Automatic Generation and Checking of Program Specifications

Jeremy W. Nimmer and Michael D. Ernst

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

Abstract

Producing specifications by dynamic (runtime) analysis of program executions is potentially unsound, because the analyzed executions may not fully characterize all possible executions of the program. In practice, how accurate are the results of a dynamic analysis? This paper describes the results of an investigation into this question, comparing specifications generalized from program runs with specifications verified by a static checker. The surprising result is that for a collection of modest programs, small test suites captured all or nearly all program behavior necessary for a specific type of static checking, permitting the inference and verification of useful specifications. For ten programs of 100–800 lines, the average precision, a measure of correctness, was .95 and the average recall, a measure of completeness, was .94.

This is a positive result for testing, because it suggests that dynamic analyses can capture all semantic information of interest for certain applications. The experimental results demonstrate that a specific technique, dynamic invariant detection, is effective at generating consistent, sufficient specifications. Finally, the research shows that combining static and dynamic analyses over program specifications has benefits for users of each technique.

1 Introduction

Dynamic (runtime) 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. This unsoundness makes dynamic analysis inappropriate for certain uses, and it may make users reluctant to depend on the results even in other contexts because of uncertainty as to their reliability.

By contrast, 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. Static analysis can be conservative and sound, and it is theoretically complete [CC77]. However, it can be inefficient, can produce weak results, and (as in the case of theorem-proving or program verification) can require explicit goals or annotations.

We have integrated and compared static and dynamic analyses over program specifications in order to understand the relationships between them. In particular, our investigation provides preliminary answers to the following questions.

How accurate is dynamic analysis? We do not have a theoretical answer to this question, nor can we predict how useful analysis results will be. (In any event, the answer depends on the particular use and user.) However, our experiments provide an interesting datapoint for the specific example of program specifications. Specifications form a particularly rich domain that captures a great deal of what is interesting about a program's semantics, and we show that a dynamic analysis can recover them accurately.

How can dynamic analysis be improved? The accuracy of a dynamic analysis can be improved in at least three ways. First, the dynamic analysis itself can be made more discriminating; we show that our specification inference analysis and its implementation are effective. Second, the dynamic analysis can be integrated with other analyses. For instance, passing potentially unsound output through a checker to remove unverifiable properties improves soundness while possibly reducing completeness. We have implemented and evaluated such a system. (The checker used by our implementation is unsound, but it nevertheless is of substantial benefit; its selection was an engineering tradeoff.) Third, feedback from the dynamic analysis can indicate how to improve test suites. Feedback about properties (not) satisfied may be at least as effective as code coverage feedback about lines (not) executed. This paper does not directly address such feedback, however.

How can dynamic analysis be used despite unsoundness? A dynamic analysis might produce results that are correct over all possible executions. If the results can be verified, then they can be used as if they resulted from a sound analysis. Our techniques produce fully verifiable results in many circumstances, but even less than perfect results can be of use. For instance, selecting and expressing goals for static verification can be difficult and tedious, and current systems have trouble postulating them. Starting from partial or nearly-true specifications could be easier for various tasks, including program verification, than starting from no specifications at all. Tool support for generating specifications has the potential to ease use of formal methods, enabling them to become more practical and more widely used. We provide preliminary evidence to support this claim.

Our results demonstrate that much of program semantics are present in test executions, as measured against verifiability of generated specifications. They also demonstrate that the technique of dynamic invariant detection is effective in capturing this information, and that the results are effective for the task of verifying absence of runtime errors. Finally, they show that static and dynamic analyses can be integrated to overcome the shortcomings of each: unsoundness for the dynamic analysis and lack of goals or tedious annotation for the static analysis.

1.1 Approach

We used program specifications to investigate the relationship between dynamically and statically available information about a program, and the accuracy of the former. Our approach is to extract specifications from program runs [Ern00, ECGN01] and determine whether they are correct and sufficient. For the purposes of this paper, our sufficiency measure is machine verifiability of the specifications. Correct specifications may be insufficient if limitations of the verifier prevent them from being proven.

The generated specifications are program invariants. These specifications are partial: they describe and constrain behavior but do not provide a full input–output mapping. The specifications are also unsound: as described later, the properties are likely, but not guaranteed, to hold.

A program invariant is a property that is true (or putatively 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; n = 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 [Gri81, LG86, HHJ+87b, HHJ+87a], document program operation [KL86, LCKS90], assist testing and enable correct modification [OC89, GKMS00], assist in test-case generation [TCMM98] and validation [CR99], form a program spectrum [AFMS96, RBDL97, HRWY98], and can enable optimizations [CFE99], among other uses. Despite their advantages, invariants are usually not stated explicitly in programs.

