else return = false
void LinkedList.remove(Object x):::EXIT
  size(header.next.closure(next)) <=
      size(orig(header.next.closure(next)))
  MISSING: if (findPrevious(s) != null)
    then size(header.next.closure(next)) =
      size(orig(header.next.closure(next))) - 1
    else size(header.next.closure(next)) =
      size(orig(header.next.closure(next)))
```

Figure 4: Linked list invariants. Invariants annotated by `:::CLASS` are valid at the entry and exit of every public method; they roughly correspond to class invariants or object invariants for the given class. The notation `closure(`*fieldname*`)` stands for the collection of objects reachable by following *fieldname* pointers; `orig(`*val*`)` stands for the value of *val* at entry to a procedure; and `size(`*val*`)` stands for the size of array or collection *val*.

other words, the routine is side-effect-free. Additionally, the routine always succeeds for this test suite: the return value is never null. The antepenultimate exit invariant indicates that the argument and the return value reference different values. The penultimate one indicates that the return value points into the original list, and the final (missing) invariant is the basic contract of the procedure. Daikon can find that invariant; see the discussion of ordered lists for details.

The first `insert` invariant indicates that, for this program, insertion always occurs at the beginning of the list. The second one shows the conditions for `p` under which insertion is successful. No size invariant is reported for the case of unsuccessful insertion because the equality of the two collections implies that they have the same size; that redundant invariant is automatically suppressed.

The `remove` invariant is an inequality over sizes because

```
OrderedList:::CLASS
  header.closure(next).element sorted by <=
  MISSING: header.next.closure(next).element
           sorted by <
void OrderedList.insert(Integer x):::EXIT
  size(header.next.closure(next)) >=
  size(orig(header.next.closure(next)))
```

Figure 5: Ordered list invariants.

```
void StackLi.push(Object x):::EXIT
  x = topOfStack.element
  topOfStack.next = orig(topOfStack)
  topOfStack.next.closure(next) =
      orig(topOfStack.closure(next))
  size(topOfStack.closure(next)) =
      size(orig(topOfStack.closure(next))) + 1
Object StackLi.topAndPop():::EXIT
  return = orig(topOfStack.element)
  topOfStack = orig(topOfStack.next)
  topOfStack.closure(next) =
      orig(topOfStack.next.closure(next))
  size(topOfStack.closure(next)) =
      size(orig(topOfStack.closure(next))) - 1
```

Figure 6: Stack invariants for list representation.

deletion is not always successful; additionally, it is not found to always occur at the beginning of the list, as was the case for insertion. We do not, however, detect the exact predicate for determining when deletion was successful. The reason is that we currently split only on conditions in the program (however, the current prototype does examine local variables, which often appear in conditionals) and zero-argument boolean member functions. If we also used unary boolean member functions, we could get the desired condition. (It would be more perspicuous if there were an `isMember` function, but the class lacks that.) Splitting on local variable `s` would also solve the problem.

Ordered lists. Most invariants over ordered lists are identical to those for arbitrary linked lists; some differences appear in Figure 5. The one additional class invariant is the expected one, indicating that the linked list pointed to by the header is always sorted in terms of `element` values. The $\leq$ relationship results from the header element's value of 0, for the first element may also be 0. The strict ordering relation over the list proper is missed only because the default depth for outputting data structures does not derive the needed variable. When we increased the depth (a command-line argument to the instrumenter) by 1, the desired invariant is produced, and likewise for `LinkedList.findPrevious`.

Insertion does not always occur because `OrderedList` permits no duplicate values, and insertion does not always occur at the list head; this eliminates some invariants that appeared in `LinkedList`. We fail to find the condition predicate over the size of `insert`'s result for the same reasons we missed the similar predicate for `LinkedList.remove` above.

Stacks: list representation. The three invariants for stack `push` completely capture the operation's semantics (see Figure 6), and the `pop` invariants are symmetric. The fourth invariant for `push`, indicating that the stack grows by one after

```
StackAr():::EXIT
  theArray=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  topOfStack = -1
boolean StackAr.isEmpty():::EXIT
  topOfStack >= 0
  return = false
void StackAr.push(Object x):::EXIT
  x = theArray[topOfStack]
  topOfStack >= 0
  orig(topOfStack) = topOfStack - 1
```

Figure 7: Stack invariants for array representation.

