Automatic Generation and Checking of Program Specifications

Technical Report #MIT-LCS-TR-852

June 10, 2002

Jeremy W. Nimmer
MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
jwnimmer@lcs.mit.edu

Abstract

This thesis introduces the idea of combining automatic generation and checking of program specifications, assesses its efficacy, and suggests directions for future research.

Specifications are valuable in many aspects of program development, including design, coding, verification, testing, and maintenance. They also enhance programmers' understanding of code. Unfortunately, written specifications are usually absent from programs, depriving programmers and automated tools of their benefits.

This thesis shows how specifications can be accurately recovered using a two-stage system: propose likely specifications using a dynamic invariant detector, and then confirm the likely specification using a static checker. Their combination overcomes the weaknesses of each: dynamically detected properties can provide goals for static verification, and static verification can confirm properties proposed by a dynamic tool.

Experimental results demonstrate that the dynamic component of specification generation is accurate, even with limited test suites. Furthermore, a user study indicates that imprecision during generation is not a hindrance when evaluated with the help of a static checker. We also suggest how to enhance the sensitivity of our system by generating and checking context-sensitive properties.

This technical report is a reformatted version of the author's Masters thesis of the same title, published by MIT in May, 2002. He was advised by Michael Ernst, Assistant Professor of Electrical Engineering and Computer Science.

Contents

Chapter 1

Introduction

Specifications are valuable in all aspects of program development, including design, coding, verification, testing, optimization, and maintenance. They provide valuable documentation of data structures, algorithms, and program operation. As such, they aid program understanding, some level of which is a prerequisite to every program manipulation, from initial development to ongoing maintenance.

Specifications can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends. The near absence of explicit specifications in existing programs makes it all too easy for programmers to introduce errors while making changes. A specification that is established at one point is likely to be depended upon elsewhere, but if the original specification is not documented, much less the dependence, then it is easy for a programmer to violate it, introducing a bug in a distant part of the program.

Furthermore, if a formal specification exists in a machine-readable form, theorem-proving, dataflow analysis, model-checking, and other automated or semi-automated mechanisms can verify the correctness of a program with respect to the specification, confirm safety properties such as lack of bounds overruns or null dereferences, establish termination or response properties, and otherwise increase confidence in an implementation.

Despite the benefits of having a specification, written specifications are usually absent from programs, depriving programmers and automated tools of their benefits. Few programmers write them before implementation, and many use no written specification at all. Nonetheless, it is useful to produce such documentation after the fact [PC86]. Obtaining a specification at any point during development is better than never having a specification at all.

We propose and evaluate a novel approach to the recovery of specifications: generate them unsoundly and then check them. The generation step can take advantage of the efficiency of an unsound analysis, while the checking step is made tractable by the postulated specification. We demonstrate that our combined approach is both accurate on its own, and useful to programmers in reducing the burden of verification.

More generally, our research goal is to suggest, evaluate, and improve techniques that recover specifications from existing code in a reliable and useful way. We seek to explore how programmers apply recovered specifications to various tasks. In the end, users are what matter, not technology for its own sake.

The remainder of this chapter surveys uses for specifications (Section 1.1), summarizes previous solutions (Section 1.2), presents our approach (Section 1.3), summarizes our contributions (Section 1.4), and outlines the sequel (Section 1.5).

1.1 Uses for specifications

Specifications are useful, to humans and to tools, in all aspects of programming. This section lists some uses of specifications, to motivate why programmers care about them and why extracting them from programs is a worthwhile goal.

Document code. Specifications characterize certain aspects of program execution and provide valuable documentation of a program's operation, algorithms, and data structures. Documentation that is automatically extracted from the program is guaranteed to be up-to-date, unlike human-written information that may not be updated when the code is.

Check assumptions and avoid bugs. Specifications can be inserted into a program and tested as code evolves to ensure that detected specifications are not later violated. This checking can be done dynamically (via `assert` statements) or statically (via stylized annotations). Such techniques can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends. Program maintenance introduces errors, and anecdotally, many of these are due to violating specifications.

Validate test suites. Dynamically detected specifications can reveal as much about a test suite as about the program itself, because the properties reflect the program's execution over the test suite. A specification might reveal that the program manipulates a limited range of values, or that some variables are always in a particular relationship to one another. These properties may indicate insufficient coverage of program values, even if the suite meets structural coverage criteria such as statement coverage.

Bootstrap proofs. Automated or semi-automated mechanisms can verify the correctness of a program with respect to a specification, confirm safety properties, and otherwise increase confidence in an implementation. However, 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]. Generated program specifications could be fed into an automated system, relieving the human of the need to fully hand-annotate their programs — a task that few programmers are either skilled at or enjoy.

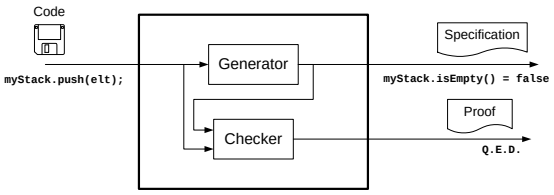


Figure 1.1: Generation and checking of program specifications results in a specification together with a proof of its consistency with the code. Our generator is the Daikon invariant detector, and our checker is the ESC/Java static checker.

1.2 Previous approaches

A full discussion of related work appears in Chapter 8, but we illustrate its main points here to characterize the deficiencies of previous approaches.

We consider the retrieval of a specification in two stages: the first is a *generation* step, where a specification is proposed; the second is a *checking* step, where the proposal is evaluated. In cases where the analysis used for generation guarantees that any result is acceptable (for instance, if the analysis is sound), checking is unnecessary. Depending on the tools used, a programmer may elect to iterate the process until an acceptable result is reached, or may use the specification for different tasks based and the result of the evaluation.

Past research has typically only addressed one stage, with one of four general approaches. A tool may generate a specification statically and soundly, so that checking is unnecessary, or may generate it dynamically and unsoundly, leaving checking to the programmer. A tool may check a model of the code and correctness properties that are both generated by the programmer, or may check a programmer-written specification against code meant to implement it.

Each of these four approaches presents an inadequate solution to the problem.

Static generation. 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 is often stymied by constructs, such as pointers, that are common in real-world programming languages. Manipulating complete representations of the heap incurs gross runtime and memory costs; heap approximations introduced to control costs weaken results. Many researchers consider it harder to determine what specification to propose than to do the checking itself [Weg74, WS76, MW77, Els74, BLS96, BBM97].

Dynamic generation. 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 programmers reluctant to depend on the results even in other contexts because of uncertainty as to their reliability.

Model checking. Specifications over models of the code are common for protocol or algorithmic verification. A programmer formally writes both a description of the system and its desired properties. Automated tools attempt to prove the desired properties, or to find counterexamples that falsify them. However, the work done by hand to create the model can be tedious and is error-prone. Furthermore, to guarantee that executable code meets the desired properties, the models still have to be matched to the code, which is a difficult task in itself.

Static checking. In the case of theorem-proving or program verification, the analysis frequently requires explicit goals or annotations, and often also summaries of unchecked code. These annotations usually must be supplied by the programmer, a task that programmers find tedious, difficult, and unrewarding, and therefore often refuse to perform [FJL01]. In contrast to model checking, annotations in a real program are often more numerous, since they are scattered throughout the codebase, and since real programs are often much larger than the models used in model checking. Additionally, languages used to annotate real programs may trade expressiveness for verifiability, so programmers may have trouble writing the properties they desire.

1.3 Combined approach

Our combined approach of generation and checking of program specifications results in a specification together with a proof of its consistency with the code (see Figure 1.1). Our approach addresses both the generation and checking stages of specification recovery, and utilizes techniques whose strengths complement each other's weaknesses.

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. Our system operates in three steps. First, it runs Daikon, which outputs a list of likely invariants obtained from running the target program over a test suite. 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 rerunning test suites, or by editing specific program annotations by hand.

The combination of the two tools overcomes the weaknesses

of each: dynamically generated properties can 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 analysis is much better than relying on only one component, or performing an error-prone hand analysis.

The remainder of this section describes our generated specifications, presents both realized and possible applications, and argues that our solution is useful.

1.3.1 Invariants as a specification

A formal specification is a precise, often mathematical, description of a program's behavior. Specifications often state properties of data structures, such as representation 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 of the variable at the time the procedure is invoked and max' represents the value 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 also describe the program's actual behavior, which may vary from the programmer's intended behavior. Finally, the specifications are also unsound: as described in Section 2.2, the proposed properties are likely, but not guaranteed, to hold. Through interaction with a checker that points out unverifiable properties, a programmer may remove any inaccuracies.

1.3.2 Applications of generated specifications

These aspects of generated specifications suggest certain uses while limiting others. Our research shows that the specifications are useful in verifying the lack of runtime exceptions. In contrast, using them as a template for automatic test case generation might add little value, since the specifications already reflect the programs' behavior over a test suite. As long as the programmer is aware of the specifications' characteristics, though, many applications are possible.

The fact that the generated specifications reflect actual rather than intended behavior is also not as detrimental as it may seem. 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 with the help of a static checker, guaranteeing the absence of certain errors and adding confidence to future maintenance tasks.

In addition to the results contained in this work, generated specifications have been shown to be useful for program refactoring [KEGN01], theorem proving [NWE02], test suite generation [Har02], and anomaly and bug detection [ECGN01, 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. One of the contributions of this work is that our generated specifications are accurate.

1.3.3 Weak formalisms and partial solutions are useful

Some may take exception to the title *Automatic Generation and Checking of Program Specifications*, or our approach to a solution. The system is "not automatic": users might have to repeat steps or edit annotations before a checkable specification is available. Also, the specification is incomplete: our system does not recover the "whole thing". While both criticisms are true in a sense, we disagree on both points.

Many specifications exist for a piece of code, each providing a different level of specificity. In theory, some analysis could be sound, complete, and recover a full formal specification. However, such a specification is infeasible to generate (by hand, or mechanically) or check (in most cases). In practice, partial specifications (such as types) are easy to write and understand, and have been widely accepted.

Furthermore, we allow that the system need not be perfect, nor completely automatic. It is not meant to be used in isolation, but will be used by programmers. Therefore, asking the programmer to do a small amount of work is acceptable, as long as there is benefit to doing so. Getting the programmer half-way there is better than getting them nowhere at all.

1.4 Contributions

The thesis of this research is that program specifications may be accurately generated from program source code by a combination of dynamic and static analyses, and that the resulting specifications are useful to programmers.