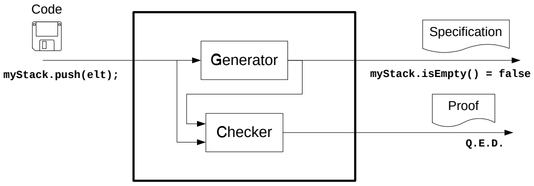

Figure 1: Generation and checking of program specifications results in a specification together with a proof of its correctness. Our generator is the Daikon invariant detector, and our checker is the ESC/Java static checker.

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.

To explore the issues listed above, we have integrated a dynamic invariant detector, Daikon [Ern00, ECGN01], with a static verifier, ESC/Java [DLNS98, LNS00]. Our system operates in three steps (see Figure 1) [NE01]. First, it runs Daikon, which outputs a list of likely invariants obtained from running the target program over a test suite. (We use the term "test suite" for any inputs over which executions are analyzed; those inputs need not satisfy any particular properties regarding code coverage or bug detection.) Second, it inserts those likely 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. 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, 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 notes challenges that arose while building and running our system. Section 5 discusses lessons learned from the experiments. Finally, Section 6 relates our results to other research, Section 7 proposes follow-on research, and Section 8 concludes.

2 Background

This section describes dynamic detection of program invariants, as performed by the Daikon tool, and static checking

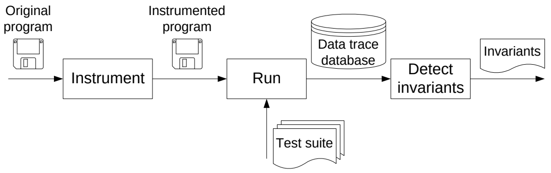

Figure 2: An overview of dynamic detection of invariants as implemented by Daikon.

of program annotations, as performed by the ESC/Java tool. Full details about the techniques and tools appear elsewhere.

2.1 Daikon: Invariant discovery

Dynamic invariant detection [Ern00, ECGN01] 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 2). 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 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 ($\mathsf{y} = \mathsf{fn}(\mathsf{x})$). Invariants involving a sequence variable (such as an array or linked list) 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 locally-checkable invariants such as node = node.child.parent (for all nodes), Daikon detects global invariants over pointer-directed data structures, such as mytree is sorted by $\leq$, by linearizing graph-like data structures. Finally, Daikon can detect conditional invariants such as "if $\mathsf{p} \neq \mathsf{null}$ then $\mathsf{p.value} > \mathsf{x}$" and "$\mathsf{p.value} > \mathsf{limit}$ or $\mathsf{p.left} \in \mathsf{mytree}$". 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 [EGKN99].

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 detecting 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.

An invariant is reported only if there is adequate statistical evidence for it. In particular, if there are an inadequate number of observations, observed patterns may be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random set of samples. The property is reported only if its probability is smaller than a user-defined confidence parameter [ECGN00].

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

2.2 ESC: Static checking

ESC [Det96, DLNS98, LN98] 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 verified 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 ESC/Modula-3. ESC/Java's annotation language (see Section 4.1) is simpler, because it is slightly weaker. This is in keeping with the philosophy of

StackAr	stack represented by an array
QueueAr	queue represented by an array
DisjSets	disjoint sets supporting union, find
Vector	`java.util.Vector` growable array
StreetNumberSet	collection of numeric ranges
GeoSegment	pair of points on the earth
Graph	generic graph data structure
RatNum	rational number
RatPoly	polynomial over rational numbers
FixedSizeSet	set represented by a bitvector

Figure 3: Description of the analyzed programs. These programs are available from the authors.

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.

ESC is not sound; for instance, it does not model arithmetic overflow, and permits the user to supply (unverified) assumptions. However, ESC provides a good approximation to soundness.

This paper uses ESC/Java not only as a lightweight technology for detecting a restricted class of runtime errors, but also as a tool for verifying representation invariants and method specifications. We chose to use ESC/Java 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 compare and combine dynamic and static techniques over the same code artifact.

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

3 Experiments

This section gives quantitative and qualitative results from a number of experiments. Results demonstrate that for certain programs, our system is able to infer specifications that are often precise and complete enough to be machine verifiable.

Section 3.1 presents our methodology. Sections 3.2 and 3.3 discuss two example programs in detail; these sections characterize the generated specifications and provide an intuition about the output of our system. Section 3.4 overviews other experiments and highlights the types of problems the system may encounter.

