Automatic Generation 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, determining how much specifications generalized from program runs must be changed in order to be verified by a static checker. Surprisingly, small test suites captured nearly all program behavior required by a specific type of static checking; the static checker guaranteed that the implementations satisfy the generated specifications, and ensured the absence of runtime exceptions. Measured against this verification task, the generated specifications scored over 90% on precision, a measure of soundness, and on recall, a measure of completeness.

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 for use by a static checker. Finally, the research shows that combining static and dynamic analyses over program specifications has benefits for users of each technique, guaranteeing soundness of the dynamic analysis and lessening the annotation burden for users of the static analysis.

1. Introduction

This paper investigates combining dynamic and static analyses for the task of recovering formal program specifications. The paper evaluates the accuracy of a dynamic analysis by measuring the static verifiability of its result. The accuracy of a dynamic analysis is of interest because its accuracy affects its utility. Recovering specifications is a valuable goal because specifications are useful in testing, debugging, verification, maintenance, and optimization, among other tasks, but are frequently absent from programs, depriving software engineers of their benefits.

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 potential 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 possible 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. Selecting a goal and annotating programs for input to a static checker can be difficult and tedious.

Combining these techniques overcomes the weaknesses of each: dynamically detected invariants can annotate a program or provide goals for static verification (easing tedious annotation), and static verification can confirm properties proposed by a dynamic tool (mitigating its unsoundness). Using the combined system is much better than relying on only one of the tools, or performing error-prone hand analysis.

We evaluate the effectiveness of the combined analysis by measuring how much dynamically generated specifications must be changed in order to be verified by a static checker. The static checker both guarantees that the implementation satisfies the generated specification and ensures the absence of runtime exceptions. (No checker can assess how well the specifications reflect programmer intent.) Measured against this verification requirement, the generated specifications scored over 90% on precision, a measure of soundness, and on recall, a measure of completeness.

Our results demonstrate that non-trivial and useful aspects of program semantics are present in test executions, as measured by verifiability of generated specifications. Our results 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. Furthermore, even imperfect specifications can be of use. For instance, current systems have trouble postulating verification goals. Users may find starting from partial or nearly-true specifications easier for various tasks, including program verification, than starting from no specifications at all.

1.1 Approach

We used formal 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 sufficient for machine verifiability of

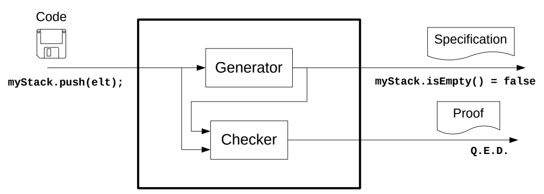

Figure 1: Generation and checking of program specifications results in a specification together with a proof of its consistency with the code. Our generator is the Daikon invariant detector, and our checker is the ESC/Java static checker.

the absence of runtime errors.

A (formal) specification is a precise description of a program's behavior. (Appendix A discusses alternate definitions.) Specifications often state properties of data structures, such as object invariants, or relate variable values in the pre-state (before a procedure call) to their post-state values (after a procedure call).

A specification for a procedure that records its maximum argument in variable *max* might include

$$\text{if } arg > max \text{ then } max' = arg \text{ else } max' = max$$

where *max* represents the value at the time the procedure is invoked and max' represents the value of the variable when the procedure returns. A typical specification contains many clauses, some of them simple mathematical statements and others involving post-state values or implications. The clauses are conjoined to produce the full specification. These specification clauses are often called *invariants*. There is no single best specification for a program; different specifications include more or fewer clauses and assist in different tasks. Likewise, there is no single correct specification for a program; correctness must be measured relative to some standard, such as the designer's intent, or task, such as program verification.

Our generated specifications consist of 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 in Section 2.1, the properties are likely, but not guaranteed, to hold. Finally, the specifications describe the program's actual behavior, which may vary from the programmer's intended behavior.