The first contribution of this research is the idea of producing specifications from a combination of dynamic and static analyses. Their combination overcomes the weaknesses of each: dynamically detected properties can 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 analysis is much better than relying on only one component, or performing an error-prone hand analysis.

The second contribution of this research is the implementation of a system to dynamically detect then statically verify program specifications. While each component had previously existed in isolation, we have improved Daikon to better suit the needs of ESC/Java, and created tools to assist their combination.

The third contribution of this research is the experimental assessment of the accuracy of the dynamic component of specification generation. We demonstrate that useful aspects of program semantics are present in test executions, as measured by verifiability of generated specifications. Even limited test suites accurately characterize general execution properties.

The fourth contribution of this research is the experimental assessment of the usefulness of the generated specifications to users. We show that imprecision from the dynamic analysis is not a hindrance when its results are evaluated with the help of a static checker.

The fifth contribution of this research is the proposal and initial evaluation of techniques to enhance the scope, sensitivity, and scalability of our system. We suggest how to account for context-sensitive properties in both the generation and checking steps, and show how such information can assist program understanding, validate test suites, and form a statically verifiable specification. We also suggest an implementation technique helps enable Daikon to analyze large and longer-running programs.

1.5 Outline

The remainder of this work is organized as follows.

Chapter 2 describes our use of terminology. It also provides background on the dynamic invariant detector (Daikon) and static verifier (ESC) used by our system.

Chapter 3 provides an extended example of how the techniques suggested in this work are applied to a small program. The chapter shows how a user might verify a program with our system, and provides sample output to characterize the specifications generated by our system.

Chapter 4 describes an experiment that studies the accuracy of generated specifications, since producing specifications by dynamic analysis is potentially unsound. The generated specifications scored over 90% on precision, a measure of soundness, and on recall, a measure of completeness, indicating that Daikon is effective at generating consistent, sufficient specifications.

Chapter 5 describes an experiment that studies the utility of generated specifications to users. We quantitatively and qualitatively evaluate 41 users in a program verification task, comparing the effects of assistance by Daikon, another related tool, or no tool at all. Statistically significant results show that both tools improve task completion, but Daikon enables users to express a fuller specification.

Chapter 6 describes extensions to the system. We introduce techniques that enable context-sensitivity — the accounting for control flow information in an analysis — in both the generation and checking steps. We provide brief experimental evidence of its efficacy for program understanding, validating test suites, and forming a statically verifiable specification.

Chapter 7 describes an implementation technique for Daikon that utilizes additional information about a program's structural semantics. The technique helps Daikon to operate online and incrementally, thus enabling the examination of larger or longer-running programs.

Chapter 8 presents related work, outlining similar research and showing how our research stands in contrast. We describe previous analyses (static and dynamic generators, and static checkers) including a similar tool, Houdini, and present related applications of generated specifications.

Chapter 9 suggests future work to improve our techniques and their evaluation.

Chapter 10 concludes with a summary of contributions.

Chapter 2

Background

This section first explains our use of specifications, in order to frame our later results in the most understandable context. We then briefly describe 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 are published elsewhere, as cited below.

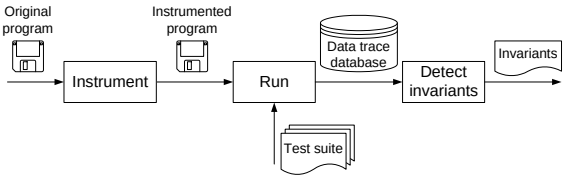


Figure 2.1: An overview of dynamic detection of invariants as implemented by the Daikon invariant detector.

2.1 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 thesis.

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.

2.2 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 2.1). 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. 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 ($x = a$) or a small set of constants ($x \in \{a,b,c\}$), lying in 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 "if $p \neq$ null then p.value $>$ x" and disjunctions such as "p.value $>$ limit or p.left $\in$ mytree". Implications result from splitting data into parts based on some condition and comparing the resulting invariants over each part; if mutually exclusive invariants appear in each part, they may be used as predicates in implications, and unconditional invariants in each part are composed into implications [EGKN99]. In this research, 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.

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

An invariant is reported only if there is adequate 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/`.

2.3 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. Its output also includes suggestions for correcting the problem and stylized counterexamples showing an execution path that violated the annotation or raised the exception.

In order to verify a program, ESC/Java translates it into a logical formula called a verification condition such that the program is correct if the verification condition is true [FS01]. The verification condition is then checked by the Simplify theorem-prover [Nel80].

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. We will not discuss ESC's checking strategy in more detail because this research treats ESC as a black box. (Its source code was until recently unavailable.)

ESC/Java is a successor to ESC/Modula-3. ESC/Java's annotation language 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.

ESC is publicly available from `http://research.compaq.com/SRC/esc/`.

Chapter 3

Extended Example

To illustrate the tools used in this work, the output they produce, and typical user interaction, this chapter presents an example of how a user might use our system on a small program to verify the absence of runtime exceptions.

Section 3.1 outlines how a user would verify the program by hand, while Section 3.2 shows how ESC/Java may be used. Section 3.3 demonstrates how Daikon complements ESC/Java for this task, and Section 3.4 discusses how the example relates to the sequel.

3.1 Verification by hand

Figure 3.1 shows the source code of a hypothetical stack implementation. We consider the case where a programmer wants to check that no runtime exceptions are generated by the `Stack` class, and first consider how a proof may be achieved by hand.

If each method is examined in isolation, it cannot be shown that errors are absent. Even the one-line `isFull` method may raise an exception if `elts` is null. One way to prove that the `elts` field is non-null within `isFull` is to prove that `elts` is non-null for any realizable instance of a `Stack`. Such a statement about all instances of a class is termed a *representation invariant* [LG01], also known as an *object invariant*. To prove a representation invariant, one must show that (1) each constructor establishes the invariant, (2) assuming that the invariant holds upon entry, each method maintains it, and (3) no external code may falsify the invariant.

Consider the proof that `elts` is always non-null. It is trivial to show (1) for the single constructor. Additionally, (2) is easily shown for the methods of Figure 3.1, since none of them sets the `elts` field. However, (3) is more interesting. Since `elts` is declared `protected` instead of `private`, subclasses of `Stack` are able to modify it. To complete the proof, we must show that all subclasses of `Stack` maintain the invariant. This task could certainly be achieved for one version of a codebase, but would have to be repeated whenever any new subclass was added, or any existing subclass was modified. (A programmer could elect to make the fields `private`, but this would not solve all problems, as noted in the next paragraph.)

A proof of legal array dereferencing in `push` and `pop` encounters a related problem. A `push` on a full stack, or `pop` on an empty one, causes an array bounds exception. To show that `pop` succeeds, we must only allow calls when `size` is strictly positive. Such a statement about allowed calls to a method is called a *precondition*, and permits proofs over methods to assume the stated precondition as long as all calling code is guaranteed to meet it.

In both of these cases, changes not local to `Stack` could invalidate a proof of its correctness. For this (and other) reasons, it is advantageous to mechanize the checking of the proof, so that as the code evolves over time, programmers can automatically check that past correctness guarantees are maintained. The ESC/Java tool (introduced in Section 2.3) performs this task.

3.2 Static verification with ESC/Java

By using a tool to verify the absence of runtime exceptions, programmers may have higher confidence that their code is free of certain errors. However, ESC/Java is unable to verify raw source code. Users must write annotations in their code that state, for instance, representation invariants and preconditions.

Without any annotations, ESC/Java reports warnings as shown in Figure 3.1. Using ESC/Java's output to guide reasoning similar to that described in the previous section, a user could add the annotations shown in Figure 3.2. This code contains the minimal number of annotations necessary to enable ESC/Java to verify `Stack`.

The minimally-annotated code contains two declarations, three representation invariants, and three preconditions. The two `spec_public` declarations are a detail of ESC/Java; they state that the specification may refer to the fields. The three representation invariants state that `elts` is never null, `size` is always in the proper range, and `elts` can store any type of object. The first precondition restricts an argument, while the two others restrict the state of the `Stack`.

Given these annotations, ESC/Java is able to statically verify that `Stack` never encounters any runtime exceptions (as long as all code in the system is checked with ESC/Java as well). While adding eight annotations may seem like an inconsequential amount of work, the annotations are about 50%

```
public class Stack {

  protected Object[] elts;
  protected int      size;

  public Stack(int capacity) {
    elts = new Object[capacity];   // Attempt to allocate array of negative length
    size = 0;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public boolean isFull() {
    return size == elts.length;    // Null dereference
  }

  public void push(Object x) {
    elts[size++] = x;              // Null dereference
                                   // Negative array index
                                   // Array index too large
                                   // RHS not a subtype of array element type
  }

  public Object pop() {
    Object item = elts[--size];    // Null dereference
                                   // Negative array index
                                   // Array index too large
    elts[size] = null;
    return item;
  }

}
```

Figure 3.1: Original source of `Stack.java`. Comments have been added to show the warnings issued when checked by ESC/Java.

of the original non-blank, non-punctuation lines. Annotating the whole of a large program may surpass what a programmer is willing to do. A tool that assisted the user in this task could overcome such a difficulty.

3.3 Assistance from Daikon

The annotations given in Figure 3.2 form an (incomplete) specification for `Stack`. The representation invariants specify properties that the class and its subclasses must maintain, while the preconditions specify when a method may legally be called. Since Daikon is able to propose likely specifications for a class, it may assist the user in the verification task.

Given a certain test suite (not shown), Daikon produces the output in Figure 3.3. Each program point's output is separated by a horizontal line, and the name of each program point follows the line. The `Stack:::OBJECT` point states representation invariants, while `ENTER` points state preconditions and `EXIT` points state postconditions.

While we will not explain every line of the output here, we note that this proposed specification is broader than the minimal one shown in Figure 3.2. In particular, the representation invariant is stronger (exactly the live elements of `elts` are non-null), as are postconditions on the constructor (`elts` is sized to `capacity`), `push` (`size` is incremented; prefix of `elts` is unchanged), and `pop` (`size` is decremented).

A tool packaged with Daikon can insert this proposed specification into the source code as ESC/Java annotations, resulting in the annotations shown in Figure 3.4.

When ESC/Java is run on this automatically-annotated version of the source code, it warns that the postcondition on `pop` involving `String` may not be met. The test suite pushed objects of varied types onto the stack, but only popped in the case where `String`s had been pushed, so Daikon postulated the (incorrect) postcondition on `pop`. The user, after being warned about that annotation by ESC/Java, could either remove it by hand, or improve the test suite by adding cases where non-`String`s were popped. After either change, the revised version of the `Stack` annotations would enable ESC/Java to verify the absence of runtime exceptions.