an insertion, does not explicitly appear in the output. Instead, the output includes the invariants

```
  size(topOfStack.next.closure(next)) =
            size(topOfStack.closure(next)) - 1
  topOfStack.next.closure(next) =
            orig(topOfStack.closure(next))
```

and these trivially imply the fourth invariant. This is an artifact of how we choose a canonical variable among equal variables; the symmetric invariant reported for `push` was found explicitly. (It may be hard to find a desired invariant in the current prototype's output; we have developed a simple tool that lists all the invariants involving a specified set of variables.)

The first invariant at the exit of the `topAndPop` method captures the return of the top stack element. The second and third capture the notion of popping the stack. And the final invariant indicates that the method decreases the size of the linked list by one.

Stacks: array representation. Figure 7 shows invariants for a stack implemented by an array. The invariants for the exit point of the constructor `StackAr` show the initialization of the stack: all elements are zeroed and the top index takes an initial value. The `isEmpty` invariants reflect a shortcoming of the test suite: no tests were done when the stack was empty. (Similarly, the stack never filled, which would result in more disjunctive invariants.) That the data structure is a stack is captured in the invariants on `push` (`pop` is very similar): the element is inserted at the top of the stack, the top index is incremented, and the stack is non-empty after a push.

Queues. Invariants inferred over queues implemented with an array representation appear in Figure 8. The constructor postconditions capture the initializations of the internal representation of the queue. The invariant currentSize = front is accurate but coincidental, since in principle the initial length is always zero but another index could have been used to represent the front of the queue.

Collectively, the invariants at the exit of `QueueAr` say that the proper element is returned, the back index changes, and the front index remains unchanged. The invariant over the back index is accurate but too weak: we would prefer to find the missing modulus invariant. Adding differences and sums of variables as derived variables would cause this invariant to be detected (in a slightly different form). Our original prototype did so, but at the time of these experiments we had

```
QueueAr():::EXIT
  currentSize = 0
  currentSize = front
  back = 15
  theArray=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Object QueueAr.dequeue():::EXIT
  return = theArray[front-1] =
           theArray[orig(front)]
  front = orig(front) + 1
  back = orig(back)
void QueueAr.enqueue(Object x):::EXIT
  x = theArray[currentSize-1] = theArray[back]
  back != orig(back)
  front = orig(front)
  currentSize = orig(currentSize) + 1
  MISSING: back = ((orig(back) + 1) mod
                   DEFAULT_CAPACITY)
```

Figure 8: Queue invariants.

Student	relevant	implied	irrelevant	missing	added
1	15	37	72	4	0
2	13	24	27	10	1
3	19	25	64	5	3
4	13	30	48	10	7

Figure 9: Object invariants detected for 3 key classes in student programs. The "relevant" column is the number of relevant invariants reported that also appear in the students' formal specifications. The "implied" column is the number of redundantly reported invariants that are implied by other (relevant) ones. The "irrelevant" column is the number of reported invariants that are not relevant. The "missing" column is invariants in the students' formal specifications that did not appear in Daikon's output. The "added" column is relevant invariants detected by Daikon that the students erroneously omitted from their formal specifications.

not yet added that to the current version, as we were unsure of its general applicability.

5.2 City map student programs

To further verify our techniques and implementation, we ran Daikon on programs written by students in a software engineering course at MIT. The students were assigned to write code, formal specifications, and test cases for a city map implemented in terms of a weighted digraph. Although a weighted digraph is inherently recursive, no student implemented it via a recursive data structure. The implementations tended to use tables indexed by graph nodes; these were sometimes nested and sometimes not, and the indexed data varied as well. This general approach to representing collections is not surprising given the availability of a library of efficiently implemented abstractions. More explicit representations of collections (for instance, as resizeable arrays) lessen the pressure for Daikon to handle recursively-defined collections; however, other properties (such as connectedness in the resulting graph) may become more difficult to infer.

The students were directed to achieve branch coverage with their test suites; most of them stopped immediately upon achieving that goal, so many methods were executed just a few times by their tests. Thus, at many program points, no invariants were statistically justified. To report results consistent across the programs, we report object invariants for three classes (`Table`, `WtDigraph`, and `DistanceChart`) that appear in all the student programs. (An object invariant holds at entry to and exit from all of the class's public methods.)

Assessing the Daikon output for these programs is far more difficult than for the data structures considered above. The primary reason is that it would be extremely time-consuming to enumerate the true set of relevant invariants for each program. One reason for the high cost is that the programs range from 1500–5000 lines of code; another reason is that each program would require custom analysis to determine the true set of invariants. In any case, we could not do these analyses impartially, since we examined the Daikon output without having done such an analysis beforehand. The students provided formal specifications, but we found that they were often inadequate, failing to note important properties of the code.

