Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java

Jeremy W. Nimmer and Michael D. Ernst

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
Email: {jwnimmer, mernst}@lcs.mit.edu

Abstract

This paper shows how to integrate two complementary techniques for manipulating program invariants: dynamic detection and static verification. Dynamic detection proposes likely invariants based on program executions, but the resulting properties are not guaranteed to be true over all possible executions. Static verification checks that properties are always true, but it can be difficult and tedious to select a goal and to annotate programs for input to a static checker. Combining these techniques overcomes the weaknesses of each: dynamically detected invariants can annotate a program or provide goals for static verification, and static verification can confirm properties proposed by a dynamic tool.

We have integrated a tool for dynamically detecting likely program invariants, Daikon, with a tool for statically verifying program properties, ESC/Java. Daikon examines run-time values of program variables; it looks for patterns and relationships in those values, and it reports properties that are never falsified during test runs and that satisfy certain other conditions, such as being statistically justified. ESC/Java takes as input a Java program annotated with preconditions, postconditions, and other assertions, and it reports which annotations cannot be statically verified and also warns of potential runtime errors, such as null dereferences and out-of-bounds array indices.

Our prototype system runs Daikon, inserts its output into code as ESC/Java annotations, and then runs ESC/Java, which reports unverifiable annotations. The entire process is completely automatic, though users may provide guidance in order to improve results if desired. In preliminary experiments, ESC/Java verified all or most of the invariants proposed by Daikon.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Static and dynamic analyses have complementary strengths and weaknesses, so combining them has great promise. Static analysis operates by examining program source code and reasoning about possible executions. It builds a model of the state of the program, such as values for variables and other expressions. Static analysis can be conservative and sound; however, it can be inefficient, can produce weak results, and can require explicit goals or annotations. Dynamic analysis obtains information from program executions; examples include profiling and testing. Rather than modeling the state of the program, dynamic analysis uses actual values computed during program executions. Dynamic analysis can be efficient and precise, but the results may not generalize to future program executions. Our research integrates static and dynamic analysis to take advantage of their complementary strengths: dynamic analysis can propose program properties to be verified by static analysis.

This paper focuses on analyses over program invariants. A program invariant is a property that is true at a particular program point or points, such as might appear in an `assert` statement or a formal specification. Invariants include procedure preconditions and postconditions, loop invariants, and object (representation) invariants. Examples include $\mathsf{y} = 4 * \mathsf{x} + 3$; $\mathsf{x} > \mathsf{abs}(\mathsf{y})$; array a contains no duplicates; $\mathsf{n} = \mathsf{n.child.parent}$ (for all nodes n); $\mathsf{size}(\mathsf{keys}) = \mathsf{size}(\mathsf{contents})$; and graph g is acyclic. Invariants explicate data structures and algorithms and are helpful for programming tasks from design to maintenance. Invariants assist in creation of better programs [30,46,35,34], document program operation [39,45], assist testing and enable correct modification [52,29], assist in test-case generation [59] and validation [7], form a program spectrum [1,55,31], and can enable optimizations [6], among other uses. Despite their advantages, invariants are usually missing from programs.

Dynamic invariant detection is a technique for postulating likely invariants from program runs: a dynamic invariant detector runs the target program, examines the values that it computes, and looks for patterns and relationships over those values, reporting the ones that are always true over an entire test suite and that satisfy certain other conditions (see Section 2.1). The outputs are likely invariants: they are not guaranteed to be universally true, because the test suite might not characterize all possible executions of the program.

Static invariant verification is a technique for checking program properties. Given a program and a set of properties over that program, the verifier reports which properties are guaranteed to be true for all executions. Unverified properties might or might not be universally true. Static verifiers can operate by dataflow analysis, theorem proving, model checking, or other techniques. Users of static verifiers must annotate their programs with the properties to be proved (and other properties on which those might depend).

Combining dynamic invariant detection with static verification has benefits for both users of invariant detectors and users of static checkers. Because the output of a dynamic invariant detector is not guaranteed to be sound, programmers may be reluctant to use it, and its output cannot be fed into other tools that require sound input. A static verifier can indicate which proposed invariants are guaranteed to be true. Users can filter out unverified invariants so that the results are sound or can use the verifications as a first approximation when determining which dynamically detected properties are functional invariants and which are usage properties — both of which are useful, but for different tasks.

Users of static verifiers benefit from decreased annotation burden. Static verification often requires extensive annotations or intermediate assertions and goals. Automatic annotation relieves users of the burden of annotating programs from scratch — a task few enjoy or are good at. Dynamically detected invariants can also indicate properties programmers might otherwise have overlooked.