3.4 Discussion

The example in this chapter shows how Daikon and ESC/Java may be used together to verify the lack of runtime exceptions in a tiny program through the automatic generation and checking of the program's specification.

The specification verified in the previous section stated more than ESC/Java's minimal specification would require. It also reported useful properties of the implementation, such as

```
public class Stack {

  /*@ invariant elts != null; */
  /*@ invariant \typeof(elts) == \type(java.lang.Object[]); */
  /*@ invariant size >= 0; */
  /*@ invariant size <= elts.length; */
  /*@ invariant (\forall int i; (0 <= i && i < size) ==> (elts[i] != null)); */
  /*@ invariant (\forall int i; (size <= i && i < elts.length) ==>
                                (elts[i] == null)); */

  /*@ spec_public */ protected Object[] elts;
  /*@ invariant elts.owner == this; */
  /*@ spec_public */ protected int      size;

  /*@ requires capacity >= 0; */
  /*@ ensures capacity == elts.length; */
  /*@ ensures size == 0; */
  public Stack(int capacity) {
    elts = new Object[capacity];
    /*@ set elts.owner = this; */
    size = 0;
  }

  /*@ ensures (\result == true)  ==  (size == 0); */
  public boolean isEmpty() { ... }

  /*@ ensures (size <= elts.length-1)  ==  (\result == false); */
  public boolean isFull() { ... }

  /*@ requires x != null; */
  /*@ requires size < elts.length; */
  /*@ modifies elts[*], size; */
  /*@ ensures x == elts[\old(size)]; */
  /*@ ensures size == \old(size) + 1; */
  /*@ ensures (\forall int i; (0 <= i && i < \old(size)) ==>
                              (elts[i] == \old(elts[i]))); */
  public void push(Object x) { ... }

  /*@ requires size >= 1; */
  /*@ modifies elts[*], size; */
  /*@ ensures \result == \old(elts[size-1]); */
  /*@ ensures \result != null; */
  /*@ ensures \typeof(\result) == \type(java.lang.String) */
  /*@ ensures size == \old(size) - 1; */
  public Object pop() { ... }

}
```

Figure 3.4: Output of Daikon automatically merged into `Stack` source as ESC/Java annotations. Unchanged method bodies have been removed to save space.

that precisely the live elements of the stack's internal array are non-null, or that `pop` returns a non-null result. Finally, it pointed out a deficiency in the test suite, allowing for its easy correction.

We also note that the specifications do not have to be used to verify the absence of runtime exceptions. By generating and checking specifications automatically, we have conveniently produced a specification that matches an existing codebase. Such a specification may be useful for many tasks, as described in Section 1.3.2.

```
public class Stack {

  /*@ spec_public */ protected Object[] elts;
  /*@ spec_public */ protected int      size;

  //@ invariant elts != null
  //@ invariant 0 <= size && size <= elts.length
  //@ invariant \typeof(elts) == \type(Object[])

  //@ requires capacity >= 0
  public Stack(int capacity) {
    elts = new Object[capacity];
    size = 0;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public boolean isFull() {
    return size == elts.length;
  }

  //@ requires size < elts.length
  public void push(Object x) {
    elts[size++] = x;
  }

  //@ requires size >= 1
  public Object pop() {
    Object item = elts[--size];
    elts[size] = null;
    return item;
  }

}
```

Figure 3.2: Minimal annotations (comments beginning with @) required by ESC/Java to verify the absence of runtime exceptions.

```
================================================
Stack:::OBJECT
this.elts != null
this.elts.class == "java.lang.Object[]"
this.size >= 0
this.size <= size(this.elts[])
this.elts[0..this.size-1] elements != null
this.elts[this.size..] elements == null
================================================
Stack.Stack(int):::ENTER
capacity >= 0
================================================
Stack.Stack(int):::EXIT
capacity == orig(capacity) == size(this.elts[])
this.size == 0
================================================
Stack.isEmpty():::EXIT
this.elts == orig(this.elts)
this.elts[] == orig(this.elts[])
this.size == orig(this.size)
(return == true)  <==>  (this.size == 0)
================================================
Stack.isFull():::EXIT
this.elts == orig(this.elts)
this.elts[] == orig(this.elts[])
this.size == orig(this.size)
(this.size <= size(this.elts[])-1)  <==>
  (return == false)
================================================
Stack.push(java.lang.Object):::ENTER
x != null
this.size < size(this.elts[])
================================================
Stack.push(java.lang.Object):::EXIT
x == orig(x) == this.elts[orig(this.size)]
this.elts == orig(this.elts)
this.size == orig(this.size) + 1
this.elts[0..orig(this.size)-1] ==
  orig(this.elts[0..this.size-1])
================================================
Stack.pop():::ENTER
this.size >= 1
================================================
Stack.pop():::EXIT
this.elts == orig(this.elts)
return == orig(this.elts[this.size-1])
return != null
return.class == "java.lang.String"
this.size == orig(this.size) - 1
```

Figure 3.3: Output of Daikon for `Stack`, given a certain test suite (not shown). For readability, output has been edited to rearrange ordering.

Chapter 4

Accuracy Evaluation

This chapter presents an experiment that evaluates the accuracy of the generation step of our system: dynamic invariant detection. In contrast, Chapter 5 studies how varying accuracy can affect users during static checking.

4.1 Introduction

We evaluate the accuracy of specification generation by measuring the static verifiability of its result. Specifically, we measure 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. Measured against this verification requirement, the generated specifications scored nearly 90% on precision, a measure of soundness, and on recall, a measure of completeness. Precision and recall are standard measures from information retrieval [Sal68, vR79]

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.

4.2 Methodology

This section presents the programs studied and the data analyzed.

4.2.1 Programs and test suites

We analyzed the programs listed in Figure 4.1. `DisjSets`, `StackAr`, and `QueueAr` come from a data structures textbook [Wei99]; `Vector` is part of the Java standard library; and the remaining seven programs are solutions to assignments in a programming course at MIT [MIT01].

Figure 4.2 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 4.4.2) 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. When methods are insufficiently tested, the reported specification is stronger than what the programmer has in mind, pointing out that the (true, unwritten) specification is not fully covered by the tests.

4.2.2 Measurements

As described in Section 1.3, 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 Dai-

Program	LOC	NCNB	Meth.	Description
FixedSizeSet	76	28	6	set represented by a bitvector
DisjSets	75	29	4	disjoint sets supporting union, find
StackAr	114	50	8	stack represented by an array
QueueAr	116	56	7	queue represented by an array
Graph	180	99	17	generic graph data structure
GeoSegment	269	116	16	pair of points on the earth
RatNum	276	139	19	rational number
StreetNumberSet	303	201	13	collection of numeric ranges
Vector	536	202	28	`java.util.Vector` growable array
RatPoly	853	498	42	polynomial over rational numbers
MapQuick	2088	1031	113	driving directions query processor
Total	4886	2449	273	

Figure 4.1: Description of programs studied (Section 4.2.1). "LOC" is the total lines of code. "NCNB" is the non-comment, non-blank lines of code. "Meth" is the number of methods.

		Original Test Suite					
	Size	Size		Coverage		Time	
Program	NCNB	NCNB	Calls	Stmt	Branch	Instr	Daikon
FixedSizeSet	28	0	0	0.00	0.00	0	0
DisjSets	29	27	745	0.67	0.70	1	6
StackAr	50	11	72	0.60	0.56	0	3
QueueAr	56	11	52	0.68	0.65	0	12
Graph	99	Sys	3k	0.76	0.54	1	3
GeoSegment	116	Sys	695k	0.89	0.75	138	455
RatNum	139	Sys	58k	0.96	0.94	7	28
StreetNumberSet	201	165	50k	0.95	0.93	7	29
Vector	202	0	0	0.00	0.00	0	0
RatPoly	498	382	88k	0.94	0.89	27	98
MapQuick	1031	445	3.31M	0.66	0.61	660	1759

		Augmented Test Suite					
	Size	Size		Coverage		Time	
Program	NCNB	+NCNB	Calls	Stmt	Branch	Instr	Daikon
FixedSizeSet	28	39	12k	1.00	1.00	2	10
DisjSets	29	15	12k	1.00	0.90	3	18
StackAr	50	39	1k	0.64	0.63	0	4
QueueAr	56	54	8k	0.71	0.71	1	11
Graph	99	1	3k	0.76	0.54	1	3
GeoSegment	116	0	695k	0.89	0.75	138	455
RatNum	139	39	114k	0.96	0.94	14	56
StreetNumberSet	201	151	197k	0.95	0.95	12	44
Vector	202	190	22k	0.90	0.90	7	37
RatPoly	498	51	102k	0.96	0.92	38	139
MapQuick	1031	49	3.37M	0.67	0.71	673	1704

Figure 4.2: Characterization of original and augmented 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 specifically focused on the specified program, but tests a higher-level system that contains the program (see Section 4.4.2). "+NCNB" is the number of lines added to yield the results described in Section 4.3. "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.

kon. 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 4.3), 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 $\frac{\text{Verif}}{\text{Verif+Unver}}$. Recall, a measure of completeness, is defined as $\frac{\text{Verif}}{\text{Verif+Miss}}$. 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.

4.3 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 4.3.1 summarizes our experiments, while Sections 4.3.2 through 4.3.4 discuss three example programs in detail to characterize the generated specifications and provide an intuition about the output of our system. Section 4.4 summarizes the problems the system may encounter.

4.3.1 Summary

We performed eleven experiments, as shown in Figure 4.3. As described in Section 4.2, Daikon's output is automatically inserted into the target program as annotations, which are edited by hand (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.

4.3.2 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. It is similar to the example of Chapter 3, but contains more methods.

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.)

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.

4.3.3 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 4.3 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

Program	Program size			Number of invariants			Accuracy	
	LOC	NCNB	Meth.	Verif.	Unver.	Miss.	Prec.	Recall
FixedSizeSet	76	28	6	16	0	0	1.00	1.00
DisjSets	75	29	4	32	0	0	1.00	1.00
StackAr	114	50	8	25	0	0	1.00	1.00
QueueAr	116	56	7	42	0	13	1.00	0.76
Graph	180	99	17	15	0	2	1.00	0.88
GeoSegment	269	116	16	38	0	0	1.00	1.00
RatNum	276	139	19	25	2	1	0.93	0.96
StreetNumberSet	303	201	13	22	7	1	0.76	0.96
Vector	536	202	28	100	2	2	0.98	0.98
RatPoly	853	498	42	70	10	1	0.88	0.99
MapQuick	2088	1031	113	145	3	35	0.98	0.81
Total	4886	2449	273	530	24	55	0.96	0.91

Figure 4.3: Summary of specifications recovered, in terms 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.