Nonetheless, the estimates and the related qualitative analysis are valuable as part of our assessment. One reason is that they provide added insight in and of themselves. Additionally, serendipitously finding *some* properties about a program may be more important than finding them all, especially if the reported properties might have been overlooked by the programmer otherwise. (An early simple use of Daikon showed the value of providing programmers with unexpected, albeit incomplete, information [ECGN].)

Figure 9 quantifies Daikon's output for the four student programs we assessed in detail. Because of the absence of a true set of invariants against which to compare Daikon's output, the data we report in this figure differs somewhat from that of Figures 2 and 3. For example, the "missing" numbers compare against student formal specifications and so are underestimates of the number of true invariants missed, because in most cases the students' formal specifications were incomplete, omitting important properties that were maintained by their code. (In at least one case, a comment also indicated that a representation invariant held at a point in the constructor where it had not yet been established.)

We now discuss the columns of the table in turn; we explicitly address the apparently large number of "irrelevant" invariants, especially as compared to the data structure programs analyzed above.

The invariants listed as "relevant" were both stated by the students and also found by Daikon. They include uniqueness and non-nullness of collection elements, constraints over data structure sizes, and the like.

The "added" invariants were missing from the student formal specifications but discovered by Daikon. As an example, some implementations of the `Table` abstraction used parallel `keys` and `values` arrays; Daikon always reported $\mathsf{size}(\mathsf{keys}) = \mathsf{size}(\mathsf{values})$, but some formally specified representation invariants omitted that property. Similarly, some but not all students explicitly noted that certain values should be non-null, a property that was also picked up by Daikon. (Daikon also discovered some invariants of the test suites, such as as $\mathsf{size}(\mathsf{nodes}) \geq \mathsf{size}(\mathsf{edges.keys})$, which

indicates that there were never more edges than nodes; and size(distanceChart.edges.keys) = 0 in addCity, which indicates that the function was never called after addHighway was called — that is, all cities were added before any highways.)

The "implied" invariants are redundant, because given other reported invariants, they are necessarily true. Most implied invariants are suppressed from the output; that some remain is an easily corrected weakness of our special-purpose logical implication checker.

The "irrelevant" invariants are nearly all comparisons between incompatible runtime types (for instance, a city is discovered to be never equal to a highway). Daikon performs these comparisons because it respects the statically declared program types, which are Object. One of two techniques would eliminate most or all of these irrelevant invariants: either using a Lackwit-like static analysis to determine that values cannot flow from one variable to the other [OJ97], or performing two stages of invariant detection, the first of which determines the actual runtime types [ECGN00]. So, applying known technologies will allow us to eliminate almost all of the "implied" and "irrelevant" invariants, which will make the Daikon output far cleaner.

The "missing" invariants fall into several categories. First, invariants stating that the graph is bidirectional or contains no self-edges require existential quantifiers or other mechanisms not currently supported by Daikon. These invariants are easy to state informally, but the programs' self-checks for these properties were sometimes a dozen lines long. Second, invariants about mutability — for example, "the keys are immutable" — would require an analysis of immutability as well as run-time types. Third, several invariants over the runtime types of objects were not detected due to inadequate test suites or polymorphism. Finally, the invariant that a collection contains no duplicates can be detected by Daikon, but in practice often was not because at runtime the specified collections were very small (for instance, maximum outdegree was often two) and thus were observed too few times for Daikon's statistical tests to permit the invariant to be reported. (In other cases, Daikon did report that collections, such as the global collection of all nodes, contained no duplicates or null entries.)

Detecting the invariants for these programs was inexpensive for Daikon. The trace files ranged in size from 500KB to 15MB, depending on the test suite and program. For each of the four programs, Daikon consumed approximately two minutes of processing time. Daikon reported between 700 and 1900 invariants per program; the number of instrumented program points ranged from 60 to 80.

6 Incremental processing

The approaches outlined in this paper perform adequately for modest programs, but we do not expect them to scale without change to large programs. After briefly discussing performance issues, we outline our approach to using online, incremental invariant inference to permit scaling Daikon to larger, more realistic programs.

Instrumentation is approximately as fast as compilation. (The Daikon Java instrumenter is a modified version of the Jikes compiler.) Instrumentation slowdowns tend to be one or two orders of magnitude (smaller when data structures being traversed are smaller, as instrumentation makes every operation be at least $O(n)$ time where n is the size of the visible data). Daikon 2 (the current version) is written in Java, and processing 1000 samples for a program point with 20 variables in scope takes less than a minute on a 143MHz Sun Ultra 1 Model 140 running SunOS 5.5.1, with no JIT compiler and all debugging assertions enabled. (These numbers are typical, although they are much larger in a few cases.) We have not carefully measured time and space performance because the system is still under development and we have not seriously attempted to optimize its performance. Despite this, we have found relevant invariants in pointer-based programs, demonstrating a proof of concept.