3.1 Methodology

We analyzed the programs listed in Figure 3. (Figure 4 summarizes the results.) The first three programs come from a data structures textbook [Wei99]; `Vector` is part of the Java standard library [Bla]; and the last six programs are staff solutions to assignments in a programming course at MIT [MIT01].

All of the programs except `Vector` came with test suites, either from the textbook or that were used for grading. Several of these test suites were small unit tests that contained just three or four calls per method and did not exercise the program's full functionality. We extended the deficient test suites, an easy task (see Section 4.4). We wrote our own test suite for `Vector`.

As described in Section 1.1, our system runs Daikon and inserts its output into the target program as ESC/Java annotations. Some of Daikon's invariants are inexpressible in ESC/Java's notation (the "Inexpr" column of Figure 4; also see Section 4.1). We did not study these further.

We determined by hand how many of Daikon's invariants were redundant because they were logically implied by other invariants (the "Redund" column of Figure 4). We ensured that redundant invariants verified exactly when their non-redundant counterparts did. We removed all of these invariants from further consideration, for two reasons. First, Daikon attempts to avoid reporting redundant invariants, but its tests are not perfect; these results indicate what an improved tool could achieve. More importantly, only one redundant invariant did not verify, so including redundant invariants would have inflated our results. Users would not need to remove the redundant invariants in order to use the tool.

We then measured how different the reported invariants are from a set of annotations that ESC/Java can verify (while also verifying that no run-time errors occur). There are potentially many such verifiable sets. For instance, one set of annotations might only ensure that no run-time errors occur, while another set might also ensure that a representation invariant is maintained. We selected as our goal set the one that required the smallest number of annotations to be added to or removed from the set that Daikon reported. This is a measure of how different the reported invariants are from a set that is both consistent and sufficient for ESC/Java's checking — an objective measure of how much of the semantics of the program was captured by Daikon from the program executions. It is also a measure of programmer effort to verify the program with ESC/Java, starting from a set of invariants detected by Daikon. One potential source of error is that we selected the goal set of annotations by hand; it is possible that we overlooked a closer goal. However, the numbers we present are a pessimistic bound, because any such error would degrade them.

Given the set of reported invariants and the goal set, we counted the number of invariants in both sets (the "Verif" column of Figure 4), the number only reported by Daikon (the "Unver" column), and the number only in the goal set (the "Miss" column). We computed precision and recall based on these three numbers.

	Program size			Number of invariants						Accuracy	
Program	LOC	NCNB	Meth.	Verif.	Unver.	Inexpr.	Redund.	Report.	Miss.	Prec.	Recall
FixedSizeSet	76	28	6	16	0	7	8	31	0	1.00	1.00
DisjSets	75	29	4	32	0	21	16	69	0	1.00	1.00
StackAr	114	50	8	25	0	6	1	32	0	1.00	1.00
QueueAr	116	56	7	42	0	11	5	58	13	1.00	0.76
Graph	180	99	17	15	0	0	1	16	4	1.00	0.79
GeoSegment	269	116	16	38	0	0	9	47	0	1.00	1.00
RatNum	276	139	19	25	2	0	9	36	1	0.93	0.96
StreetNumberSet	303	201	13	22	7	6	6	41	1	0.76	0.96
Vector	536	202	28	100	2	33	8	143	2	0.98	0.98
RatPoly	853	498	60	66	10	2	45	123	5	0.87	0.93
Average	280	142	18	38	2	9	11	60	3	0.95	0.94

Figure 4: Summary of invariants detected by Daikon and verified by ESC/Java. The programs are described in Figure 3. "LOC" is the total lines of code. "NCNB" is the non-comment, non-blank lines of code. "Meth" is the number of methods. "Verif" is the number of reported invariants that ESC/Java verified. "Unver" is the number of reported invariants that ESC/Java failed to verify. "Inexpr" is the number of reported invariants that were inexpressible in ESC/Java's annotation language. "Redund" is the number of reported invariants that were redundant; these are not counted in previous columns. "Report" is the total number of reported invariants, the sum of the previous four columns. "Miss" is the number of invariants not reported by Daikon but required by ESC/Java for verification. "Prec" is the precision of the reported invariants, the ratio of verifiable to verifiable plus unverifiable invariants. "Recall" is the recall of the reported invariants, the ratio of verifiable to verifiable plus missing.

3.2 StackAr: array-based stack

The `StackAr` example is an array-based stack implementation [Wei99]. The source contains 50 non-comment lines of code in 8 methods, along with comments that 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 verified that these properties hold. Daikon invariant detector finds 32 invariants, of which 25 are candidates for verification. In addition, our system heuristically added 2 annotations involving aliasing of the array.