These aspects of the generated specification suggest certain uses while limiting others. Our research shows that the specification is useful in verifying the lack of runtime exceptions. In contrast, using it as a template for automatic test case generation might add little value, since the specification already reflects the program's behavior over a test suite. If the program is correct or nearly so, the generated specification is near to the intended behavior, and can be corrected to reflect the programmer's intent. Likewise, the generated specification can be corrected to be verifiable by a static checker, guaranteeing the absence of certain errors and adding confidence to future maintenance tasks.

Users need not mimic our evaluation strategy by statically verifying generated specifications: even uncorrected specifications can be useful. Generated specifications are useful for program refactoring [KEGN01], theorem proving [NWEL02], test suite generation [Har02], and anomaly and bug detection [ECGN01, RKS02, HL02, Dod02]. In many of these tasks, the accuracy of the generated specification (the degree to which it matches the code) affects the effort involved in performing the task. This paper evaluates the accuracy of the specifications, with respect to verification by ESC/Java.

To evaluate the accuracy of the generated specifications, we have integrated a dynamic invariant detector, Daikon [Ern00, ECGN01], with a static verifier, ESC/Java [DLNS98, LNS00], resulting in a system that produces machine verifiable specifications (see Figure 1). Our system operates in three steps [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 fault detection.) Second, it inserts the 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, but users may improve results by editing and re-running test suites, or by adding or removing specific program annotations by hand.

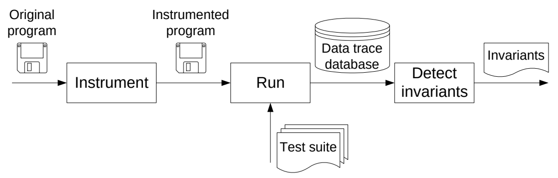

Figure 2: An overview of dynamic detection of invariants as implemented by the Daikon invariant detector.

The remainder of this paper is organized as follows. Section 2 provides background on the dynamic specification generator and the static verifier used by our system. Section 3 presents methodology for several experiments, and Section 4 presents their results. Section 5 notes challenges that arose while building and running our system. Section 6 discusses lessons learned from the experiments. Finally, Section 7 relates our research to previous work, and Section 8 concludes.

2. Background

This section briefly describes dynamic detection of program invariants, as performed by the Daikon tool, and static checking of program annotations, as performed by the ESC/Java tool. Full details about the techniques and tools appear elsewhere.

2.1 Daikon: Specification generation

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, Java, and IOA [GLV97].

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 ($\mathsf{x} \in \{\mathsf{a,b,c}\}$), lying in

	Program size			Number of invariants			Accuracy		
Program	LOC	NCNB	Meth.	Verif.	Unver.	Miss.	Prec.	Recall	Description
FixedSizeSet	76	28	6	16	0	0	1.00	1.00	set represented by a bitvector
DisjSets	75	29	4	32	0	0	1.00	1.00	disjoint sets supporting union, find
StackAr	114	50	8	25	0	0	1.00	1.00	stack represented by an array
QueueAr	116	56	7	42	0	13	1.00	0.76	queue represented by an array
Graph	180	99	17	15	0	2	1.00	0.88	generic graph data structure
GeoSegment	269	116	16	38	0	0	1.00	1.00	pair of points on the earth
RatNum	276	139	19	25	2	1	0.93	0.96	rational number
StreetNumberSet	303	201	13	22	7	1	0.76	0.96	collection of numeric ranges
Vector	536	202	28	100	2	2	0.98	0.98	`java.util.Vector` growable array
RatPoly	853	498	42	70	10	1	0.88	0.99	polynomial over rational numbers
MapQuick	2088	1031	113	145	3	35	0.98	0.81	driving directions query processor
Total	4886	2449	273	530	24	55	0.96	0.91	

Figure 3: Summary of invariants detected by Daikon and verified by ESC/Java. "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. "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.

a range ($a \leq x \leq b$), non-zero, modulus ($x \equiv a \pmod{b}$), linear relationships ($z = ax + by + c$), ordering ($x \leq y$), and functions ($y = fn(x)$). 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 ($x \in y$). Given two sequences, some example checked invariants are elementwise linear relationship, lexicographic comparison, and subsequence relationship. Finally, Daikon can detect implications such as "$\text{if } p \neq \text{null then } p.\text{value} > x$" and disjunctions such as "$p.\text{value} > \text{limit or } p.\text{left} \in \text{mytree}$". In this paper, we ignore those invariants that are inexpressible in ESC/Java's input language; for example, many of the sequence invariants are ignored.