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 4.2) 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.

4.3.4 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 that 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 4.3 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 ignorance 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.

4.4 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.

4.4.1 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`.

4.4.2 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.

4.4.3 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. (The invariants are shown in Figure 5.8 on page 29). This omission prevented verification of many method postconditions. In a user study (Chapter 5), no subject was able to write this invariant or an equivalent one.

4.4.4 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.

4.4.5 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.

4.5 Discussion

The most surprising result of this experiment 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, including representation invariants 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 4.4.2). 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.

Chapter 5

User Evaluation

This chapter describes an evaluation of the effectiveness of two tools to assist static checking. We quantitatively and qualitatively evaluate 41 users in a program verification task over three small programs, using ESC/Java as the static checker, and either Houdini (described in Section 5.2) or Daikon for assistance. Our experimental evaluation reflects the effectiveness of each tool, validates our approach to generating and checking specifications, and also indicates important considerations for creating future assistant tools.

5.1 Introduction

Static checking can verify the absence of errors in a program, but often requires written annotations or specifications. As a result, static checking can be difficult to use effectively: it can be difficult to determine a specification and tedious to annotate programs. Automated tools that aid the annotation process can decrease the cost of static checking and enable it to be more widely used.

We evaluate techniques for easing the annotation burden by applying two annotation assistant tools, one static (Houdini) and one dynamic (Daikon), to the problem of annotating Java programs for the ESC/Java static checker.

Statistically significant results show that both tools contribute to success, and neither harms users in a measurable way. Additionally, Houdini helps users express properties using fewer annotations. Finally, Daikon helps users express more true and potentially useful properties than strictly required for a specific task, with no time penalty.

Interviews indicate that beginning users found Daikon to be helpful but were concerned with its verbosity on poor test suites; that Houdini was of uncertain benefit due to concerns with its speed and opaqueness; that static checking could be of potential practical use; and that both assistance tools to have unique benefits.

The remainder of this chapter is organized as follows. Section 5.2 briefly describes the Houdini tool. Section 5.3 presents our methodology. Sections 5.4 and 5.5 report quantitative and qualitative results. Section 5.6 examines the results and Section 5.7 concludes.

5.2 Houdini

Houdini is an annotation assistant for ESC/Java [FL01, FJL01]. (A similar system was previously proposed by Rintanen [Rin00].) It augments user-written annotations with additional ones that complement them. This permits users to write fewer annotations and end up with less cluttered, but still automatically verifiable, programs.

Houdini postulates a candidate annotation set and computes the greatest subset of it that is valid for a particular program. It repeatedly invokes the ESC/Java checker as a subroutine and discards unprovable postulated annotations, until no more are refuted. If even one required annotation is missing, then Houdini eliminates all other annotations that depend on it. Correctness of the loop depends on two properties: the set of true annotations returned by the checker is a subset of the annotations passed in, and if a particular annotation is not refuted, then adding additional annotations to the input set does not cause the annotation to be refuted.

Houdini's initial candidate invariants are all possible arithmetic and (in)equality comparisons among fields (and "interesting constants" such as -1, 0, 1, array lengths, `null`, `true`, and `false`), and also assertions that array elements are non-`null`. Many elements of this initial set are mutually contradictory.

According to its creators, over 30% of Houdini's guessed annotations are verified, and it tends to reduce the number of ESC/Java warnings by a factor of 2–5 [FL01]. With the assistance of Houdini, programmers may only need to insert about one annotation per 100 lines of code.

5.2.1 Emulation

Houdini was not publicly available at the time of this experiment, so we were forced to re-implement it from published descriptions. For convenience in this section only, we will call our emulation "Whodini."

For each program in our study, we constructed the complete set of true invariants in Houdini's grammar and used that as Whodini's initial candidate invariants. This is a subset of Houdini's initial candidate set and a superset of verifiable Houdini invariants, so Whodini is guaranteed to behave exactly like published descriptions of Houdini, except that it will run

faster. Fewer iterations of the loop (fewer invocations of ESC/Java) are required to eliminate unverifiable invariants, because there are many fewer such invariants in Whodini's set. Whodini typically takes 10–60 seconds to run on the programs used for this study.

5.3 Methodology

The section presents our experimental methodology and its rationale. We are interested in studying what factors affect a user's performance in a program verification task. We are primarily interested in the effect of the annotation assistant on performance, or its effect in combination with other factors. For instance, we study how the level of imprecision of Daikon's output affects user performance.

Section 5.3.1 presents the participants' task. Section 5.3.2 describes participant selection and demographics. Section 5.3.3 details the experimental design. Section 5.3.4 describes how the data was collected and analyzed.

5.3.1 User Task

Study participants were posed the goal of writing annotations to enable ESC/Java to verify the absence of runtime errors. Each participant performed this task on two different programs in sequence.

Before beginning, participants received a packet (reproduced in Appendix A) containing 6 pages of written instructions, printouts of the programs they would annotate, and photocopies of figures and text explaining the programs, from the book from which we obtained the programs. The written instructions explained the task, our motivation, ESC/Java and its annotation syntax, and (briefly) the assistance tools. The instructions also led participants through an 11-step exercise using ESC/Java on a sample program. The sample program, an implementation of fixed-size sets, contained examples of all of the annotations participants would have to write to complete the task (`@invariant`, `@requires`, `@modifies`, `@ensures`, `@exsures`). Participants could spend up to 30 minutes reading the instructions, working through the exercises, and further familiarizing themselves with ESC/Java. Participants received hyperlinks to an electronic copy of the ESC/Java user's manual [LNS00] and quick reference [Ser00].

The instructions explained the programming task as follows.

> Two classes will be presented: an abstract data type (ADT) and a class that calls it. You will create and/or edit annotations in the source code of the ADT. Your goal is to enable ESC/Java to verify that neither the ADT nor the calling code may ever terminate with a runtime exception. That is, when ESC/Java produces no warnings or errors on both the ADT and the calling code, your task is complete.

The ADT source code was taken from a data structures textbook [Wei99]. We wrote the client (the calling code). Participants were instructed to edit only annotations of the ADT — neither the ADT implementation code nor any part of the client was to be edited.

Program	Methods	NCNB LOC		Minimal Annot.
		ADT	Client	
`DisjSets`	4	28	29	17
`StackAr`	8	49	79	23
`QueueAr`	7	55	70	32

Figure 5.1: Characteristics of programs used in the study. "Methods" is the number of methods in the ADT. "NCNB LOC" is the non-comment, non-blank lines of code in either the ADT or the client. "Minimal Annot" is the minimal number of annotations necessary to complete the task.

We met with each participant to review the packet and ensure that expectations were clear. Then, participants worked at their own desks, unsupervised. (Participants logged into our Linux machine and ran ESC/Java there.) Some participants received assistance from Houdini or Daikon, while others did not. Participants could ask us questions during the study. We addressed environment problems (e.g., tools crashing) but did not answer questions regarding the task itself.

After the participant finished the second annotation task, we conducted a 20-minute exit interview. (Section 5.5 presents qualitative results from the interviews.)

Programs

The three programs used for this study were taken from a data structures textbook [Wei99]. Figure 5.1 gives some of their characteristics.

We selected three programs for variety. The `DisjSets` class is an implementation of disjoint sets supporting `union` and `find` operations without path compression or weighted union. The original code provided only an unsafe `union` operation, so we added a safe `union` operation as well. The `StackAr` class is a fixed-capacity stack represented by an array, while the `QueueAr` class is a fixed-capacity wrap-around queue represented by an array. We fixed a bug in the `makeEmpty` method of both to set all storage to `null`. In `QueueAr`, we also inlined a private helper method, since ESC/Java requires that object invariants hold at private method entry and exit, which was not the case for this helper.

We selected these programs because they are relatively straightforward ADTs. The programs are not trivial for the annotation task, but are not so large as to be unmanageable. We expect results on small programs such as these to scale to larger programs, since annotations required for verifying absence of runtime errors overwhelmingly focus on class-specific properties

5.3.2 Participants

A total of 47 users participated in the study, but six were disqualified, leaving data from 41 participants total. Five participants were disqualified because they did not follow the writ-

	Mean	Dev.	Min.	Max.
Years of college education	7.0	2.6	3	14
Years programming	11.7	5.0	4	23
Years Java programming	3.6	1.5	1	7

	Frequencies
Usual environment	Unix 59%; Win 13%; both 29%
Writes asserts in code	"often" 30%; less frequently 70%
. . . in comments	"often" 23%; less frequently 77%
Gender	male 89%; female 11%

Figure 5.2: Demographics of study participants. "Dev" is standard deviation.

Program	Suite	NCNB LOC	Calls Stat.	Calls Dyn.	Coverage Stmt.	Coverage Bran.	Prec.	Rec.
DisjSets	Tiny	23	5	389	0.67	0.70	0.65	0.57
	Small	28	5	1219	1.00	0.90	0.71	0.74
	Good	43	13	11809	1.00	1.00	0.94	0.97
StackAr	Tiny	14	4	32	0.60	0.56	0.54	0.52
	Small	24	5	141	0.64	0.63	0.83	0.73
	Good	54	14	2783	0.96	0.94	1.00	0.95
QueueAr	Tiny	16	4	32	0.68	0.65	0.37	0.44
	Small	44	10	490	0.71	0.71	0.47	0.56
	Good	66	10	7652	0.94	0.88	0.74	0.75

Figure 5.3: Test suites used for Daikon runs. "NCNB LOC" is the non-comment, non-blank lines of code. "Stat" and "Dyn" are the static and dynamic number of calls to the ADT. "Stmt" and "Bran" indicate the statement and branch coverage of the test suite. "Prec" is precision, a measure of correctness, and "Rec" is recall, a measure of completeness; see Section 5.3.4.

ten instructions; the sixth was disqualified because the participant declined to finish the experiment. We also ran 6 trial participants to refine the instructions, task, and tools; we do not include data from those participants. All participants were volunteers.

Figure 5.2 provides background information on the 41 participants. Participants were experienced programmers and were familiar with Java programming, but none had ever used ESC/Java, Houdini, or Daikon before. Participants had at least 3 years of post-high-school education, and most were graduate students in Computer Science at MIT or the University of Washington. No participants were members of the author's research group.

Participants reported their primary development environment (options: Unix, Windows, or both), whether they write assert statements in code (options: never, rarely, sometimes, often, usually, always), and whether they write assertions in comments (same options). While the distributions are similar, participants frequently reported opposite answers for assertions in code vs. comments — very few participants frequently wrote assertions in both code and comments.