We have started to explore these benefits by integrating a dynamic invariant detector, Daikon [16,17], with a static verifier, ESC/Java [14,44]. Our system operates in three steps. First, it runs Daikon, which outputs a list of likely invariants obtained from running the target program over its test suite. Second, it inserts those invariants into the target program as annotations. Third, it runs ESC/Java on the annotated target program to report which of the likely invariants can be statically verified and which cannot. Section 4 gives more details about this process. All three steps are completely automatic, though users may provide guidance in order to obtain better results if desired. Users may edit and re-run test suites when deficiencies are found, or may add or remove specific program annotations by hand.

The remainder of this paper is organized as follows. Section 2 provides background on the dynamic invariant detector and static verifier used by our system. Section 3 presents results from several experiments. Section 4 describes how we integrated these tools, and Section 5 discusses problems that arose while building and running our system. Finally, Section 6 relates our results to other research, Section 7 proposes followon research, and Section 8 concludes.

2 Background

2.1 Daikon: Invariant discovery

Dynamic invariant detection [16,17] 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

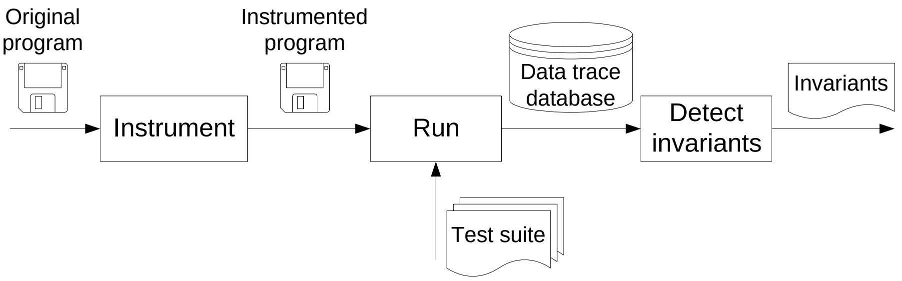

Fig. 1. An overview of dynamic detection of invariants as implemented by Daikon.

such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. The Daikon invariant detector is language independent, and currently includes instrumenters for C++ and Java.

Daikon detects invariants at specific program points such as 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 one, two, or three traced variables.

For scalar variables x, y, and z, and computed constants a, b, and c, some examples of checked invariants are: equality with a constant ($\mathsf{x} = \mathsf{a}$) or a small set of constants ($\mathsf{x} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$), lying in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), linear relationships ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), ordering ($\mathsf{x} \leq \mathsf{y}$), and functions ($\mathsf{x} = \mathsf{fn}(\mathsf{y})$). Invariants involving a sequence variable include minimum and maximum sequence values, lexicographical ordering, element ordering, invariants holding for all elements in the sequence, or membership ($\mathsf{x} \in \mathsf{y}$). Given two sequences, some example checked invariants are elementwise linear relationship, lexicographic comparison, and subsequence relationship.

In addition to local invariants such as $\mathsf{node} = \mathsf{node.child.parent}$ (for all nodes), Daikon detects global invariants over pointer-directed data structures, such as $\mathsf{mytree\ is\ sorted\ by} \leq$ by linearizing graph-like data structures. Finally, Daikon can detect conditional invariants that are not universally true, such as "$\mathsf{if\ p} \neq \mathsf{null\ then\ p.value} > \mathsf{x}$" and "$\mathsf{p.value} > \mathsf{limit\ or\ p.left} \in \mathsf{mytree}$". Conditional invariants result from splitting data into parts based on the condition and comparing the resulting invariants; if the invariants in the two halves differ, they are composed into a conditional invariant [19].

For each variable or tuple of variables in scope at a given program point, 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. Daikon maintains acceptable performance as program size increases because false invariants tend to be falsified quickly, so 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, properties of aggregates, and other values not stored in program variables, 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 it 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 $\mathsf{i} < \mathsf{size}(\mathsf{A})$ 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 [18].

The Daikon invariant detector is available for download from `http://sdg.lcs.mit.edu/daikon/`.

2.2 ESC: Static checking

ESC [13,14,43] is an Extended Static Checker that has been implemented for Modula-3 and Java. It statically detects common errors that are usually not detected until run time, such as null dereference errors, array bounds errors, and type cast errors.

ESC is intermediate in both power and ease of use between typecheckers and theorem-provers, but it aims to be more like the former and is lightweight by comparison with the latter. Rather than proving complete program correctness, ESC detects only certain types of errors. Programmers must write program annotations, many of which are similar in flavor to `assert` statements, but they need not interact with the checker as it processes the annotated program. ESC issues warnings about annotations that cannot be proven and about potential run-time errors.

ESC performs modular checking: it checks different parts of a program independently and can check partial programs or modules. It assumes that specifications for missing or unchecked components are correct. ESC's implementation uses a theorem-prover internally. We will not discuss ESC's checking strategy in more detail because this research treats ESC as a black box (it is distributed in binary form).

ESC/Java is a successor to the previous ESC/Modula-3. ESC/Java's annotation language (see Section 4.2) is simpler, because it is slightly weaker. This is in keeping with the philosophy of a tool that is easy to use and useful to programmers rather than one that is extraordinarily powerful but so difficult to use that programmers shy away from it.