Figure 5 shows part of the automatically-annotated source code for `StackAr`. The first six annotations describe the representation invariant, stating that the array index is legal and only unused storage is null. The next three annotations describe the specification for the constructor. Daikon also detects that after construction, all elements of the array are null, but this property is implied by the representation invariant, so Daikon does not report the property and it is not included in the results.

Our system generated specifications for all operations of the class, and verified that the implementation met the specification. For example, a postcondition for the pop method was the bi-implication:

```
(\old(topOfStack) == -1) == (\result == null)
```

This invariant states that the method returns `null` if and only if the stack is empty upon entry.

Without these annotations, ESC/Java issues warnings about many potential runtime errors, such as null dereferences and array bounds errors. With the addition of the detected invariants, ESC/Java issues no warnings, successfully checks that the `StackAr` class avoids runtime errors, and verifies that the implementation meets its generated specification.

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

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

  ...

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

  ...
}
```

Figure 5: The object invariants, first method, and field declarations of the annotated `StackAr.java` file [Wei99]. The ESC/Java 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.

	Verif.	Unver.	Inexpr.	Redund.	Report.	Miss.
Object	4	0	0	1	5	2
Requires	33	6	1	24	64	2
Ensures	29	4	1	20	54	1
Total	66	10	2	45	123	5

Figure 6: Breakdown of invariants detected by Daikon in the `RatPoly` program. The invariants are divided into object invariants, preconditions, and postconditions. The columns are the same as the "Number of invariants" columns of Figure 4.

3.3 RatPoly: polynomial over rational numbers

A second example further illustrates our results, and provides examples of verification problems.

The `RatPoly` program is an implementation of rational-coefficient polynomials that support basic algebraic operations [MIT01]. The source contains 498 non-comment lines of code, in 3 classes and 42 methods. Informal comments state the representation invariant and method specifications. Our system produced an annotation set that was close to a verifiable set. Additionally, the annotation set reflected some properties of the programmer's specification, which was given by informal comments.

Figure 6 shows that Daikon reported 123 invariants over the class; 10 of those did not verify, and 5 more had to be added.

The unverifiable invariants were all true, but other missing invariants prevented them from being verified. For instance, the `RatPoly` implementation maintains an object invariant that no zero-value coefficients are ever explicitly stored, so Daikon reported that a `get` method never returns zero. However, since elements of Java collection classes may not be accessed in ESC/Java annotations, the object invariant is not expressible and the `get` method failed to verify. Similarly, the `mul` operation exits immediately if one of the polynomials is undefined, but the determination of this condition also required annotations accessing Java collections. Thus, ESC/Java could not prove that helper methods used by `mul` never operated on undefined coefficients, as reported by Daikon.

The invariants that had to be added were of two categories. Some were due to a specification language mismatch between ESC/Java and Daikon. Daikon uses consistent notation to state the runtime type of elements in a sequence, whether it is an array or a Java collection class; ESC/Java expresses the two in unrelated notations. In our experiments, these properties had to be translated by hand, but automating this step is straightforward. The rest of the missing invariants were detected by Daikon, but suppressed for lack of statistical justification. Providing a more extensive test suite, or improving Daikon's statistical measures, would correct this problem.

3.4 Other experiments

We also performed eight other experiments, as shown in Figure 4. The results were positive and ranged from complete success as for `StackAr` to the occasional problems as outlined for `RatPoly`. The average precision (a measure of correctness) and recall (a measure of completeness) were 0.95 and 0.94, respectively.

Unverifiable invariants were either test suite artifacts or lacked supporting invariants. Test suite artifacts arise when the test suite maintains a property, even though that property is not generally true. These problems often indicate a deficiency in testing, but did not arise frequently in these experiments (see Section 4.4).

Unverifiable invariants more commonly occur when supporting invariants are outside the scope of the tools. For instance, in the `RatNum` class, Daikon found that the `negate` method preserves the denominator and negates the numerator. However, verifying that property would require detecting and verifying that the `gcd` operation called by the constructor has no effect because the numerator and denominator of the argument are relatively prime.

Missing invariants that could have reasonably been expected to be detected can also lead to failed verification. For example, the `QueueAr` class guarantees that unused storage is set to null. The representation invariants that maintain this property were missing from Daikon's output, because they were conditioned on a predicate more complicated than Daikon currently attempts. This omission prevented verification of many method postconditions.

Redundant invariants — those implied by other invariants — are often unhelpful to the user because they convey no new information. For instance, in the `DisjSets` class, Daikon reported that the `union` method ensured a certain property over all elements, but also reported the same property for various subsets of the elements. Redundant invariants may occasionally highlight important conclusions not obvious to the programmer, such as when a conclusion depends on invariants from several other objects in the system. However, in general redundant invariants are not useful, and we plan to improve Daikon's redundancy checks (see Section 7).

4 Limitations