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. Because false invariants tend to be falsified quickly, 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://pag.lcs.mit.edu/daikon/`.

2.2 ESC: Static checking

ESC [Det96, DLNS98, LN98], the Extended Static Checker, 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 type-checkers 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 supplied 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 5.5) 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.

ESC/Java is not sound; for instance, it does not model arithmetic overflow or track aliasing, it assumes loops are executed 0 or 1 times, and it permits the user to supply (unverified) assumptions. However, ESC/Java provides a good approximation to soundness: it issues false warnings relatively infrequently, and successful verification increases confidence in a piece of code. (Essentially every verification process over programs contains an unsound step, but it is sometimes hidden in a step performed by a human being, such as model creation.)

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. Methodology

We analyzed the programs listed in Figure 3. `DisjSets`, `StackAr`, and `QueueAr` come from a data structures text-

	Size	Original Test Suite						Augmented Test Suite					
		Size		Coverage		Time		Size		Coverage		Time	
Program	NCNB	NCNB	Calls	Stmt	Branch	Instr	Daikon	+NCNB	Calls	Stmt	Branch	Instr	Daikon
FixedSizeSet	28	0	0	0.00	0.00	0	0	39	12k	1.00	1.00	2	10
DisjSets	29	27	745	0.67	0.70	1	6	15	12k	1.00	0.90	3	18
StackAr	50	11	72	0.60	0.56	0	3	39	1k	0.64	0.63	0	4
QueueAr	56	11	52	0.68	0.65	0	12	54	8k	0.71	0.71	1	11
Graph	99	Sys	3k	0.76	0.54	1	3	1	3k	0.76	0.54	1	3
GeoSegment	116	Sys	695k	0.89	0.75	138	455	0	695k	0.89	0.75	138	455
RatNum	139	Sys	58k	0.96	0.94	7	28	39	114k	0.96	0.94	14	56
StreetNumberSet	201	165	50k	0.95	0.93	7	29	151	197k	0.95	0.95	12	44
Vector	202	0	0	0.00	0.00	0	0	190	22k	0.90	0.90	7	37
RatPoly	498	382	88k	0.94	0.89	27	98	51	102k	0.96	0.92	38	139
MapQuick	1031	445	3.31M	0.66	0.61	660	1759	49	3.37M	0.67	0.71	673	1704

Figure 4: Characterization of test suites. "NCNB" is non-comment, non-blank lines of code in the program or its original, accompanying test suite; in this column "Sys" indicates a system test: one that is not focused on the specified program, but tests a higher-level system that contains the program (see Section 5.2). "+NCNB" is the number of lines added to yield the results described in Section 4. "Calls" is the dynamic number of method calls received by the program under test (from the test suite or internally). "Stmt" and "Branch" indicate the statement and branch coverage of the test suite. "Instr" is the runtime of the instrumented program. "Daikon" is the runtime of the Daikon invariant detector. Times are wall-clock measurements, in seconds.

book [Wei99]; `Vector` is part of the Java standard library; and the remaining seven programs are solutions to assignments in a programming course at MIT.

As described in Section 1.1, our system runs Daikon and inserts its output into the target program as ESC/Java annotations.

We measured how different the reported invariants are from a set of annotations that enables ESC/Java to verify that no run-time errors occur (while ESC/Java also verifies the annotations themselves). There are potentially many sets of ESC/Java-verifiable annotations for a given program. In order to perform an evaluation, we must choose one of them as a goal.