This research uses ESC not only as a lightweight technology for detecting a restricted class of runtime errors, but also as a tool for verifying representation invariants. We chose to use ESC because we are not aware of other equally capable technology for statically checking properties of runnable code. Whereas many other verifiers operate over non-executable specifications or models, our research aims to combine dynamic and static techniques over the same code artifact. Furthermore, we wished to explore the limits of what invariants can be dynamically detected and statically verified. In any event, good representation invariants are often required to determine that variables are non-null and array accesses are within bounds.

Both versions of ESC are publicly available from `http://research.compaq.com/SRC/esc/`.

3 Experiments

This section gives both quantitative and qualitative results from several experiments with statically verifying dynamically detected invariants. Sections 3.1 and 3.2 discuss in detail two examples taken from a data structures textbook [61]; these sections characterize the generated invariants and provide an intuition about the output of our system. Section 3.3 overviews other experiments and highlights the types of problems the system may encounter.

3.1 StackAr: array-based stack

The `StackAr` example is an array-based stack implementation [61]. The source contains 40 non-comment lines of code in seven methods, along with comments which describe the behavior of the class but do not mention its representation invariant.

Our system determined the representation invariant, method preconditions, modification targets, and postconditions, and statically proved that these properties hold. Without these annotations, ESC issues warnings about many potential runtime errors. With the addition of the detected invariants, ESC successfully checks that the `StackAr` class avoids runtime errors, meets its specification, and maintains important properties during execution.

Figure 2 shows that the Daikon invariant detector finds 88 invariants: 6 object invariants, 5 requires clauses (method preconditions), 3 modifies clauses (modification targets), and 74 ensures clauses (method postconditions). How-

	Expressible		Inexpressible		
	Unique	Redun.	Unique	Redun.	Total
Object	6	0	0	0	6
Requires	4	0	0	1	5
Modifies	3	0	0	0	3
Ensures	17	40	0	17	74
Total	30	40	0	18	88

Fig. 2. Invariants detected by Daikon in the `StackAr` program. The table classifies the invariants by expressibility (whether it can be stated in the ESCJML language; see Section 4.2) and redundancy (whether it is logically implied other invariants). Our system discovered and proved 70 invariants, of which 30 were non-redundant.

ever, 18 of the invariants were inexpressible in ESC (see Section 4.2). Also, 58 invariants were implied by other other invariants and could have been removed by improved redundancy checks in Daikon (see Section 7). Finally, our system heuristically added 2 annotations involving the owner of the array (see Section 4.3).

Figure 3 shows part of the automatically-annotated source code for `StackAr`. The first six annotations describe the representation invariant. The array is never null, and its runtime type is `Object[]`. The `topOfStack` index is at least -1 and is less than the length of the array. Finally, the elements of the array are non-null if their index is no more than `topOfStack` and are null otherwise.

The next four annotations describe the specification for the constructor. If the capacity is non-negative on entry, then on exit the array length matches the given capacity, the `topOfStack` index indicates an empty stack, and all elements of the array are null. (The final assertion is redundant: it is implied by the representation invariant.)

In addition to proving the absence of errors, our system generated specifications for all operations of the class, and verified that the implementation met the specification. For example, two postconditions for the `topAndPop` method were:

```
/*@ ensures (\old(topOfStack) == -1) == (\result == null) */
/*@ ensures (\old(topOfStack) >= 0) == (\result != null) */
```

These invariants state that `topAndPop` returns `null` if and only if the stack is empty upon entry.

The assertions for a method provide a partial specification, but do not necessarily give a full input-output relation. The specifications derived from detected invariants are useful for several reasons.

First, users can understand the behavior of a method by reading the specifications instead of reasoning about the implementation. Similarly, static tools

```
public class StackAr
{
/*@ invariant this.theArray != null */
/*@ invariant \typeof(this.theArray) == \type(java.lang.Object[]) */
/*@ invariant this.topOfStack >= -1 */
/*@ invariant this.topOfStack <= this.theArray.length-1 */
/*@ invariant (\forall int i; (0 <= i && i <= this.topOfStack)
                              ==> (this.theArray[i] != null)) */
/*@ invariant (\forall int i; (this.topOfStack+1 <= i &&
          i <= this.theArray.length-1) ==> (this.theArray[i] == null)) */

  public StackAr( int capacity )
/*@ requires capacity >= 0 */
/*@ ensures capacity == this.theArray.length */
/*@ ensures this.topOfStack == -1 */
/*@ ensures (\forall int i; (0 <= i && i <= this.theArray.length-1)
                            ==> (this.theArray[i] == null)) */
  {
    theArray = new Object[ capacity ];
    topOfStack = -1;
    /*@ set theArray.owner = this */
  }

  ...

  /*@ spec_public */ private Object [ ] theArray;
  /*@ invariant theArray.owner == this */
  /*@ spec_public */ private int        topOfStack;

  ...

}
```