This section discusses limitations of automatic generation and checking of program specifications. These limitations fall into three general categories: problems with the tools, problems with the target programs, and problems with the test suites for the target programs.

4.1 ESC/Java

ESC's input language is a variant of the Java Modeling Language JML [LBR99, LBR00], an interface specification language that specifies the behavior of Java modules. We use “ESCJML” for the JML variant accepted as input by ESC/Java.

Limitations of ESCJML prevent certain properties from being expressed. As a result, these properties must be omitted from the generated specifications, even though Daikon reports them as true over a program’s test suite. 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.

ESCJML requires that object invariants hold at entry to and exit from all methods, so it warned that the object invariants Daikon reported were violated by private helper methods. We worked around this problem by inlining one such method from the `QueueAr` program.

ESCJML cannot express invariants over strings, although Daikon reports few such invariants in any event. As a result, ESC/Java cannot verify 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.

The full JML language permits method calls in assertions, `\reach()` for expressing reachability via transitive closure, and specifies that object invariants hold only at entry to and exit from public methods.

Some of this functionality might be missing from ESC/Java because it is designed not for proving general program properties but as a lightweight method for verifying absence of runtime errors. However, our investigations revealed examples where such verification required each of these missing capabilities. In some cases, ESC/Java users may be able to restructure their code to work around these problems. In others, users can insert unsound pragmas that cause ESC/Java to assume particular properties without proof, permitting it to complete verification despite its limitations. We did not use any such pragmas in our experiments.

4.2 Daikon

A limitation of automatic generation of specifications involves invariants that Daikon does not detect — missing classes of invariants. Section 3.4 discussed problems with a `negate` method for rational numbers; a possible solution is to detect when numbers are relatively prime. We had previously rejected that invariant as of insufficiently general applicability.

Compared with previously published work, the version of Daikon used in this experiment incorporates several improvements essential to generating verifiable specifications. Of most interest, Daikon’s conditioning predicates were enhanced to include boolean procedure return values and procedure exit points. Daikon uses these predicates to produce implications and disjunctions, which are critical to specifying methods that take different actions depending on internal state.

Program	NCNB	Original	Added
FixedSizeSet	28	0	39
DisjSets	29	27	15
StackAr	50	11	39
QueueAr	56	11	54
Graph	99	Sys	1
GeoSegment	116	Sys	0
RatNum	139	Sys	39
StreetNumberSet	201	165	151
Vector	202	0	190
RatPoly	498	382	15

Figure 7: Comparison of program size to test suite size, given in non-comment, non-blank lines of code. “NCNB” is size of the program; “Original” is the size of its original, accompanying test suite; “Added” is the number of lines added to yield the results described in Section 3. “Sys” indicates a system test not specifically focused on the program (see Section 4.4).

4.3 Target programs

Another challenge to verification of invariants is the likelihood that programs contain errors that falsify the desired invariant. (Although it was never our goal, we have previously identified such errors in textbooks [Gri81, Wei99], in programs used in testing research [HFGO94, RH98], and elsewhere.) 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. The `pop` operations maintain this invariant (which approximately doubles the size of their code), but the `makeEmpty` operation does not. We noticed this when the expected object invariant was not inferred, and we corrected the error in our version of `StackAr`.

4.4 Test suites

A final challenge to generation is deficient or missing test suites. If the executions provided by a test suite are not characteristic of a program’s general behavior, properties observed during testing may not generalize. However, one of the key results of this research is that even limited test suites can capture certain semantics of a program.

Figure 7 shows relative sizes of test suites and programs used in this experiment. Test suites for the smaller programs were larger in comparison, but no test suite was unreasonably sized.

System tests — tests that check end-to-end behavior of a system — tended to produce good invariants immediately, confirming earlier experiences [ECGN01]. These system tests were for a system containing the module we examined, rather than being just for the module itself.

Unit tests — tests that check specific boundary values of procedures in a single module in isolation — were not immediately successful. When the initial test suites were unit tests that came from the textbooks or were used for grading, they often contained just three or four calls per method. Some methods on `StreetNumberSet` were not tested at all.

We corrected these test suites, but did not attempt to make them minimal. The corrections were not difficult. When failed ESC/Java verification attempts indicate a test suite is deficient, the unverifiable invariants specify the unintended property, so a programmer knows exactly how to improve the tests. For example, the original tests for the `div` operation on `RatPoly` exercised a wide range of positive coefficients, but all tests with negative coefficients used a numerator of -1. Other examples included certain stack operations not being performed on a full stack, calls to a safe stack pop operation always being protected by a check whether the array was empty, and a queue implemented via an array not being forced to wrap around. These properties were detected and reported as unverifiable by our system, and extending the tests to cover additional values was effortless.

Test suites are an important part of any programming effort, so time invested in their improvement is not wasted. In our experience, the additional effort (if any) required to obtain accurate invariants is indistinguishable from that required to create a general test suite. In short, poor verification results indicate specific failures in testing, and reasonably-sized test suites are able to accurately capture semantics of a program.