5.3.3 Experimental Design

Treatments

The experiment used five experimental treatments: a control group, Houdini, and three Daikon groups.

No matter the treatment, all users started with a minimal set of 3 to 6 annotations already inserted in the program. This minimal set of ESC/Java annotations included `spec_public` annotations on all private fields, permitting them to be mentioned in specifications, and `owner` annotations for all private `Object` fields, indicating that they are not arbitrarily modified by external code. We provided these annotations in order to reduce both the work done and the background knowledge required of participants; they confuse many users and are not the interesting part of the task. Our tools add this boilerplate automatically. These annotations are ignored during data collection, since users never edit them.

Control. This group was given the original program without any help from an annotation assistant. To complete the task, these participants had to add enough annotations on their own so that ESC/Java could verify the program. The minimal number of such invariants is given in Figure 5.1.

Houdini. This group was provided the same source code as the control group, but used a version of ESC/Java enhanced with (our re-implementation of) Houdini. Houdini was automatically invoked (and a message printed) when the user ran `escjava`. To complete the task, these participants had to add enough annotations so that Houdini could do the rest of the work towards verifying the program.

Daikon. The three Daikon groups received a program into which Daikon output had been automatically inserted as ESC/Java annotations. To complete the task, these participants had to both remove unverifiable invariants inferred by Daikon and also add other uninferred annotations.

These participants ran an unmodified version of ESC/Java. There was no sense also supplying Houdini to participants who were given Daikon annotations. Daikon always produces all the invariants that Houdini might infer, so adding Houdini's inference would be of no benefit to users.

Participants were not provided the test suite and did not run Daikon themselves. (Daikon took only a few seconds to run on these programs.) While it would be interesting to study the process where users are able to both edit the test suite and the annotations, we chose to study only annotation correction. Looking at the entire task introduces a number of additional factors that would have been difficult to control experimentally.

To study the effect of test suite size on users' experience, we used three Daikon treatments with varying underlying test suite sizes (See Figure 5.3). In later sections, discussion of the "Daikon" treatment refers any of the three below treatments; we use a subscript to refer to a specific treatment.

Daikon$_{\text{tiny}}$: This group received Daikon output produced using example calling code that was supplied along with the ADT. The example code usually involved just a few calls, with many methods never called and few corner cases exposed

(see Figure 5.3). We call these the "tiny" test suites, but the term "test suites" is charitable. They are really just examples of how to call the ADT.

These suites are much less exhaustive than would be used in practice. Our rationale for using them is that in rare circumstances users may not already have a good test suite, or they may be unwilling to collect operational profiles. If Daikon produces relatively few desired invariants and relatively many test-suite-specific invariants, it might hinder rather than help the annotation process; we wished to examine that circumstance.

Daikon$_{\text{small}}$: This group received a program into which a different set of Daikon output had been inserted. The Daikon output for these participants was produced from an augmented form of the tiny test suite. The only changes were using the `Stack` and `Queue` containers polymorphically, and varying the sizes of the structures created (since the tiny test suites used a constant size).

When constructing these suites, the author limited himself to 3 minutes of wall clock time (including executing the test suites and running Daikon) for each of `DisjSets` and `StackAr`, and 5 minutes for `QueueAr`, in order to simulate low-cost testing methodology. As in the case of Daikon$_{\text{tiny}}$, use of these suites measures performance when invariants are detected from an inadequate test suite – one worse than programmers typically use in practice. We call these the "small" suites.

Daikon$_{\text{good}}$: This group received Daikon output produced using a test suite constructed from scratch, geared toward testing the code in full, instead of giving sample uses. These adequate test suites took the author about half an hour to produce.

Assignment of treatments

There are a total of 150 possible experimental configurations: there are six choices of program pairs; there are five possible treatments for the first program; and there are five possible treatments for the second program. No participant annotated the same program twice, but participants could be assigned the same treatment on both trials.

In order to reduce the number of subjects, we ran only a subset of the 150 configurations. We assigned the first 32 participants to configurations using a randomized partial factorial design, then filled in the gaps with the remaining participants. (Participants who were disqualified had their places taken by subsequent participants, in order to preserve balance.)

5.3.4 Analysis

This section explains what quantities we measured, how we measured them, and what values we derive from the direct measurements.

Variable	Domain
Independent	
Annotation assistant	none, Houdini, Daikon$_{\text{T,S,G}}$
Program	StackAr, QueueAr, DisjSets
Experience	first trial, second trial
Location	MIT, Univ. of Wash.
Usual environment	Unix, Windows, both
Years of college education	
Years programming	
Years Java programming	
Writes asserts in code	never, rarely, sometimes,
Writes asserts in comments	often, usually, always
Dependent	
Success	yes, no
Time spent	up to 60 minutes
Final written answer	set of annotations (Fig. 5.6)
Nearest verifiable answer	set of annotations (Fig. 5.6)

Figure 5.4: Variables measured (Section 5.3.4), and their domain (set of possible values). We also analyze computed values, such as precision and recall (Section 5.3.4).

Quantities Measured

We are interested in studying what factors affect a user's performance in a program verification task. Figure 5.4 lists the independent and dependent variables we measured to help answer this question.

We are primarily interested in the effect of the annotation assistant on performance, or its effect in combination with other factors. We also measure other potentially related independent variables in order to identify additional factors that have an effect, and to lend confidence to effects shown by the assistant.

We measure four quantities to evaluate performance. Success (whether the user completed the task) and the time spent are straightforward. We also compare the set of annotations in a user's answer to the annotations in the nearest correct answer. When users do not finish the task, this is their degree of success.

The next section describes how we measure the sets of annotations, and the following section describes how we numerically relate the sets.

Measurement Techniques

This section explains how we measured the variables in Figure 5.4. The annotation assistant, program, and experience are derived from the experimental configuration. Participants reported the other independent variables. For dependent variables, success was measured by running ESC/Java on the solution. Time spent was reported by the user. If the user was unsuccessful and gave up early, we rounded the time up to 60 minutes.

The most complex measurements were finding the nearest correct answer and determining the set of invariants the user had written. To find the nearest correct answer, we repeatedly ran ESC/Java and made small edits to the user's answer until

One lexical annotation; two semantic annotations:

```
//@ ensures (x != null) && (i = \old(i) + 1)
```

Two lexical annotations; one semantic annotation:

```
//@ requires (arg > 0)
//@ ensures (arg > 0)
```

Figure 5.5: Distinction between lexical and semantic annotations. The last annotation is implied because `arg` is not declared to be modified across the call.

there were no warnings, taking care to make as few changes as possible. A potential source of error is that we missed a nearer answer. However, many edits were straightforward, such as adding an annotation that must be present in all answers. Removing annotations was also straightforward: incorrect annotations cause warnings from ESC/Java, so the statements may be easily identified and removed. The most significant risk is declining to add an annotation that would prevent removal of others that depend on it. We were aware of this risk and were careful to avoid it.

Determining the set of invariants present in source code requires great care. First, we distinguish annotations based on whether they are class annotations (`@invariant`) or method annotations (`@requires`, `@modifies`, `@ensures`, or `@exsures`). Then, we count invariants lexically and semantically (Figure 5.5).

Lexical annotation measurements count the textual lines of annotations in the source file. Essentially, the lexical count is the number of stylized comments in the file.

Semantic annotation measurements count how many distinct properties are expressed. The size of the semantic set is related to the lexical count. However, an annotation may express multiple properties, for instance if it contains a conjunction. Additionally, an annotation may not express any properties, if a user writes essentially the same annotation twice or writes an annotation that is implied by another one.

We measure the semantic set of annotations (in addition to the lexical count) because it is truer to the actual content of the annotations: it removes ambiguities due to users' syntactic choices, and it accounts for unexpressed but logically derivable properties.

To measure the semantic set, we created specially-formed calling code to grade the ADT. For each semantic property to be checked, we wrote one method in the calling code. The method instructs ESC/Java to assume certain conditions and check others; the specific conditions depend on the property being checked. For instance, to check a class invariant, the grading method takes a non-null instance of the type as an argument and asserts that the invariant holds for the argument. For preconditions, the grading method attempts one call that meets the condition, and one call that breaks it. If the first passes but the second fails, the precondition is present. Similar techniques exist for modifies clauses and postconditions.

The grading system may fail if users write bizarre or unexpected annotations, so we verified all results of the grading system by hand. We found that mistakes were infrequent, and where they did occur, we counted the invariants by hand. However, by and large, this automatic grading system ensures that our measurements are reliable, unbiased, and reproducible.

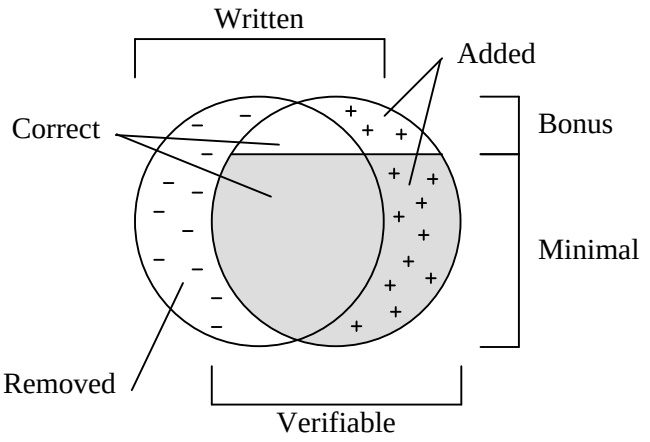


Figure 5.6: Visualization of written and verifiable sets of annotations. The left circle represents the set of invariants written by the user; the right circle represents the nearest verifiable set. The overlap is correct invariants, while the fringes are additions or deletions. In general, the nearest verifiable set (the right circle) is not necessarily the smallest verifiable set (the shaded region). See Section 5.3.4 for details.

Computed Values

From the quantities directly measured (Figure 5.4) we computed additional variables to compare results across differing programs, and to highlight illuminating quantities.

Figure 5.6 visualizes the measured and derived quantities. The measured quantities are the user's final annotations ("Written") and the nearest verifiable set ("Verifiable"). Both are sets of annotations as measured by the *semantic* counting technique described in the preceding section. If the user was not successful, then the written and verifiable sets differ. To create a verifiable set, we removed unverifiable annotations ("Removed"), and added other annotations ("Added"). Verifiable annotations written by the user fall into the middle section ("Correct"). Finally, compared with the minimal possible verifiable answer ("Minimal"), the user may have expressed additional annotations ("Bonus"). The minimal set does not depend on the user's written annotations.