Fig. 3. The object invariants, first method, and field declarations of the annotated `StackAr.java` file [61]. The JML annotations (comments starting with "`/*@`") are produced automatically by Daikon, are automatically inserted into the source code by our system, and are automatically verified by ESC/Java.

can check the assertions, and can use the (checked) assertions to perform reasoning about calling code. Furthermore, programmers modifying existing code may be aided by knowledge of existing invariants which the code preserves. They may check that specifications previously generated and proved over the unmodified program still hold true over the new source. Finally, the invariants explicate potentially important properties of the implementation. For example, the representation invariant on `StackAr` guarantees that unused array elements are set to null. Thus, objects popped from the stack are not prevented from being garbage collected.

	Expressible			Inexpressible		
	Unique	Redun.	Unver.	Unique	Redun.	Total
Object	4	1	0	0	0	5
Requires	14	5	1	0	0	20
Modifies	2	0	0	0	0	2
Ensures	14	40	1	7	55	117
Total	34	46	2	7	55	144

Fig. 4. Invariants detected by Daikon in the `DisjSets` program. The table classifies the invariants by expressibility (whether it can be stated in the ESCJML language), redundancy (whether it is logically implied other invariants), and verifiability (whether ESC was able to verify it). Our system discovered and proved 80 invariants, of which 34 were non-redundant. Two coincidental invariants due to specifics of the test suite could not be proved.

3.2 DisjSets: union-find disjoint sets

A second example further illustrates our results, and provides an example of invariants which could not be verified.

The `DisjSets` class is an array-based implementation of disjoint sets, which partition a range of integers into disjoint subsets that support the `union` and `find` operations [61]. The source contains 30 non-comment lines of code in four methods, along with comments which describe the behavior of the class but do not mention its representation invariant. Our system determined the representation invariant, method preconditions, modification targets, and postconditions, and statically proved that most of these properties hold.

Figure 4 shows that Daikon found 144 invariants over the class; 62 of the invariants were not expressible in ESC, and 46 of the remaining ones were redundant. Again, 2 annotations involving the owner of the array were added by a heuristic. ESC proved 80 of the 82 expressible invariants, and it warned about two (unprovable) test suite artifacts.

The unprovable invariants were coincidences of the test suite used to detect invariants. In the `DisjSets` implementation, `s` is the integer array used to represent the sets; `s[i]` references another integer in the same set, or is `-1` if the element is the leader of its set. For the `union` operation, Daikon reported the following precondition:

```
/*@ requires s[s.length-2] < s.length-1 */
```

This invariant states that the neighbor of the penultimate element is never the last element. The test cases did not include a case where the penultimate element was added to the set of the last element, so this assertion was true given the input data, but is not true in general. Tests which contradict this assertion could be added to the suite, but are arguably not of general utility.

3.3 Other experiments

We have run our system on seven other examples, primarily chosen from textbooks and from staff solutions to assignments in a programming course at MIT. We selected these particular programs because they contain interesting, nontrivial representation invariants that are not obviously beyond the capabilities of ESC. Our system was not able to verify all the detected invariants for these other programs (as for `StackAr`). Section 5 discusses challenges to static verification, but we illustrate them briefly here.

We found there were three general classes of problems. First and foremost were artifacts of the test suites, which initially resulted in many irrelevant (and not universally true) invariants. For instance, integer bounds on a variables, such as $\mathsf{denom} \leq \mathsf{19719720}$, were common artifacts of the test suites. The initial test suites were unit tests that came from the textbooks or were used for grading. We speculate that unit tests, which tend to be smaller and more stylized than typical usage, throw off Daikon's statistical justification tests (see Section 2.1), which seem to work well when running system tests [17].

The second class of verification problems involved invariants that Daikon could not detect — missing classes of invariants. For instance, in a `negate` method for rational numbers, Daikon detected the equality of the denominators of the argument and result. Proving that property would require detecting that the numerator and denominator of the argument are relatively prime, so the `gcd` operation called by the constructor has no effect. We had previously rejected such invariants as of insufficiently general applicability. Users can easily add invariants to Daikon, however, by writing a Java class that satisfies an interface with four methods.

The third class of problems involved ESC's inability to prove certain invariants. We found that discovering the source of the second and third class of problems was easy and quick, and we had little trouble convincing ourselves of the correctness or incorrectness of the invariant or the code. By comparison, extending the unit test suites to find the interesting invariants in Daikon's output was time-consuming and tedious. In the future we will avoid starting with unit tests.

4 Implementation

This section discusses our implementation. We enhanced Daikon's invariant detection capabilities to permit it to report certain invariants (Section 4.1). To permit ESC to verify the detected invariants, they must be converted into ESC's input language (Section 4.2). Finally, some annotations are added heuristically (Section 4.3).