There is no one "correct" or "best" specification for a program: different specifications support different tasks. For instance, one set of ESC/Java annotations might ensure that no run-time errors occur, while another set might ensure that a representation invariant is maintained, and yet another set might guarantee correctness with respect to externally imposed requirements.

We chose as our goal task verifying the absence of run-time errors. Among the sets of invariants that enable ESC/Java to prove that condition, we selected as our goal set the one that required the smallest number of changes to the Daikon output. The distance to this goal set is a measure of the minimal (and the expected) effort needed to verify the program with ESC/Java, starting from a set of invariants detected by Daikon. Our choice 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 the program semantics captured by Daikon from the executions.

Given the set of invariants reported by Daikon and the changes necessary for verification, we counted the number of reported and verified invariants (the "Verif" column of Figure 3), reported but unverifiable invariants (the "Unver" column), and unreported, but necessary, invariants (the "Miss" column). We computed precision and recall, standard measures from information retrieval [Sal68, vR79], based on these three numbers. Precision, a measure of soundness, is defined as $\frac{\text{Verif}}{\text{Verif}+\text{Unver}}$. Recall, a measure of completeness, is defined as $\frac{\text{Verif}}{\text{Verif}+\text{Miss}}$. For example, if Daikon reported 6 invariants (4 verifiable and 2 other unverifiable), while the verified set contained 5 invariants (the 4 reported by Daikon plus 1 added by hand), the precision would be 0.67 and the recall would be 0.80.

We determined by hand how many of Daikon's invariants were redundant because they were logically implied by other invariants. Users would not need to remove the redundant invariants in order to use the tool, but we removed all of these invariants from consideration (and they appear in none of our measurements), 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, almost all redundant invariants were verifiable, so including redundant invariants would have inflated our results.

3.1 Test suites

Figure 4 shows relative sizes of the test suites and programs used in this experiment. Test suites for the smaller programs were larger in comparison to the code size, but no test suite was unreasonably sized.

All of the programs except `Vector` and `FixedSizeSet` came with test suites, from the textbook or that were used for grading. We wrote our own test suites for `Vector` and `FixedSizeSet`. The textbook test suites are more properly characterized as examples of calling code; they contained just a few calls per method and did not exercise the program's full functionality. We extended the deficient test suites, an easy task (see Section 5.2) and one that would be less necessary for programs with realistic test suites.

We generated all but one test suite or augmentation in less than 30 minutes. `MapQuick`'s augmentation took 3 hours due to a 1 hour round-trip time to evaluate changes. We found that examining Daikon's output greatly eased this task.

4. Experiments

This section gives quantitative and qualitative experimental results. The results demonstrate that the dynamically inferred specifications are often precise and complete enough to be machine verifiable.

Section 4.1 summarizes our experiments, while Sections 4.2 through 4.4 discuss three example programs in detail to characterize the generated specifications and provide an intuition about the output of our system. Section 5 summarizes the problems the system may encounter.

4.1 Summary

We performed eleven experiments, as shown in Figure 3. As described in Section 3, Daikon's output is inserted into the target program as annotations, which are edited (if necessary) until the result verifies. When the program verifies, the implementation meets the generated and edited specification, and runtime errors are guaranteed to be absent.

In programs of up to 1031 non-comment non-blank lines of code, the overall precision (a measure of soundness) and recall (a measure of completeness) were 0.96 and 0.91, respectively. Later sections describe specific problems that lead to unverifiable or missing invariants, but we summarize the imperfections here.

Most unverifiable invariants correctly described the program, but could not be proved due to limitations of ESC/Java. Some limitations were by design, while others appeared to be bugs in ESC/Java.

Most missing invariants were beyond the scope of Daikon. Verification required certain complicated predicates or element type annotations for non-`List` collections, which Daikon does not currently provide.

4.2 StackAr: array-based stack

`StackAr` 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.

When run on an unannotated version of `StackAr`, ESC/Java issues warnings about many potential runtime errors, such as null dereferences and array bounds errors. Our system generated specifications for all operations of the class, and 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 the generated specification.