From these measurements, we compute several values.

Precision, a measure of correctness, is defined as the fraction of the written annotations that are correct ($\frac{\text{correct}}{\text{written}}$). Precision is always between 0 and 1. The fewer "$-$" symbols in Figure 5.6, the higher the precision. We measure precision to determine the correctness of a user's statements.

Recall, a measure of completeness, is defined as the fraction of the verifiable annotations that are written ($\frac{\text{correct}}{\text{verifiable}}$). Recall is always between 0 and 1. The fewer "$+$" symbols

in Figure 5.6, the higher the recall. We measure recall to determine how many necessary statements a user wrote. In this study, recall was a good measure of a user's progress in the allotted time, since recall varied more than precision.

Density measures the average amount of semantic information per lexical statement. We measure density by dividing the size of the user's semantic annotation set by the lexical annotation count. We measure density to determine the textual efficiency of a user's annotations. If a user writes many redundant properties, density is low. Additionally, when Houdini is present, programs may have higher invariant densities, since some properties are inferred and need not be present as explicit annotations.

Redundancy measures unnecessary effort expended by Houdini users. For the Houdini trials, we computed the semantic set of properties written explicitly by the user and then restricted this set to properties that Houdini could have inferred, producing a set of redundantly-written properties. We created a fraction from this set by dividing its size by the number of properties inferable by Houdini for that program. This fraction lies between 0 and 1 and measures the redundancy level of users' annotations in relation to the annotations that Houdini may infer.

Bonus, a measure of additional information, is defined as the ratio of verifiable annotations to the minimal set ($\frac{\text{verifiable}}{\text{minimal}}$). The larger the top section of the right circle in Figure 5.6, the larger the bonus. We measured bonus to judge, in a program-independent way, the amount of properties the user expressed.

Bonus annotations are true properties that were not needed for the verification task studied in this experiment, but may be helpful for other tasks or provide valuable documentation. They generally specify the behavior of the class in more detail. In `StackAr`, for instance, frequently-written bonus annotations specify that unused storage is set to null, and that `push` and `pop` operations may modify only the top of the stack (instead of any element of the stack).

Miscellaneous Values

We studied the statistical significance of other computed variables, such as the effect of the first trial's treatment on the second trial, the first trial's program on the second trial, the distinction between object or method annotations, whether the user used Windows, etc. None of these factors was statistically significant.

5.4 Quantitative Results

This section presents quantitative results of the experiment, which are summarized in Figure 5.7. Each subsection discusses one dependent variable and the factors that predict it.

We analyzed all of the sensible combinations of variables listed in Section 5.3.4. All comparisons discussed below are statistically significant at the $p = .10$ level. Comparisons that are not discussed below are not statistically significant at the $p = .10$ level. To control experimentwise error rate (EER), we always used a multiple range test [Rya59] rather than direct pairwise comparisons, and all of our tests took account of experimental imbalance. As a result of these safeguards, some large absolute differences in means are not reported here. For instance, $\text{Daikon}_{\text{good}}$ users averaged 45 minutes, while users with no tool averaged 53 minutes, but the result was not statistically justified. The lack of statistical significance was typically due to small sample sizes and variations in individual performance.

Effects predicted by *Treatment*

Depend. var.	None	Houd.	$D_{T,S}$	D_{good}	Confidence
Success ($\neg$`Q`)	36%	71%			$p = 0.03$
Recall	72%	85%			$p = 0.02$
Density	1.00	1.78	1.00		$p < 0.01$
Bonus	1.25		1.47	1.75	$p < 0.01$

Effects predicted by *Program*

Depend. var.	DisjSets	StackAr	QueueAr	Confidence
Success	72%	52%	0%	$p < 0.01$
Time	44	50	60	$p < 0.01$
Precision	98%		88%	$p = 0.01$
Recall	95%	85%	64%	$p < 0.01$

Effects predicted by *Experience*

Depend. var.	First trial	Second trial	Confidence
Redundancy	36%	65%	$p = 0.02$

Figure 5.7: Summary of important numerical results. For each independent variable, we report the dependent variables that it predicted and their means. If a mean spans multiple columns, the effect was statistically indistinguishable among those columns. Variables are explained in Section 5.3.4, and effects are detailed in Section 5.4. ($\neg$`Q` means that the result holds only if `QueueAr` data, for which no users were successful, is omitted.)

5.4.1 Success

We measured user success to determine what factors may generally help or hurt a user; we were particularly interested in the effect of the annotation assistant. Perhaps Daikon's annotations are too imprecise or burdensome to be useful, or perhaps Houdini's longer runtime prevents users from making progress.

The only factor that unconditionally predicted success was the identity of the program under test ($p < 0.01$). Success rates were 72% for `DisjSets`, 52% for `StackAr`, and 0% for `QueueAr`. This variety was expected, since the programs were selected to be of varying difficulty. However, we did not expect `QueueAr` to have no successful users.

If the data from `QueueAr` trials are removed, then whether a tool was used predicted success ($p = 0.03$). Users with no tool succeed 36% of the time, while users with either Daikon or Houdini succeeded 71% of the time (the effects of the as-

sistants were statistically indistinguishable).

These results suggest that some programs are difficult to annotate, whether or not either Daikon or Houdini is assisting. `QueueAr` requires complex invariants due to ESC/Java's expression language. (Section 5.6 considers this issue in more detail.) Furthermore, for more easily-expressible invariants, tool assistance improves the success rate by a factor of two.

5.4.2 Time

We measured time to determine what factors may speed or slow a user. Perhaps evaluating Daikon's suggested annotations takes extra time, or perhaps Houdini's longer runtime adds to total time spent.

As with success, a major predictor for time spent was the program under test ($p < 0.01$). Mean times (in minutes) were 44 for `DisjSets`, 50 for `StackAr`, and 60 for `QueueAr`. If the `QueueAr` trials are again removed, then experience also predicted time ($p = 0.08$). First-time users averaged 49 minutes, while second-time users averaged 43.

Since no other factors predict time, even within successful users, these results suggest that the presence of the assistance tools neither slow down nor speed up the annotation process, at least for these programs. This is a positive result for both tools since the time spent was not affected, yet other measures were improved.

5.4.3 Precision

We measured precision, the fraction of a user's annotations that are verifiable, to determine what factors influence the correctness of a user's statements. Successful users have a precision of 100% by definition. Perhaps the annotations supplied by Daikon cause unsuccessful users to have incorrect annotations remaining when time is up.

As expected, precision was predicted by the program under test ($p = 0.01$). Together, `StackAr` and `DisjSets` were indistinguishable, and had a mean precision of 98%, while `QueueAr` had a mean of 88%.

These results suggest that high precision is relatively easy to achieve in the time allotted. Notably, Daikon users did not have significantly different precision than other users. Since ESC/Java reports which annotations are unverifiable, perhaps users find it relatively straightforward to correct them. Supporting qualitative results appear in Section 5.5.3.

5.4.4 Recall

We measured recall, the fraction of the necessary annotations that the user writes, to determine what factors influence the progress a user makes. Successful users have a recall of 100% by definition. Perhaps the assistants enabled the users to make more progress in the allotted time.

As expected, recall was predicted by the program under test ($p < 0.01$). Mean recall was 95% for `DisjSets`, 85% for `StackAr`, and 64% for `QueueAr`.

Recall was also predicted by treatment ($p = 0.02$). Mean recall increased from 72% to 85% when any tool was used, but the effects among tools were statistically indistinguishable. If the `QueueAr` trials are removed, the effect is more pronounced ($p < 0.01$): mean recall increased from 76% to 95% when any tool was used.

This suggests that both tools assist users in making progress toward a specific task, and are equally good at doing so. More surprisingly, users of Daikon were just as effective starting from a tiny test suite as from a good one – a comprehensive test suite was not required to enhance recall.

5.4.5 Density

We measured the semantic information per lexical statement to determine what factors influence the textual efficiency of a user's annotations. Perhaps the annotations provided by Daikon cause users to be inefficiently verbose, or perhaps Houdini enables users to state properties using fewer written annotations.

The only factor that predicted the density was treatment ($p < 0.01$). Houdini users had a mean density of 1.78 semantic properties per written statement, while non-Houdini users had a mean of 1.00.

Notably, density in Daikon trials was statistically indistinguishable from the null treatment, with a mean 2% better than no tool.

5.4.6 Redundancy

For Houdini trials, we measured the redundancy of users' annotations in relation to the annotations that Houdini may infer (the computation is explained in Section 5.3.4). Perhaps users understand Houdini's abilities and do not repeat its efforts, or perhaps users repeat annotations that Houdini could have inferred.

The only factor that predicted redundancy was experience ($p = 0.02$). Users on the first trial had a mean redundancy of 36%, while users on the second trial had a mean redundancy of 65%. Surprisingly, second-time users were more likely to write annotations that would have been inferred by Houdini.

Overall, users redundantly wrote 51% of the available method annotations. For object invariants, though, users wrote more redundant annotations as program difficulty increased (14% for `DisjSets`, 45% for `StackAr`, and 60% for `QueueAr`).

These results suggest that users in our study (who have little Houdini experience) do not understand what annotations Houdini may infer, and frequently write out inferable invariants. This effect is more prevalent if users are more familiar with what invariants are necessary, or if the program under study is difficult. Related qualitative results are presented in Section 5.5.2.

5.4.7 Bonus

We measured the relative size of a user's verifiable set of annotations compared to the minimal verifiable set of annotations for the same program. The ratio describes the total semantic amount of information the user expressed in annotations.

The only factor that predicted the bonus information was the tool used ($p < 0.01$). Users with the $\text{Daikon}_{\text{good}}$ treatment had a mean bonus of 1.75. Users with $\text{Daikon}_{\text{tiny}}$ or $\text{Daikon}_{\text{small}}$ had a mean bonus of 1.47, while users with Houdini or no tool had a mean of 1.25.

Examining the same measurements for successful users is informative, since the verifiable set for unsuccessful users includes annotations that they did not write (consider the top three $+$'s in Figure 5.6). The results held true: for successful users, the treatment also predicted bonus information ($p < 0.01$). $\text{Daikon}_{\text{good}}$ users had a mean ratio of 1.71, $\text{Daikon}_{\text{tiny}}$ and $\text{Daikon}_{\text{small}}$ users had a mean ratio of 1.50, while others had a mean of 1.21.