4.1 Daikon additions

We made several enhancements to Daikon to make its output easier for ESC to prove.

We added some invariants over sequence elements, such as comparing all elements to another variable or a constant. Such invariants were present in a previous version of Daikon [17] but had not been added to the current implementation.

We listed which variables are modified by the routine. This output can sometimes be misleading. For instance, the disjoint-set `union` method modifies `s[set2]`; but `set2` might be 0, so `s[0]` is also listed as possibly modified, even though it is never modified unless `set2` is 0. We plan to eliminate this extraneous listing by a combination of statically analyzing the method text and heuristically omitting from the modification list sometimes-modified variables that overlap with always-modified variables.

We enhanced Daikon's list of splitting criteria to consider boolean procedure return values and procedure exit points. Daikon uses these criteria to produce implications [19] by splitting data into two parts; if different invariants are true in the parts of the data, they can be combined into implications or disjunctions. Therefore, Daikon was able to report what preconditions caused a boolean function to return true or false, or what preconditions caused a certain return statement to be executed (and what other properties hold there).

Finally, we altered Daikon so that it did not report invariants from non-private methods when they were implied by an object invariant. Even though Daikon was not successful in finding all redundant invariants, this greatly reduced the number of redundant reported invariants, making them more manageable without removing any information. The output changes did not affect the provability of invariants, but did ease the interpretation of ESC's output.

4.2 ESC notation

ESC's input language is a variant of JML, the Java Modeling Language [41,42]. JML is an interface specification language that can specify the behavior of Java modules. Most relevant to our research is its ability to specify object representation invariants and method preconditions and postconditions. JML expressions are written in a syntax closely resembling Java. We use "ESC-JML" for the JML variant accepted as input by ESC/Java.

Daikon's default output language is also similar to Java, with extensions that permit certain varieties of invariant to be expressed more concisely or clearly than would be possible in Java. As a user option, Daikon can produce output in ESCJML. The differences between these formats fall into two categories. When the semantics differ because ESCJML is less convenient or concise but the languages are equally expressive, we usually convert Daikon's output to ESCJML. In cases where ESCJML cannot express concepts that

Daikon discovers and expresses in its own language, we omit those invariants when attempting verification with ESC.

4.2.1 Semantic differences

Both (full) JML and Daikon's default output format support array comprehensions such as `a[i..j]` to represent the subarray of `a` from indices `i` to `j` inclusive. Daikon also permits quantification via the expression "*`array`* `elements`"; for instance, this.s elements $\leq$ this.s.length. Daikon represents accesses to arrays, vectors, and linked lists uniformly and succinctly with subscripting notation, `a[i]`. Field accesses may be applied to sequences, indicating a sequence of the specified fields; for instance, `a[].fld` represents the sequence `a[0].fld`, `a[1].fld`, ... By contrast, ESCJML states expressions over arrays via an explicit `\forall` quantifier and cannot access vector or linked list elements.

By default, expressions in Daikon's output are assumed to hold only when their subexpressions are sensible. For instance, foo.bar = 22 in Daikon's output means "foo = null or foo.bar = 22", and a[i] > x means "i < 0 or i $\geq$ a.length or a[i] > x". A Daikon switch makes these guards explicit in the output or eliminates invariants over expressions that are sometimes nonsensical. In ESC, use of an expression like `a[i]` when `i` may not be a legal index can result in failure to verify and uninformative error messages.

Daikon's object invariants are specified to hold at entry and exit of non-private methods, whereas ESC's are required to hold at entry and exit of all methods. However, private helper methods need not require or maintain object invariants. To match semantics, we could remove Daikon's object invariants and repeat them at all appropriate method entries and exits, but we judged that to be too verbose and confusing; this prevents some true (public) object invariants from being proved by ESC.

4.2.2 Invariants inexpressible in ESCJML

Daikon and ESCJML method postconditions can indicate (via `orig()` in Daikon or `\old()` in ESCJML) that expressions should be evaluated in the pre-state. For instance, `return = orig(x)` indicates that the procedure returns the value which `x` held before the method was called, even though the procedure may have modified `x` during its execution. Daikon's `orig()` can apply to any variable, and distinguishes between array identity, array contents, and array subsequences. ESCJML's `\old()` cannot apply to array contents or to method parameters of primitive type. Furthermore, there is no way to mix expressions from the post-state within expressions in pre-state. (Some of these limitations can be worked around by tricks such as existential quantifiers, but the resulting invariants are not particularly readable.)

ESCJML annotations cannot include method calls, even ones that are side-effect-free. Daikon uses these for obtaining `Vector` elements and as predicates in implications.

Unlike Daikon, ESCJML cannot express closure operations, such as all the elements in a linked list. Properties over such collections are often the most interesting and important invariants over recursively defined data structures.

The full JML language permits method calls in assertions, `\old()` applied to primitive parameters, and `\reach()` for expressing reachability via transitive closure.