The Daikon invariant detector reported 25 invariants, including the representation invariant, method preconditions, modification targets, and postconditions. (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: the array is non-null and contains elements of arbitrary run-time type, the array index is legal, and only unused array elements are null. The next three annotations specify the constructor. Daikon also detects that after construction, all elements of the array are null, but this property is implied by the representation invariant and the fact that `topOfStack` is -1, so Daikon does not report the property. The last five invariants specify the `topAndPop` method. Only elements above `topOfStack` are modified, and `topOfStack` never increases. If `topOfStack` was originally non-negative, it is decremented and a non-null result is returned.

```
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)) */

 private Object [ ] theArray;
 private int        topOfStack;

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

 //@ modifies topOfStack, theArray[*]
 /*@ ensures (\forall int i; (0 <= i && i <= topOfStack)
   ==> (theArray[i] == \old(theArray[i]))) */
 //@ ensures topOfStack <= \old(topOfStack)
 /*@ ensures (\old(topOfStack) >= 0) ==>
   (topOfStack == \old(topOfStack) - 1) */
 /*@ ensures (\old(topOfStack) >= 0) ==>
   (\result == \old(theArray[topOfStack])) */
 //@ ensures (\old(topOfStack) >= 0) == (\result != null)
 public Object topAndPop( ) {
   if( isEmpty( ) )
     return null;
   Object topItem = top( );
   theArray[ topOfStack-- ] = null;
   return topItem;
 }

 ...
}
```

Figure 5: The object invariants and two method specifications 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. For clarity, the figure wraps some lines; it also omits four auxiliary annotations that are irrelevant to the specification but are required by ESC/Java and are inserted by our system.

4.3 RatPoly: polynomial over rational numbers

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

`RatPoly` is an implementation of rational-coefficient polynomials that support basic algebraic operations. 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 a nearly-verifiable annotation set. Additionally, the annotation set reflected some properties of the programmer's specification, which was given by informal comments. Figure 3 shows that Daikon reported 80 invariants over the program; 10 of those did not verify, and 1 more had to be added.

The 10 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, ESC/Java annotations may not reference elements of Java collection classes; thus, 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.

When using the provided test suite, three invariants were detected by Daikon, but suppressed for lack of statistical justification. Small test suite augmentations (Figure 4) more extensively exercised the code and caused those invariants to be printed. (Alternately, a command-line switch to Daikon sets its justification threshold.) With the test suite augmentations, only one invariant had to be edited by hand (thus counting as both unverified and missing): an integer lower bound had to be weakened from 1 to 0 because ESC/Java's incompleteness prevented proof of the (true, but subtle) stricter bound.

4.4 MapQuick: driving directions

A final example further illustrates our results.

The `MapQuick` application computes driving directions between two addresses provided by the user, using real-world geographic data. The source contains 1835 non-comment lines of code in 25 classes, but we did not compute specifications for 7 of the classes. Of the omitted classes, three classes were used so frequently (while loading databases) that recording traces for offline processing was infeasible due to space limitations. One class (the entry point) was only called a few times, so not enough data was available for inference. Two classes had too little variance of data for inference (only a tiny database was loaded). Finally, one class had a complex inheritance hierarchy which prevented local reasoning (and thus hindered modular static analysis). All problems but the last could have been overcome by an invariant detector that runs online, allowing larger data sets to be processed and a more varied database to be loaded.

We verified the other 18 classes (113 methods, 1031 lines). The verified classes include data types (such as a priority queue), algorithms (such as Dijkstra's shortest path), a user interface, and various file utilities. Figure 3 shows that Daikon reported 148 invariants; 3 of those did not verify, and 35 had to be added.

The 3 unverifiable invariants were beyond the capabilities of ESC/Java, or exposed bugs in the tools.

The largest cause of missing invariants was ESC/Java's incompleteness. Its modular analysis or incomplete knowledge of Java semantics forced 9 annotations to be added, while 3 more were required because other object invariants were inexpressible.