5 Discussion

The most surprising result of our research is that specifications generated from program executions are reasonably accurate: they form a set that is (nearly) self-consistent and self-sufficient, as measured by verifiability by an automatic specification checking tool. This result was not at all obvious *a priori*. One might expect that dynamically detected invariants would suffer from serious unsoundness by expressing artifacts of the test suite and would fail to capture enough of the (formal) semantics of the program.

This positive result implies that dynamic invariant detection is effective, at least in our domain of investigation. A second, broader conclusion is that executions over relatively small test suites capture a significant amount of information about program semantics. This detected information is equivalent to that resulting from, and verifiable by, a static analysis. Although we do not yet have a theoretical model to explain this, nor can we predict for a given test suite how much of a program's semantic space it will explore, we have presented a datapoint from a set of experiments to explicate the phenomenon and suggest that it may generalize.

We speculate that three factors may contribute to our success. First, our specification generation technique does not attempt to report all properties that happen to be true during a test run. Rather, it produces partial specifications that intentionally omit properties that are unlikely to be of use or that are unlikely to be universally true. It uses statistical, algorithmic, and heuristic tests to make this judgment. Second, the information that ESC/Java needs for verification may be particularly easy to obtain via a dynamic analysis. ESC/Java's requirements are modest: it does not need full formal specifications of all aspects of program behavior. However, it does require some specifications and input–output relations, and we were able to verify detected properties that were not strictly necessary for ESC's checking, but provided additional information about program behavior. Third, our test suites were of acceptable quality. Unit tests are inappropriate, for they produce very poor invariants. However, Daikon's output makes it extremely easy to improve the test suites by indicating exactly what is wrong with them. Furthermore, existing system tests were adequate, and these are more likely to exist and often easier to produce.

Our results suggest a new metric for test suite quality, which we call "value coverage" [Ham87, CR99] or "specification coverage." Specifications are closer than code coverage is to the abstract, semantic level at which programs are often understood. Software engineers may more readily interpret program properties than specific paths through the program, even if they would eventually equate the two. We are unsure whether specification-complete (or specification-verifiable) test suites — that is, test suites from whose executions complete or verifiable specifications can be dynamically extracted — are good for catching bugs, and whether they tend to be coverage-complete. We would like to further investigate these topics.

We do know that dynamically detected program invariants make it easy to construct and extend test suites to achieve specification completeness. There is substantial anecdotal evidence that they also assist in detection of bugs. For example, in addition to the `StackAr` problem noted in Section 4.3, our experiments also revealed a bug in the `Vector` class from JDK 1.1.8. The `toString` method throws an exception for vectors with null elements. Our original (code coverage complete) test suite did not reveal this bug, but Daikon reported that the vector elements were always non-null on entry to `toString`, leading to discovery of the bug. The bug is corrected in JDK 1.3.

The goal of producing program specifications is so important that it is worthwhile to consider many approaches. Our research suggests that a novel approach can complement existing ones: generate the specification unsoundly, then check it, resulting in a specification and a verification of its correctness. We believe that unsound specifications can also be used to advantage in other situations: this can expand the applicability and utility of specifications and provide many of the benefits of sound specifications, in more situations. Even if full input–output relations are hard to generate automatically, universally true properties (especially conditional invariants) that characterize the relation are a step in the right direction.

5.1 Benefits of integration

Static and dynamic analyses have complementary strengths and weaknesses, so combining them has great promise: dynamic analysis can propose program properties to be verified by static analysis. 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 (as proven true or potentially incorrect) to permit programmers to use it more effectively, permits verified 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. This may eventually lead 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 can bootstrap static verification by providing initial program annotations, goals, and intermediate assertions. Few programmers enjoy or are good at annotating programs, a time-consuming, tedious, and error-prone task. This automation may speed the adoption of static analysis tools by lessening the user burden, even if some work still remains for the user. Dynamically detected invariants can also check and refine existing specifications and indicate properties programmers might otherwise have overlooked. These improvements could lead to prevention and to earlier detection of errors, aiding in the production of more robust, reliable, and correct computer systems.

6 Related work

This is the first research we are aware of that has dynamically generated, then statically verified, program specifications, or has used such information to investigate the amount of information about program semantics available in test runs. The two component techniques are well-known, however.