4.3 Other annotations

Our system makes private variables accessible to the specification with the `spec_public` annotation. More significantly, in each constructor it sets the `owner` ghost field of each non-primitive field to the object being constructed. This states that the contents of the field are not aliased by other objects. Without this annotation, ESC reasons that the field can be arbitrarily modified at any time by another method, and very little whatsoever can be proved. Adding this annotation without source code analysis is potentially unsafe, but this discipline is very frequently followed, so it has been acceptable in our experiments to date.

5 Challenges

This section discusses challenges to static verification of dynamically detected program invariants. These challenges fall into three general categories: problems with the tools, problems with the target programs, and problems with the test suites for the target programs. In some cases we have largely solved the problems, and in other cases difficulties remain to be overcome.

5.1 Tools

Section 4.1 lists enhancements made to the Daikon invariant detector as a part of this research. As Daikon is still a prototype, we anticipate that additional changes may be required in the future, particularly as it is extended to new varieties of invariant. Also, strengthening its checks for redundant invariants will reduce the size of its output and improve comprehensibility without removing any information.

Section 4.2 noted problems with ESC's input language, a variant of JML that cannot express certain important invariants and cannot concisely and clearly express others. In some cases ESC does not appear to be strong enough to verify certain true invariants, and its error messages are occasionally cryptic. However, in general we have been pleased with ESC: it has operated effectively and efficiently. For instance, though we have not run ESC on Daikon's source code, ESC has detected at least two bugs in Daikon by failing to verify reported invariants that, upon closer inspection, were not true. (Both bugs were cut-and-paste errors: in one case, the invariant formatting routine was incorrect, and in another case, the first element of an array was being ignored.)

ESC cannot express invariants over strings, and Daikon reports few such invariants in any event. As a result, ESC cannot prove that object invariants hold at the exit from a constructor or other method that interprets a string argument, even though it can show that the invariant is maintained by other methods.

In a few cases, ESC cannot prove properties Daikon reports because the property depends on an object invariant that is beyond Daikon's scope. Users can either add such invariants by hand or delete the properties that depend on them.

5.2 Target programs

Another challenge to static verification of invariants is the fact that programs are likely to contain errors that prevent the desired invariant from being true. (Although it was never our goal, we have previously identified such errors in textbooks [30,61] and in programs used in testing research [36,56].) As an example of a likely error that we detected in the course of this project, one of the object invariants for `StackAr` states that unused elements of the stack are null; this permits objects to be garbage-collected after the stack is popped and permits earlier detection of certain types of error. The `topAndPop` operation maintains this invariant (which approximately doubles the size of its code), but the `makeEmpty` routine fails to do so — a non-obvious oversight which the implementor and clients should be appraised of.

5.3 Test suites

Dynamic invariant detection may produce properties that are true for the test suite over which the target program was run, but which are not true for arbitrary runs of the program. However, that problem is solved by integrating dynamic invariant detection with static verification. The static verifier indicates that some invariants are universally true; the others might be true but beyond the capabilities of the verifier, might be true of the context in which the program is always run, or might be accidental usage properties of the test suite. In the latter case, the reported invariants specify the unintended property of the test suite that makes it less general than it should be, so a programmer knows exactly what is wrong with, and how to improve, the test suite.

Because static verification partly solves the question of which invariants are necessarily true in all contexts, the remainder of this section only treats this problem in the absence of static verification: how difficult is it to eliminate all properties that are not universally true from the output, so that it verifies with no warnings whatsoever?

In some cases the "bad" invariants gave valuable hints about test cases that needed to be added to the test suite. For instance, in some of our experiments, certain stack operations were not performed on a completely full stack, and a queue implemented via an array was not forced to wrap around by adding and deleting more elements than its capacity. As another example of a serious oversight, a test suite's calls to a safe stack pop operation were always protected by a check whether the array was empty. The resulting invariants stated that the result was always non-null, indicating that the full functionality of the method was not being tested.

In other cases, however, eliminating the undesirable invariants was a tedious chore. It required finding a test case that falsified a particular special case that had little to do with the abstraction (it was relevant to the data structures, but not the logic, of the particular implementation). The largest problems were undesirable upper and lower bounds for variables. We speculate that Daikon's statistical tests for these particular invariants need to be adjusted. It is also possible that, since those statistical tests strive to be time- and space-efficient, they make too many approximations and do not produce an accurate result.

6 Related work

This is the first research we are aware of that has dynamically generated, then statically proved, program properties.

Dynamic analysis has been used for a variety of tasks; for instance, inductive logic programming (ILP) [54,8] produces a set of Horn clauses (first-order if-then rules) and can be run over program traces [4], though with limited success. Programming by example [12] is similar but requires close human guidance, and version spaces can compactly represent sets of hypotheses [50,33,40]. Value profiling [5,57,6] can efficiently detect certain simple properties at runtime. Event traces can generate finite state machines that explicate potential system organization or behavior [9,10]. Program spectra [1,55,31,2] also capture aspects of system runtime behavior. None of these other techniques have been as successful as Daikon in detecting invariants in programs, though many have been valuable in other domains. Many static inference techniques also exist, but space prohibits discussing them here.