Invariants were also missing because they were outside the scope of Daikon. Daikon does not currently report invariants of non-`List` Java collections, but 4 invariants about type and nullness information of these collections were required for ESC/Java verification. Daikon also missed 5 invariants because it does not instrument interfaces, and 3 invariants over local variables, which are also not instrumented. (We are currently enhancing Daikon to inspect interfaces and all collection classes.)

Finally, 5 missing annotations were needed to suppress ESC/Java warnings about exceptions. `MapQuick` handles catastrophic failure (such as filesystem errors) by raising an unchecked exception. The user must disable ESC's verification of these exceptions, as they can never be proven to be absent. This step requires user intervention no matter the tool, since specifying which catastrophic errors to ignore is an engineering decision.

The remaining 6 missing invariants arose from distinct causes that cannot be generalized, and that do not individually add insight.

5. Remaining challenges

Fully automatic generation and checking of program specifications is not always possible. This section categorizes problems we encountered in our experimental investigation. These limitations fall into three general categories: problems with the target programs, problems with the test suites, and problems with the Daikon and ESC/Java tools.

5.1 Target programs

One 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, in programs used in testing research, and elsewhere.) In this case, the desired invariant is not a true invariant of the program.

Program errors may prevent verification even if the error does not falsify a necessary invariant. The test suite may not reveal the error, so the correct specification will be generated. However, it will fail to verify because the static checker will discover possible executions that violate the invariant.

Our experiments revealed an error 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 fault, but Daikon reported that the vector elements were always non-null on entry to `toString`, leading to discovery of the error. The error is corrected in JDK 1.3. (We were not aware of the error at the time of our experiments.)

As another example of a likely error that we detected, 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`.

5.2 Test suites

Another challenge to generation is deficient or missing test suites. In general, realistic test suites tend to produce verifiable specifications, while poor verification results indicate specific failures in testing.

If the executions induced 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 partial semantics of a program. This is surprising, even on small programs, because reliably inferring patterns from small datasets is difficult. Furthermore, larger programs are not necessarily any better, because some components may be underexercised in the test suite. (For example, a main routine may only be run once.)

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

Unit tests — tests that check specific boundary values of procedures in a single module in isolation — were less successful. This may seem counter-intuitive, since unit tests often achieve code coverage and generally attempt to cover boundary cases of the module. However, in specifically targeting boundary cases, unit tests utilize the module in ways statistically unlike the application itself, throwing off the statistical techniques used in Daikon. Equally importantly, unit tests tend to contain few calls, preventing statistical inference.

When the initial test suites came from textbooks or were unit tests that 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 has a suggestion for 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 that were never performed on a full (or empty) stack and a queue implemented via an array that never wrapped 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, creating a test suite that induces accurate invariants is little or no more difficult than creating a general test suite. In short, poor verification results indicate specific failures in testing, and reasonably-sized and realistic test suites are able to accurately capture semantics of a program.

5.3 Inherent limitations of any tool

Every tool contains a *bias*: the grammar of properties that it can detect or verify. Properties beyond that grammar are insurmountable obstacles to automatic verification, so there are specifications beyond the capabilities of any particular tool.

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. Daikon does not include such invariants because they are of insufficiently general applicability, nor can ESC/Java verify such a property. (Users can add new invariants for Daikon to detect by writing a Java class that satisfies an interface with four methods.)

As another example, neither Daikon nor ESC/Java operates with invariants over strings. As a result, our combined system did not detect or verify that object invariants hold at the exit from a constructor or other method that interprets a string argument, even though the system showed that other methods maintain the invariant.

As a final 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:

```
(\forall int i;
 (0 <= i && i < theArray.length) ==>
  (theArray[i] == null) <==>
   ((currentSize == 0) ||
    ((currentSize > 0) &&
     (((front <= back) &&
       (i < front || i > back)) ||
      ((front > back) &&
       (i > back && i < front))))))
```

This omission prevented verification of many method postconditions.