Dynamic analysis has been used for a variety of tasks; for instance, inductive logic programming (ILP) [Qui90, Coh94] produces a set of Horn clauses (first-order if-then rules) and can be run over program traces [BG93], though with limited success. Programming by example [CHK$^+$93] is similar but requires close human guidance, and version spaces can compactly represent sets of hypotheses [Mit78, Hir91, LDW00]. Value profiling [CFE97, SS98, CFE99] can efficiently detect certain simple properties at runtime. Event traces can generate finite state machines that explicate potential system organization or behavior [BG97, CW98a, CW98b]. Program spectra [AFMS96, RBDL97, HRWY98, Bal99] also capture aspects of system runtime behavior. None of these other techniques has been as successful as Daikon for detecting invariants in programs, though many have been valuable in other domains.

Many static inference techniques also exist, including abstract interpretation (often implemented by symbolic execution or dataflow analysis), model checking, and theorem proving. (Space prohibits a complete review here.) A sound, conservative static analysis reports properties that are true for any program run, and theoretically can detect all sound invariants if run to convergence [CC77]. Static analyses omit properties that are true but uncomputable and properties of the program context. To control time and space complexity (especially the cost of modeling program states) and ensure termination, they make approximations that introduce inaccuracies, weakening their results. For instance, accurate and efficient alias analysis is still beyond the state of the art [CWZ90, LR92, WL95], though for specific applications, contexts, or assumptions, efficient pointer analyses can be sufficiently accurate [Das00].

There are many other tools besides ESC/Java for statically checking specifications [Pfe92, DC94, EGHT94, Det96, Eva96, NCOD97, LN98]. These other systems have different strengths and weaknesses than ESC/Java, but few have the polish of its integration with a real programming language.

An independent project [JvH$^+$98, HJv01] verified an object invariant in Java's `Vector` class, using automatic translation to PVS [ORS92, ORSvH95], user-specified goals, and some user interaction with PVS.

6.1 Houdini

The research most closely related to our integrated system is Houdini, an annotation assistant for ESC/Java [FL01, FJL01]. (A similar system was proposed by Rintanen [Rin00].) 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.

Houdini has been used to find bugs in several programs. Over 30% of its guessed annotations are verified, and it tends to reduce the number of ESC/Java warnings by a factor of 2–5. At present, Houdini may be more scalable than our system. Houdini took 62 hours to run on a 36,000-line program.

Daikon has run in under an hour on several 10,000-line programs. Because it currently operates offline in batch mode, its memory requirements make Daikon unlikely to scale to significantly larger systems without re-engineering; such an effort is now underway. This is a limitation of the Daikon prototype, not of the technique of dynamic invariant detection.

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 eliminates all other 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 [ECGN00]), 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 dynamic invariant detector — has been built [FL01], 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 verifiable by ESC/Java 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 4 listed a number of limitations of our system (and its components Daikon and ESC/Java) that should be corrected. We would also like to investigate what test suites lead to good specifications, as noted in Section 5.

Another obvious way to extend this work is to use different invariant detectors than Daikon or different verifiers than ESC/Java. Section 6 lists some other invariant detectors. Examples of static verifiers that are connected with real programming languages include LCLint [EGHT94, Eva96, Eva00], ACL2 [KM97], LOOP [JvH+98], Java PathFinder [HP00], and Bandera [CDH+00].

We are currently integrating Daikon with IOA [GLV97, GL00], a formal language for describing computational processes that are modeled using I/O automata [Lyn96, LT87, LT89]. 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 [GG90, GG91, SAGG+93], 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 or representations (but not in all possible states or 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 [Weg74, BLS96, BBM97]. Our preliminary experiments have resulted in the automatic detection of invariants used in a published proof [GL00].

We are also interested in recovering from failed attempts at static verification. Broadly speaking, verification fails because the goal properties are too strong or 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 successful in several application domains, we believe that truly successful program analysis requires both static and dynamic components. Some of the properties that are difficult to obtain from a dynamic analysis 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). For example, 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 proposed, implemented, and experimentally assessed a novel approach to producing correct specifications: generate them unsoundly from program executions, then verify them. To our knowledge, ours is the first system to dynamically detect and then statically verify program specifications.

Our experiments indicate that even limited test suites accurately characterize general execution properties: they can generate a consistent and sufficient set of specifications that can be automatically verified with little or no change. This surprising result suggests that runtime properties may not be as unreliable as general opinion holds, given an effective method for extracting them. We do not yet have a principled description of the static characteristics of a test suite that result in a high-quality generated specification, but even simple system tests seem to be sufficient.

Our experiments also demonstrate the effectiveness of dynamic invariant detection, and of the Daikon implementation. More specifically, in our tests, it generated specifications with high (about 95%) precision and recall, when measured against the task of static verification by ESC/Java. This validates the approach of producing invariants from program executions.