These results suggest that Daikon users express a broader range of verifiable properties, with no harm to time or success at the given task. For instance, in `StackAr` or `QueueAr` Daikon users frequently specified that unused storage is set to null, and that mutator operations may modify only a single element. Many Daikon users also wrote full specifications for the methods, describing exactly the result of each operation. These bonus properties were not needed for the task studied in this experiment, but they make the specification more complete. This completeness may be helpful for other tasks, and it provides valuable documentation.

5.5 Qualitative Results

This section presents qualitative results gathered from exit interviews conducted after each user finished all tasks. Section 5.5.1 briefly covers general feedback. Section 5.5.2 describes experiences with Houdini and Section 5.5.3 describes experiences with Daikon.

5.5.1 General

While the main goal of this experiment is to study the utility of invariant inference tools, exploring users' overall experience provides background to help evaluate the more specific results of tool assistance.

Incremental approach

Users reported that annotating the program incrementally was not efficient. That is, running ESC/Java and using the warnings to figure out what to add was less efficient than spending a few minutes studying the problem and then writing all seemingly relevant annotations in one go. Four users switched to the latter approach for the second half of the experiment and improved their relative time and success (but the data does not statistically justify a conclusion). Additionally, a few users who worked incrementally for the whole experiment believed that an initial attempt at writing relevant annotations at the start would have helped. All users who were given Daikon annotations decided to work incrementally.

Confusing warnings

Users reported difficulty in figuring out how to eliminate ESC/Java warnings. Users said that ESC/Java's suggested fixes were obvious and unhelpful. The `exsures` annotations were particularly troublesome, since many users did not realize that the exceptional post-conditions referred to post-state values of the variables. Instead, users interpreted them like Javadoc `throws` clauses, which refer to pre-state conditions that cause the exception. Additionally, users wanted to call pure methods in annotations, define helper macros for frequently-used predicates, and form closures, but none of these are possible in ESC/Java's annotation language.

Users reported that ESC/Java's execution trace information — the specific execution path leading to a potential error — was helpful in diagnosing problems. Many users found the trace to be sufficient, while other users wanted more specific information, such as concrete variable values that would have caused the exception.

5.5.2 Houdini

Users' descriptions of experiences with Houdini indicate its strengths and weaknesses. A total of 14 participants used Houdini for at least one program. Three users had positive opinions, five were neutral, and six were negative.

Easier with less clutter

The positive opinions were of two types. In the first, users expressed that Houdini "enabled me to be faster overall." Houdini appeared to ease the annotation burden, but users could not identify specific reasons short of "I didn't have to write as much down." In the second, users reported that Houdini was "easier than Daikon," often because they "didn't have to see everything." In short, the potential benefits of Houdini — easing annotation burden and leaving source code cleaner — were realized for some users.

No noticeable effect

The five users with neutral opinions did not notice any benefit from Houdini, nor did they feel that Houdini hurt them in any way. As it operated in the background, no effect was manifest.

Slow and confusing

Users' main complaint was that Houdini was too slow. Some users who had previously worked incrementally began making more edits between ESC/Java runs, potentially making erroneous edits harder to track down.

Additionally, users reported that it was difficult to figure out what Houdini was doing (or could be doing); this result is supported by Section 5.4.6. Some users wished that the annotations inferred by Houdini could have been shown to them upon request, to aid in understanding what properties were already present. (The actual Houdini tool contains a front-end that is capable of showing verified and refuted annotations [FL01], but it was not available for use in this study.)

5.5.3 Daikon

Benefits

Of the users who received Daikon's invariants, about half commented that they were certainly helpful. Users frequently suggested that the provided annotations were useful as a way to become familiar with the annotation syntax. Additionally, the annotations provided an intuition of what invariants should be considered, even if what was provided was not accurate. Finally, provided object invariants were appreciated because some users found object invariants more difficult to discover than method annotations.

Overload

About a third of the $\text{Daikon}_{\text{tiny}}$ and $\text{Daikon}_{\text{small}}$ users suggested that they were frustrated with the textual size of the provided annotations. Users reported that the annotations had an obscuring effect on the code, or were overwhelming. Some users said they were able to learn to cope with the size, while others said the size was a persistent problem. $\text{Daikon}_{\text{good}}$ users reported no problems with the output size.

Incorrect suggestions

A significant question is how incorrect suggestions from an unsound tool affect users. A majority of users reported that removing incorrect annotations provided by Daikon was easy. Others reported that many removals were easy, but some particularly complex statements took a while to evaluate for correctness. Users commented that, for `ensures` annotations, ESC/Java warning messages quickly pointed out conditions that did not hold, so it was likely that the annotation was in error.

This suggests that when a user sees a warning about an invalid provided annotation and is able to understand the meaning of the annotation, deciding its correctness is relatively easy. The difficulty only arises when ESC/Java is not able to verify a correct annotation (or the absence of a runtime error), and the user has to deduce what else to add.

The one exception to this characterization occurred for users who were annotating the `DisjSets` class. In the test suites used with $\text{Daikon}_{\text{tiny}}$ and $\text{Daikon}_{\text{small}}$ to generate the annotations, the parent of every element happened to have a lower index than the child. The diagrams provided to users from the data structures textbook also displayed this property, so some users initially believed it to be true and spent time trying to verify annotations derived from this property. Nevertheless, the property indicated a major deficiency in the test suite, which a programmer would wish to correct if his or her task was broader than the simple one used for this experiment.

5.5.4 Uses in practice

A number of participants believed that using a tool like ESC/Java in their own programming efforts would be useful and worthwhile. Specifically, users suggested that it would be especially beneficial if they were more experienced with the tool, if it was integrated in a GUI environment, if syntactic hurdles could be overcome, or if they needed to check a large existing system.

A small number of participants believed that ESC/Java would not be useful in practice. Some users cared more about global correctness properties, while others preferred validating by building a better test suite rather than annotating programs. One user suggested that ESC/Java would only be useful if testing was not applicable.

However, the majority of participants were conditionally positive. Users reported that they might use ESC/Java occasionally, or that the idea was useful but annotating programs was too cumbersome. Others suggested that writing and checking only a few properties (not the absence of exceptions) would be useful. Some users felt that the system was useful, but annotations as comments were distracting, while others felt that the annotations improved documentation.

In short, many users saw promise in the technique, but few were satisfied with the existing application.

5.6 Discussion

We have carefully evaluated how static and dynamic assistance tools affect users in a verification task. This section considers potential threats to the experimental design.

Any experimental study approximates what would be observed in real life, and selects some set of relevant factors to explore. This study attempts to provide an accurate exploration of program verification, but it is useful to consider potential threats to the results, and how we addressed them.

Disparity in programmer skill is known to be very large and might influence our results. However, we have taken programmers from a group (MIT and UW graduate students in computer science) in which disparity may be less than in the general population of all programmers. Furthermore, we perform statistical significance tests rather than merely comparing means. The significance tests account for spread. (In many cases large disparities in means were not reported as significant.) Use of $p = .10$ means that if two samples are randomly selected from the same population, then there is only a 10% chance that they are (incorrectly) reported as statistically significantly different. If the samples are drawn from populations with different means, the chance of making such an error is even less.

// Only live elements are non-null

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

// Array indices are consistent

```
((currentSize == back - front + 1) ||
 (currentSize == back - front + 1 +
   ((currentSize > 0) ? theArray.length :
                       -theArray.length)))
```

Figure 5.8: Object invariants required by `QueueAr` for ESC/Java verification. The written form of the invariants is made more complicated by the limits of ESC/Java's annotation language.

Our re-implementation of Houdini may also affect the results. As explained in Section 5.2.1, our implementation will produce the same results and will run faster, since it postulates only the true subset of the invariants in Houdini's grammar.

Experienced ESC/Java users may be affected by tool assistance in different ways than users with only an hour of experience, but it is infeasible to study users with experience. We know of no significant number of experienced ESC/Java users, and training a user to become an expert in ESC/Java takes months, so is too time-consuming for both the researcher and the user.

We limited users to a few hours of time at most, which restricted the size of the programs we could study. Our results may not generalize to larger programs, if larger programs have more complex invariants. That no users succeeded on `QueueAr` may also seem to support this argument. However, we believe that `QueueAr` is not representative because it requires unusually complicated invariants, whereas larger programs do not generally require complicated invariants.

We chose `QueueAr` as a particularly difficult example for the user study; it guarantees that unused storage is set to null via the invariants shown in Figure 5.8. These invariants were particularly difficult for users to generate (and Daikon does not suggest such complicated properties).

However, larger programs do not necessarily require such complex invariants. In general, even if a program is large and maintains complex invariants, not all invariants are needed for ESC/Java's verification goals, and the necessary invariants are both relatively simple and sparse. Chapter 4 (Figure 4.3 on page 17) showed that a 498-line program required one annotation per 7.0 non-comment, non-blank lines of code, and a 1031-line program required one annotation per 7.1 lines, whereas `QueueAr` required an annotation every 1.7 lines. Additionally, neither of the larger programs required complex invariants.

Finally, since both Daikon and ESC/Java are modular (operate on one class at a time), the invariants detected and required are local to one class, which both simplifies the scaling of the tools and limits the difficulty for a user.

5.7 Conclusion

Static checking is a useful software engineering practice. It can reveal errors that would otherwise be detected only during testing or even deployment. However, static checkers require explicit goals for checking, and often also summaries of unchecked code. To study the effectiveness of two potential specification generators, we have evaluated two annotation assistance tools: Houdini and Daikon. A number of important conclusions can be drawn from this experiment.

Foremost is the importance of sensible and efficient generation of candidate specifications. In cases like `QueueAr` where no tool postulates the difficult but necessary invariants, users spend a significant amount of time trying to discover it. Efficiency is also important: even with a subset of its grammar in use, Houdini users complained of its speed. While Daikon fared no better than Houdini in terms of success or time, $\text{Daikon}_{\text{good}}$ users were given a more complete candidate specification, leading to a notably higher bonus. A larger bonus means that a more complete specification was achieved, which may be useful for other tasks or serve as a form of documentation. In short, efficient generation of candidate invariants is an important task, and one Daikon performs well.

Users suggested that a permanent (final) set of annotations should not clutter the code. Therefore Houdini's method of inferring unwritten properties should be helpful. However, the results show that hiding even simple inferred annotations is confusing (leading to redundancy). Perhaps a user interface that allows users to toggle annotations could help.

A key result is that assistants need not be perfect, supporting our claim (Section 1.3.3) that partial solutions are useful. Daikon's output contained numerous incorrect invariants (see Figure 5.3), but Daikon did not slow down users, nor did it decrease their precision. In fact, Daikon helped users write more correct annotations, and this effect was magnified as better suites were used. Users effortlessly discard poor suggestions and readily take advantage of correct ones.