The major hurdle to making invariant detection scale is the large number of possibly sizable variables presented by the front end to the back end. Invariant detection per se has a modest cost: its time grows with the number of invariants reported, not with the number of invariants checked [ECGN]. The former number tends to be relatively small, whereas the latter is cubic in the number of variables in scope at a program point. The reason for this good performance is that the overwhelming majority of invariants are false, and false invariants tend to be falsified very quickly [ECGN]. Only true invariants need be checked for every sample of variable values.

Large data structures are costly for the front end to traverse and output, and costly for the back end to input and examine for invariants. Additionally, wide data structures result in a large number of fields being output to the data trace.

A straightforward approach to scaling is to give programmers control over instrumentation. Daikon permits instrumentation to be suppressed for classes and functions specified via a command-line argument; likewise, users can specify detection of only class invariants, only procedure preconditions and postconditions, or both.

To make the remaining tracing and inference go faster, we can eliminate I/O costs by running the invariant detector online in cooperation with the program under test, directly examining its data structures. When testing invariants online, we need not store all data values indefinitely. Values need only be accumulated initially to permit instantiating all viable invariants in a staged fashion, then discarded. Staging permits simpler invariants to be tested first so that redundant invariants can be suppressed before being instantiated. In particular, for each variable we test, in order: (1) whether the variable is a constant or can be missing from the trace, (2) whether the variable is equal to any other variable, (3) unary invariants, (4) binary invariants with each other variable, and (5) ternary invariants with each pair of other variables.

At each step, invariants discovered earlier can suppress later ones. For instance, if two variables are equal, then one of them is marked non-canonical and not considered any further (see Section 5 for an artifact of this decision). Once invariant inference is complete for a set of variables,

then derived variables are introduced (with some of them being suppressed by the presence of invariants that illustrate they would be found equal to another variable or would be non-sensical). Then, inference is performed over the derived variables. This staging of derivation and inference, and the sub-staging of inference, is not a mere performance enhancement. The system simply does not run when they are omitted, for it is swamped in large numbers of derived variables and vast numbers of invariants to check; both memory use and runtime skyrocket.

This approach requires that when an invariant is falsified, then other invariants and variable derivations that were suppressed by its presence must be reinstated.

Beyond eliminating the I/O bottleneck and reducing memory costs, incremental processing can also reduce the amount of work performed by the instrumentation itself. When a variable is determined to be no longer of interest, then the instrumentation can be informed to stop recording its value, thus reducing its overhead. Given that most invariants are quickly falsified, we speculate that this will provide the largest speedup.

Implementation of this design is underway. Daikon supports incremental inference, but it does not yet reinstate suppressed derived variables and invariants when an invariant is falsified, and it is not integrated with the instrumented code.

7 Related work

Dynamic inference. Dynamic analysis [Bal99] is useful for a variety of tasks. Value profiling [CFE97, SS98, CFE] addresses a subset of our problem: detection of constant or near-constant variables or instruction operands. Such information can support runtime specialization where the program branches to a specialized version if a variable value is as expected. Runtime disambiguation [Nic89, SHZ+94, HSS94] has a particular focus on pointer aliasing to support optimization; for pairs of pointers that profiling shows are rarely aliased, runtime reductions of 16–77% have been realized [Nic89].

Static inference. Most pointer analysis research addresses determining alias or points-to relations. Such information can be used to compute the may definitions of an assignment in static program slicing or to verify the independence of two pointer references to enable an optimization. Precise pointer analysis is computationally difficult [LH88, LR92]. The high cost of flow-sensitive approaches [Wei80, JM81, JM82, CWZ90, HN90, LR92, HEGV93], has led to the development of flow-insensitive techniques [Ste96, And94, SH97], which are often nearly as precise for a fraction of the cost [HP98].

Shape analysis is a static analysis that infers properties of pointer structures that could be used by programmers as invariants. In particular, shape analysis produces a graph structure for a structure pointer reference that summarizes the abstract memory locations that it can reach [JM81, LH88, CWZ90, HHN92, SRW99]. ADDS, for example, uses traditional gen/kill analysis to propagate descriptions like pointer dimensionality and reachability through a program [GH96], permitting the determination of tree, dag, and cyclic properties. The necessity to summarize actual properties in static approaches is akin to our choice to limit the depth to which we derive variables. Our depth limiting is similar to the simple static approach of k-limiting.