5.4 Daikon

Aside from the problems inherent in any analysis tool, the tools used in this evaluation exhibited additional problems that prevented immediate verification. Daikon had three deficiencies.

First, Daikon does not examine the contents of non-List java collections such as maps or sets. This prevents it from reporting type or nullness properties of the elements, but those properties are often needed by ESC/Java for verification.

Second, Daikon operates offline by examining traces written to disk by an instrumented version of the program under test. If many methods are instrumented, or if the program is long-running, storage and processing requirements can exceed available capacity.

Finally, Daikon uses Ajax [O'C01] to determine comparability of variables in Java programs. If two variables are incomparable, no invariants relating them should be generated or tested. Ajax fails on some large programs; all variables are considered comparable, and spurious invariants are generated and printed constraining unrelated quantities.

All three problems are currently being addressed in re-engineering efforts.

5.5 ESC/Java

ESC/Java'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.

ESCJML cannot express certain properties that Daikon reports. 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.

The full JML language permits method calls in assertions, includes \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 unchecked pragmas that cause ESC/Java to assume particular properties without proof, permitting it to complete verification despite its limitations.

6. 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 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. One reason the results should generalize is that both Daikon and ESC/Java operate modularly, one class at a time. Generating or verifying specifications for a single class of a large system is no harder than doing so for a system consisting of a single class.

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, its verification does require some specifications and input–output relations. Our system also verified additional 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 (see Section 5.2). However, Daikon's output makes it extremely easy to improve the test suites by indicating their deficiencies. Furthermore, existing system tests were adequate, and these are more likely to exist and often easier to produce.

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, or the dynamic analysis can focus on code or properties that stymie a static analysis.

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 soundness (up to the limitations of the verifier). 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 implications) that characterize the relation are a step in the right direction.

6.1 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 — as the limitations noted in Section 5 are overcome — 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, which is 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. (A user study we performed supports this hypothesis [Nim02].) 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.

7. 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 programming 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 [LDW00]. Value profiling [CFE97, SS98] can efficiently detect certain simple properties at runtime. Event traces can generate finite state machines that explicate system behavior [BG97, CW98]. 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 generating specifications for 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 limitations prohibit 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 infeasible, though for specific applications, contexts, or assumptions, efficient pointer analyses can be sufficiently accurate [Das00].

The LOOP project verified an object invariant in Java's `Vector` class [JvH$^+$98, HJv01]. The technique involved automatic translation of Java to PVS [ORS92, ORSvH95], user-specified goals, and user interaction with PVS.

Many other tools besides ESC/Java statically check specifications [Pfe92, EGHT94, Det96, NCOD97]. Examples of static verifiers that are connected with real programming languages include LCLint [EGHT94], ACL2 [KM97], LOOP [JvH$^+$98], Java PathFinder [HP00], and Bandera [CDH$^+$00]. These other systems have different strengths and weaknesses than ESC/Java, but few have the polish of its integration with a real programming language.

We are currently integrating Daikon with IOA [GLV97], a formal language for describing computational processes that are modeled using I/O automata [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], an interactive theorem-proving system for multisorted first-order logic. Daikon proposes goals, lemmas, and intermediate assertions for the theorem prover. Representation invariants can assist in proofs of properties 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 enough representation invariants to be proved, and current systems have trouble postulating them; some researchers consider that task harder than performing the proof [Weg74, BLS96, BBM97]. In preliminary experiments [NWEL02], users found Daikon of substantial help in proving Peterson's 2-process mutual exclusion algorithm (leading to a new proof that would not have otherwise been obtained), a cache coherence protocol, and Lamport's Paxos algorithm.

7.1 Houdini

The research most closely related to our integrated system is Houdini [FL01, FJL01], an annotation assistant for ESC/Java. (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.

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. Houdini's user interface permits users to ask why a candidate invariant was refuted; this capability is orthogonal to proposal of candidates. Finally, Houdini is not publicly available, so we cannot perform a direct comparison.