The results generally justify the use of unsound techniques in appropriate ways in program development and suggest that these may be extended to program specifications, which have traditionally required complete correctness. We also found that integrating static and dynamic techniques in our system produces benefits in each direction, because of their complementary strengths and weaknesses. Finally, dynamically generated specifications may assist in bug detection and prove to be a valuable measure of test suite quality.

Acknowledgments

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

References

[AFMS96] David Abramson, Ian Foster, John Michalakes, and Rok Socič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

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

[BBM97] Nicolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[BG93] 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.

[BG97] Bernard Boigelot and Patrice Godefroid. Automatic synthesis of specifications from the dynamic observation of reactive programs. In *TACAS ’97*, pages 321–333, Twente, April 1997.

[Bla] Blackdown project. Java Development Kit (JDK) version 1.1.8 for Linux. `http://www.blackdown.org/`.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *CAV*, pages 323–335, July 31–August 3, 1996.

[CC77] Patrick M. Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Rochester, NY, August 1977.

[CDH+00] 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.

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

[CFE99] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999. `http://www.jilp.org/vol1/`.

[CHK+93] 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.

[Coh94] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, August 1994.

[CR99] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *ESEC/FSE*, pages 285–302, September 6–10, 1999.

[CW98a] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, July 1998.

[CW98b] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, November 1998.

[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.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, June 18–23, 2000.

[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.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[ECGN01] 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.

[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.

[EGKN99] 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.

[Ern00] Michael D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

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

[Eva00] David Evans. *LCLint User's Guide, Version 2.5*, May 2000. `http://lclint.cs.virginia.edu/guide/`.

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, February 2001.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, volume 2021 of *LNCS*, pages 500–517, Berlin, Germany, March 2001.

[GG90] Stephen Garland and John Guttag. LP, the Larch Prover. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction*, volume 449 of *LNCS*, Kaiserslautern, West Germany, 1990. Springer-Verlag.

[GG91] 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.

[GKMS00] 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.

[GL00] 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.

[GLV97] 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.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Ham87] Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25(1):17–25, April 20, 1987.

[HFGO94] 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.

[HHJ$^+$87a] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Corrigenda: "Laws of programming". *Communications of the ACM*, 30(9):771, September 1987. See [HHJ$^+$87b].

[HHJ$^+$87b] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987. See corrigendum [HHJ$^+$87a].

[Hir91] Haym Hirsh. Theoretical underpinnings of version spaces. In *IJCAI*, pages 665–670, August 1991.

[HJv01] Marieke Huisman, Bart P.F. Jacobs, and Joachim A.G.M. van den Berg. A case study in class library verification: Java's Vector class. *International Journal on Software Tools for Technlogy Transfer*, 2001.

[HP00] 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.

[HRWY98] 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.

[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.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE TSE*, 12(1):96–109, January 1986.

[KM97] 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.

[LBR99] 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.

[LBR00] 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`.

[LCKS90] 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.

[LDW00] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, Stanford, CA, June 2000.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.

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

[LNS00] 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.

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

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, Vancouver, BC, Canada, August 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[Mit78] 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.

[MIT01] MIT Dept. of EECS. 6.170: Laboratory in software engineering. `http://www.mit.edu/~6.170/`, Spring 2001.

[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.

[NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[OC89] Mitsuru Ohba and Xiao-Mei Chou. Does imperfect debugging affect software reliability growth? In *ICSE*, pages 237–244, May 1989.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607, pages 748–752, Saratoga Springs, NY, June 1992.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE TSE*, 21(2):107–125, February 1995. Special Section—Best Papers of FME (Formal Methods Europe) '93.

[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.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RBDL97] 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.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, 24(6):401–419, June 1998.

[Rin00] Jussi Rintanen. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, pages 806–811, Austin, TX, July 30–August 3, 2000.

[SAGG$^+$93] Jørgen F. Søgaard-Anderson, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-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.

[SS98] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *ASPLOS*, pages 35–45, October 1998.

[TCMM98] 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.

[Weg74] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, February 1974.

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

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, pages 1–12, June 1995.