Checking formal specifications. Considerable research has addressed statically checking formal specifications [Pfe92, DC94, NCOD97, LN98, JvH+98]. (One idea we are pursuing is to use Daikon as a hypothesis generator that can provide specifications that these kinds of systems can check; this would be an ideal synergy between static and dynamic analysis.) Recently, some realistic static specification checkers have been implemented. LCLint [EGHT94, Eva96] verifies that programs respect annotations in the Larch/C Interface Language [Tan94]; in addition to modularity properties, LCLint also checks pointer-based properties such as definedness, nullness, and allocation state. ESC [Det96, LN98, DLNS98], the Extended Static Checker, permits programmers to write type-like annotations including arithmetic relationships and declarations about mutability; it catches array bound errors, nil dereferences, synchronization errors, and other programming mistakes. Neither LCLint nor ESC is completely sound, but they do provide programmers substantial confidence in the annotations that they check.

8 Conclusion

We have extended invariant detection techniques and the Daikon prototype to enable the discovery of invariants involving collections of data. A key technique is a linearization process that traverses an implicitly represented collection, recording that collection's elements in an array in a program trace. The Daikon invariant detector, which can handle scalars and arrays, can then infer properties over the linearized versions of the collections. Furthermore, we have developed a technique for inferring conditional invariants by splitting trace data based on predicates extracted from the source text. While this paper applies conditional invariants to recursive data structures, such invariants are more broadly applicable as well.

Our initial experience demonstrates the feasibility and potential both of dynamic inference of invariants over collections and conditionals and also of our implementation approach.

The Daikon invariant detector is available for download from `http://www.cs.washington.edu/homes/mernst/daikon/`.

Acknowledgments

Jake Cockrell and Adam Czeisler made many contributions to this project. Daniel Jackson and Nick Mathewson shared students programs from MIT's 6.170 course. This research was supported in

part by NSF grant CCR-9970985, an IBM Graduate Fellowship, and gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation. Griswold is currently on leave from UCSD at Xerox PARC. Kataoka is currently visiting UW from the System Engineering Laboratory, Research and Development Center, Toshiba Corporation.

References

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *ESEC/FSE*, pages 216–234, September 6–10 1999.

[CFE] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*. To appear.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *MICRO-97*, pages 259–269, December 1–3, 1997.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, White Plains, NY, June 20-22, 1990.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE*, pages 62–75, December 1994.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18 1998.

[ECGN] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*. To appear. A previous version appeared in *ICSE*, pages 213–224, May 19–21, 1999.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, Limerick, Ireland, June 7-9, 2000.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *FSE*, pages 87–97, December 1994.

[Eva96] David Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, May 21–24, 1996.

[GH96] 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*, pages 1–15, January 21–24, 1996.

[HEGV93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural alias analysis framework for C compilers. ACAPS Technical Memo 72, McGill University School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems Group, Montreal, Quebec, July 24, 1993.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *PLDI*, pages 249–260, June 17–19, 1992.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[HP98] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *5th International Symposium on Program Analysis, SAS'98*, pages 57–81, September 1998.

[HSS94] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *ISCA*, pages 200–210, Chicago, Illinois, April 18–21, 1994.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, chapter 4. Prentice-Hall, 1981.

[JM82] Neil D. Jones and Steve S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74, January 1982.

[JvH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes. In *OOPSLA*, pages 329–340, Vancouver, BC, Canada, October 18-22, 1998.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI*, pages 21–34, Atlanta, Georgia, June 1988.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction '98*, pages 302–305, Lisbon, April 1998.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, 1992.

[NCOD97] Gleb Naumovich, Lori A. Clarke, Leon J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *ICSE*, pages 594–595, May 1997.

[Nic89] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[Pfe92] Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.

[SH97] Marc Shapiro and Susan Horwitz. The effects of precision on pointer analysis. In *Fourth International Symposium on Program Analysis*, September 1997.

[SHZ+94] Bogong Su, Stanley Habib, Wei Zhao, Jian Wang, and Youfeng Wu. A study of pointer aliasing for software pipelining using run-time disambiguation. In *MICRO-97*, pages 112–117, November 30–December 2, 1994.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, January 20–22, 1999.

[SS98] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. *ACM SIGPLAN Notices*, 33(11):35–45, November 1998.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, January 21–24, 1996.

[Tan94] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report MIT-LCS//MIT/LCS/TR-619, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1994.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *POPL*, pages 83–94, January 1980.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.