There are many other techniques and tools besides ESC for statically checking formal specifications [53,15,22,13,20,51,43]. These other systems have different strengths and weaknesses than ESC, but few have the polish of its integration with a real programming language (see Section 7).

6.1 Houdini

The research most closely related to ours is Houdini, an annotation assistant for ESC/Java [24,23]. Houdini is motivated by the observation that users are reluctant to annotate their programs with invariants; it attempts to lessen the burden by providing an initial set. Houdini takes a candidate annotation set as input and computes the greatest subset of it that is valid for a particular program. It repeatedly invokes the checker and removes refuted annotations, until no more annotations are refuted. The candidate invariants are all possible arithmetic comparisons among fields (and "interesting constants" such as -1, 0, 1, array lengths, and `null`); many elements of this initial set are mutually contradictory.

Daikon's candidate invariants are richer than those of Houdini; Daikon outputs implications and disjunctions, and its base invariants are also richer, including more complicated arithmetic and sequence operations. If even one required invariant is missing, then Houdini will eliminate all other true invariants that depend on it. Houdini makes no attempt to eliminate implied (redundant) invariants, as Daikon does (reducing its output size by an order of magnitude [18]), so it is difficult to interpret numbers of invariants produced by Houdini. Finally, Houdini is not publicly available, so we cannot perform a direct comparison.

Merging the two approaches could be very useful. For instance, Daikon's output could form the input to Houdini, permitting Houdini to spend less time eliminating false invariants. (A prototype "dynamic refuter" — essentially a limited dynamic invariant detector — has been built [24], but no details or results about it are provided.) Houdini has a different intent than Daikon: Houdini does not try to produce a complete specification or annotations that are good for people, but only to make up for missing annotations and permit programs to be less cluttered; in that respect, it is similar to type inference. However, Daikon's output could perhaps be used in place of Houdini's. Invariants that are true but depend on missing invariants or are not provable by ESC would not be eliminated, so users might be closer to a completely annotated program, though they might need to eliminate some invariants by hand.

7 Future work

Section 5 listed a number of problems with our system (and with its components Daikon and ESC) that should be corrected.

Another obvious way to extend this work is to use different invariant detectors than Daikon or different verifiers than ESC. Section 6 lists some other invariant detectors. Examples of static verifiers that are connected with real programming languages include LCLint [22,20,21], ACL2 [38], LOOP [37], Java PathFinder [32], and Bandera [11].

We are currently integrating Daikon with IOA [28,27], a formal language for describing computational processes that are modeled using I/O automata [47,48,49]. The IOA toolset (`http://theory.lcs.mit.edu/tds/ioa.html`) permits IOA programs to be run and also provides an interface to the Larch Prover (LP) [25,26,58], an interactive theorem-proving system for multisorted first-order logic. Daikon will propose goals, lemmas, or intermediate assertions for the theorem prover. Side conditions such as representation invariants can enable proofs that hold in all reachable states/representations (but not in all possible states/representations). It can be tedious and error-prone for people to specify the properties to be proved, and current systems have trouble postulating them; some researchers consider that task harder than performing the proof [60,3].

We are also interested in recovering from failed attempts at static verification. Broadly speaking, verification fails because the goal properties are too strong or are too weak. Properties that are too strong may be true but beyond the capabilities of the verifier, or may not be universally true (for instance, guaranteed by the program context or artifacts of the test suite). Properties that are too weak are true, but cannot be proved by the static verifier or are not useful to it — for instance, loop invariants may need to be strengthened to be proved. We anticipate that dynamic invariant detection will propose more overly-strong invariants than overly-weak ones. When verification fails, we would like to know how to strengthen and weaken invariants in a principled way, by examining the source code, program executions, patterns of invariants, and verifier output, to increase the likelihood of successful verification.

While dynamic invariant detection has been quite successful in a number of application domains, we believe that truly successful program analysis requires both static and dynamic components. What is hard for one variety of analysis is easy for the other. Some of the properties that are difficult to obtain from a dynamic analyses are apparent from an examination of the source code, and properties that are beyond the state of the art in static analysis can be easily checked at runtime. We plan to integrate more static analysis into our system (and particularly into Daikon). The dynamic analysis need not check properties discovered by the static analysis, and the dynamic analysis can focus on statically indicated code.

8 Conclusion

We have demonstrated the feasibility of dynamically detecting, then statically verifying, program invariants. In particular, we have built a system that takes the output of the Daikon invariant detector and feeds it to the ESC static checker. To our knowledge, ours is the first system to dynamically detect and then statically prove program properties. Preliminary experiments over small programs demonstrate that Daikon is effective at proposing useful invariants and that ESC is effective at verifying those invariants.