Combining 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. 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.

8. Conclusion

We have proposed and experimentally assessed a novel approach to producing 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 specifications that are consistent and sufficient for automatic verification 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. In our tests, the Daikon implementation generated specifications with high (over 90%) 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 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 and pre-implementation creation. Integrating static and dynamic techniques in our system produces benefits in each direction, because of their complementary strengths and weaknesses.

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, Stephen Garland, William Griswold, Daniel Jackson, Josh Kataoka, Rustan Leino, Greg Nelson, David Notkin, Gregg Rothermel, James Saxe, and Kevin Sullivan. The presentation was improved by comments from Stephen Garland, William Griswold, Rachel Pottinger, Steven Wolfman, and the anonymous referees.

This research was supported in part by NSF grant CCR-0133580 and CCR-9970985 and by a gift from NTT Corporation.

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.

[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.

[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.

[CW98] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, November 1998.

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

[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.

[Dod02] Nii Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT Dept. of EECS, 2002.

[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.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[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.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1 edition, 1991.

[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.

[Har02] Michael Harder. Improving test suites via generated specifications. Master's thesis, MIT Dept. of EECS, May 2002.

[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.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, May 2002.

[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.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, November 2001.

[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.

[Lam88] David Alex Lamb. *Software Engineering: Planning for Change*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[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`.

[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.

[LG01] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA, 2001.

[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.

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

[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.

[Nim02] Jeremy W. Nimmer. Automatic generation and checking of program specifications. Master's thesis, MIT Dept. of EECS, May 2002.

[NWEL02] Toh Ne Win, Michael Ernst, and Nancy Lynch. Static and dynamic analysis of I/O automata. Technical Report 841, MIT Lab for Computer Science, May 2002.

[O'C01] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 2001.

[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.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE TSE*, SE-12(2):251–257, February 1986.

[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.

[Pre92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, third edition, 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.

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

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE*, May 2002.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[Sem94] Semiconductor Industry Association. The national technology roadmap for semiconductors. San Jose, CA, 1994.

[Som96] Ian Sommerville. *Software Engineering*. Addison-Wesley, Wokingham, England, fifth edition, 1996.

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

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

[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.

A. Specifications

Specifications are used in many different stages of development, from requirements engineering to maintenance. Furthermore, specifications take a variety of forms, from a verbal description of customer requirements to a set of test cases or an executable prototype. In fact, there is no consensus regarding the definition of "specification" [Lam88, GJM91].

Our research uses *formal specifications*. We define a (formal, behavioral) specification as a precise mathematical abstraction of program behavior [LG01, Som96, Pre92]. This definition is standard, but our *use* of specifications is novel. Our specifications are generated automatically, after an executable implementation exists. Typically, software engineers are directed to write specifications before implementation, then to use them as implementation guides — or simply to obtain the benefit of having analyzed requirements at an early design stage [Som96].

Despite the benefits of having a specification before implementation, in practice few programmers write (formal or informal) specifications before coding. Nonetheless, it is useful to produce such documentation after the fact [PC86]. Obtaining a specification at any point during the development cycle is better than never having a specification at all. *Post hoc* specifications are also used in other fields of engineering. As one example, speed binning is a process whereby, after fabrication, microprocessors are tested to determine how fast they can run [Sem94]. Chips from a single batch may be sold with a variety of specified clock speeds.

Some authors define a specification as an *a priori* description of intended or desired behavior that is used in prescribed ways [Lam88, GJM91]. For our purposes, it is not useful to categorize whether a particular logical formula is a specification based on who wrote it, when, and in what mental state. (The latter is unknowable in any event.) Readers who prefer the alternative definition may replace the term "specification" by "description of program behavior" (and "invariant" by "program property") in the text of this paper.

We believe that there is great promise in extending specifications beyond their traditional genesis as pre-implementation expressions of requirements. One of the contributions of our research is the insight that this is both possible and desirable, along with evidence to back up this claim.