Integrating dynamic invariant detection with static verification has benefits for both tools. Use of a static verifier to augment dynamic invariant detection overcomes a potential objection about possibly unsound output, classifies the output to permit programmers to use it more effectively, permits proven invariants to be used in contexts (such as input to certain programs) that demand sound input, and may improve the performance or output of dynamic invariant detection. As a result, more programmers can take advantage of dynamically detected invariants in a variety of contexts, directly leading to fewer bugs (by introducing fewer and detecting more), better documentation, less time wasted on program understanding, better test suites, more effective validation of program changes, and more efficient programs.

Use of dynamically detected invariants to bootstrap static verification, by annotating programs or by providing goals and intermediate assertions, will speed the adoption of static analysis tools by lessening the user burden, even if some work remains for the user. The direct effect of increased use of these tools will be the detection of more errors earlier in the software development process, statically at compile time rather than dynamically at test time (or, worse, after an application has been fielded). The indirect effect will be the production of more robust, reliable, and correct computer systems. Both visible faults and silent errors will occur less often, and it will be easier to maintain these properties during a program's life because of machine checking of conditions that program correctness depends upon.

Acknowledgments

We thank the members of the Daikon group — particularly Ben Morse, Michael Harder, and Melissa Hao — for their contributions to this project. We also had fruitful conversations with William Griswold, Josh Kataoka, Rustan Leino, Greg Nelson, David Notkin, and James Saxe. This research was supported in part by NSF grants CCR-9970985 and CCR-6891317.

References

[1] David Abramson, Ian Foster, John Michalakes, and Rok Sosič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

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

[3] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *CAV*, pages 323–335, July/August 1996.

[4] Ivan Bratko and Marko Grobelnik. Inductive learning applied to program construction and verification. In José Cuena, editor, *AIFIPP '92*, pages 169–182. North-Holland, 1993.

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

[6] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999. `http://www.jilp.org/vol1/`.

[7] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and expenmental evaluation. In *ESEC/FSE*, pages 285–302, September 6–10, 1999.

[8] William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, August 1994.

[9] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[10] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, November 1998.

[11] James Corbett, Matthew Dwyer, John Hatcliff, Corina Păsăreanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448, June 7–9, 2000.

[12] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[13] 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.

[14] 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.

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

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

[17] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001. A previous version appeared in *ICSE*, pages 213–224, Los Angeles, CA, USA, May 1999.

[18] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[19] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 16, 1999.

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

[21] David Evans. *LCLint User's Guide, Version 2.5*, May 2000. `http://lclint.cs.virginia.edu/guide/`.

[22] 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.

[23] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, February 2001.

[24] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 500–517, Berlin, Germany, March 2001.

[25] Stephen Garland and John Guttag. LP, the Larch Prover. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (Kaiserslautern, West Germany)*, volume 449 of *LNCS*. Springer-Verlag, 1990.

[26] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 31 December 1991.

[27] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[28] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory for Computer Science, 1997.

[29] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE TSE*, 26(7):653–661, July 2000.

[30] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[31] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE '98*, pages 83–90, June 16, 1998.

[32] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[33] Haym Hirsh. Theoretical underpinnings of version spaces. In *IJCAI*, pages 665–670, August 1991.

[34] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Corrigenda: "Laws of programming". *Communications of the ACM*, 30(9):771, September 1987. See [35].

[35] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987. See corrigendum [34].

[36] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, May 1994.

[37] 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.

[38] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE TSE*, 23(4):203–213, April 1997.

[39] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[40] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, Stanford, CA, June 2000.

[41] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[42] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06m, Iowa State University, Department of Computer Science, February 2000. See `www.cs.iastate.edu/~leavens/JML.html`.

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

[44] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, Palo Alto, California, October 12, 2000.

[45] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE TSE*, 16(4):432–443, 1990.

[46] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, 1986.

[47] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CA, 1996.

[48] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, Vancouver, BC, Canada, August 1987.

[49] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[50] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, December 1978. Stanford University Technical Report, HPP-79-2.

[51] 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.

[52] Mitsuru Ohba and Xiao-Mei Chou. Does imperfect debugging affect software reliability growth? In *ICSE*, pages 237–244, May 1989.

[53] 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.

[54] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[55] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE*, pages 432–449, September 22–25, 1997.

[56] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, 24(6):401–419, June 1998.

[57] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *ASPLOS*, pages 35–45, October 1998.

[58] Jørgen F. Søgaard-Anderson, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computed-assisted simulation proofs. In Costas Courcoubetis, editor, *Fifth Conference on Computer-Aided Verification*, pages 305–319, Heraklion, Crete, June 1993. Springer-Verlag Lecture Notes in Computer Science 697.

[59] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *ASE '98*, pages 285–288, October 1998.

[60] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, February 1974.

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