Furthermore, Daikon can use even the tiniest "test suites" (as small as 32 dynamic calls) to produce useful annotations. Even test suites drawn from example uses of the program noticeably increase user recall. In short, test suites sufficiently large to enable Daikon to be an effective annotation assistant should be easy to come by.

In sum, both assistants increased users' effectiveness in completing the program verification task, but each had its own benefits. Houdini was effective at reducing clutter, but was too slow to use. Daikon was just as effective, increased the amount of true properties expressed by the user, and was effective even in the presence of limited test suites. These results validate our approach to the automatic generation and checking of program specifications. Users are effectively able to refine inaccurate output of an unsound generation step into a verifiable specification more effectively than writing a specification from scratch.

Chapter 6

Context sensitivity

While we have shown our techniques to be useful, enhancements could improve them further. The chapter suggests the addition of context sensitivity (Section 6.1) during both generation (Section 6.2) and checking (Section 6.3).

6.1 Introduction

A *context-sensitive* analysis accounts for control flow information, as illustrated by the following example. Consider a method that picks an element from a collection:

```
public Object pick(Collection c) {
  return c.iterator().next();
}
```

If a certain caller always passes in an ordered collection, such as a `Vector`, a context-sensitive analysis that inferred the specification based on each individual caller could report the post-condition `\result == c[0]`: the first element of the `Vector` was always returned to that caller. Similarly, if a caller always passed a collection with a homogeneous element type, a context-sensitive analysis could report `\typeof(\result) == \elementtype(c)`: the return type of `pick` matched the element type of its argument.

In contrast, *context-insensitive* specification generation may report that the argument and result are always non-null, but if `pick` is called from multiple locations, inference is unlikely to produce any invariants over `pick` that relate the argument to the return value.

6.2 Context-sensitive generation

As shown above, invariant detection that reports properties that are true for all calls to a procedure may fail to report properties that hold only in certain contexts. This section proposes uses for context-sensitive specification generation, describes a technique to implement it, and provides brief experimental evidence of its efficacy.

6.2.1 Applications

Context-sensitive invariants can reveal differences in the way methods are used based upon the nature of the caller. If these properties can be effectively recovered, they may be applied to a variety of applications:

Program understanding. In a sense, context-sensitivity may be seen as an incremental refinement of Daikon, permitting it to be more specific in its output. For example, Daikon's output concerning elements of a polymorphic container type might report $\mathsf{elt.class} \in \{\mathsf{String}, \mathsf{Integer}\}$: the elements are limited to two types. With context-sensitivity, Daikon would be able to report that the container was used with either `String`s or `Integer`s consistently — that all instances of the container were used homogeneously.

On the other hand, it is possible that context-sensitive invariants may reveal completely new information. A context-insensitive invariant detector may fail to find anything significant in calls made by a `Square` and `Rectangle` class to a `drawRect` method. However, a context-sensitive detector would notice that the `Square` always used an equal `width` and `height`. When code must be modified, understanding the different uses from varied call-sites may be useful, especially in methods with a complex behavior that is not all exercised during a single call.

Test suite evaluation. By reporting more properties that are true over a test suite, a more specific analysis may help identify unfortunately-true invariants that make the suite less general than it should be. Depending on the verbosity of the output, the properties may by processed automatically [Har02] or may be evaluated by programmers.

Optimization or refactoring identification. The most obvious example is partial specialization. When certain state has constant or constrained values depending on the origin of the call, it may be appropriate to split one method into several, one for each caller, or to move some of the code across the function boundary into the caller. Static analysis may be able to effect these optimizations when the parameters are literal constants, but exploiting dynamic information may give the programmer a deeper insight than is possible with a static analysis (such as with [KEGN01]).

Granularity

We can change the character of the analysis by varying the level of abstraction, or *granularity*, with which we distinguish different callers during inference. At the finest level, every

call-site is considered a distinct caller; at an intermediate level, all calls made from within the same method are considered together; and at the coarsest level, all calls made from within the same class are merged — progressively decreasing the "magnification" at which we inspect the caller.

Other methods of grouping are possible, such as by the thread of the caller, or by a time index as a program moves through different stages. We chose a lexical approach due to its ease of implementation and likelihood to both match programmer intuition and produce useful results.

We note that as granularity becomes coarser, a wider variety of callers is needed to produce any distinction. Indeed, we suspect that the benefits of a context-sensitive analysis will be most evident with larger test suites, or in examples where the classes under test have a greater number of different clients.

Implicit vs. Explicit

It is worthwhile to consider how invariants produced by a context-sensitive analysis may contribute to a generated specification. It may seem that new properties will describe facts about the users of the class, instead of about its own specification.

However, a context-sensitive analysis may be able to relate behavior that varies with the caller to input parameters or internal state by partitioning the set of samples in a novel way. The use of context information for partitioning has the power to distinguish behavioral aspects where formerly no pattern was apparent.

We categorize new invariants as *explicitly* contextual when the invariant makes explicit reference to the origin of a method call, or *implicitly* contextual when the invariant was discovered due to partitioning on a call-site but does not mention a caller explicitly. Implicitly contextual invariants will certainly contribute to the specification since they are no different than any other invariant. The effects of explicitly contextual invariants on a specification are explored in Section 6.3.

6.2.2 Implementation

We have implemented a context-sensitive dynamic invariant detector by augmenting Daikon's instrumentation and inference steps. We briefly describe our implementation techniques here.

We collect context information using a pre-pass to Daikon's normal instrumentation. We transform the program by augmenting every in-program procedure call with an additional argument, a numeric identifier that uniquely identifies the call-site. The identifier may be added as a formal parameter, or may be passed via global variable; we chose the former. Daikon's normal instrumentation then treats the argument the same as any other: its runtime value is written to a trace database as part of the record describing the procedure's pre-state.

To detect conditional invariants predicated on call-sites, we must supply Daikon with predicates that identify the call-sites at the granularity we desire. Since the mapping from a caller at some granularity to the set of identifiers that represent it is a function, the predicates necessarily have properties that can be used to form implications. We augmented Daikon to read side files produced during context instrumentation and form the appropriate predicates at inference-time.

Predicate generation can be done at different granularities while the instrumented source (and resulting trace) stay the same. This allows varying granularities to be applied to a trace without having to run the program every time the setting is changed.

6.2.3 Evaluation

Chapters 4 and 5 propose two ways to evaluate a generated specification: by studying its static verifiability, and by examining its utility to users. For the evaluation of our context-sensitive analysis, we repeat the accuracy experiments, but substitute a qualitative evaluation by the author for the user evaluation.

Static Verifiability

Given the additional sensitivity of our analysis, we were curious how the results of Chapter 4 might change. The output of our augmented system is a superset of its previous result, which may either enable additional proofs or add more unverifiable, test-suite dependent properties.

We repeated the experiments described in Chapter 4. Invariants that explicitly mentioned a call-site were deemed inexpressible in ESC/Java and thus discarded.

Interestingly, most of the 11 programs had no significant changes. Since many were a single ADT and its test suite, there was only one calling context, and no additional information could be gained. The only program with any large differences was `RatPoly`, described immediately below.

These results confirm our suspicion that small programs with one or two classes in isolation will not contain many context-specific invariants. Furthermore, we see that in some cases, the addition of sensitivity does not worsen the highly-accurate results already obtained.

RatPoly: polynomial over rational numbers

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.

We note that `RatPoly`'s implementation was written by a graduate student in computer science, and its test suite was written by software engineering instructors for the purpose of grading student assignments. We believe that the test suite authors wanted to produce a suite with an exceedingly good ability to find faults. Even with that motivation, though, the test suite lacked coverage of certain key values.

This author, who is familiar with the `RatPoly` code, was able to pick out the following properties of the program and its test suite in approximately five minutes, using the output

```
RatNum.approx():::ENTER
  (<Called from RatPoly.eval>)  ==>  (this.denom == 1)
RatNum.div(PolyCalc.RatNum):::ENTER
  (<Called from RatPoly.divAndRem>)  ==>  (arg.denom == 1)
RatNum.mul(PolyCalc.RatNum):::ENTER
  (<Called from RatPoly.divAndRem>)  ==>  (arg.numer >= 0)
```

Figure 6.1: Indications of test suite deficiencies, as found in partial output of context-sensitive invariant detection on `RatPoly`, using method-level granularity.

of our context-sensitive analysis. Three context-conditional invariants of `RatNum` stood out (Figure 6.1); each indicates a deficiency in the test suite. The first shows that when evaluating the polynomial to produce a floating-point value, `RatPoly.eval` only called `RatNum.approx` with integers; this indicates that only fraction-less polynomials were used while testing `eval`, a clear deficiency in the test suite. The same situation is evidenced by the second invariant: only integer-coefficient polynomials are divided. Furthermore, the third invariant shows that only positive divisors are used. These invariants point out flaws in the test suite and immediately suggest how to improve it, but do not show up without a context-sensitive analysis.

Two additional context-conditional invariants of `RatNum` also stood out (Figure 6.2); each indicates a property of `RatPoly`'s representation. The first reflects that zero-valued coefficients are never stored or manipulated, the second that `NaN` values are never stored or manipulated.

Finally, four context-conditional invariants of `RatPoly` stood out (Figure 6.3); each indicates an inefficient implementation choice and candidate for partial specialization. The first shows that `div` repeatedly uses the `parse` route to generate `NaN` polynomials (instead of having a static reference to a single instance). The second and third show that `scaleCoeff` is used in a limited way when called from `negate`, while the fourth shows that `divAndRem` only removes the first element of a vector. These usage patterns are candidates for partial specialization, with the potential for speed improvements.

In short, context-sensitive dynamic invariant detection revealed useful properties of both the program and its test suite that were not known beforehand, and that would have otherwise gone unnoticed.

6.2.4 Summary

We have presented techniques that augment a dynamic invariant detector to enable context-sensitivity, have described our implementation, and have provided experimental evidence of its utility. We introduce the distinction between *explicitly* and *implicitly* contextual invariants, and the idea of a *granularity* to context-sensitivity, where data from multiple paths is coalesced based upon the paths' lexical position in the source.

6.3 Context-sensitive checking

The previous section introduced context-sensitive invariant detection, the distinction between implicitly and explicitly contextual invariants, and how explicitly contextual invariants can assist program understanding and test suite validation. Explicitly contextual invariants may also help form a statically verifiable specification.

6.3.1 Polymorphic specifications

Polymorphic code provides generalized behavior that is specialized for the needs of the caller. The caller may, for instance, define types processed or stored ("a list of `String`s") or may supply a maximum storage requirement ("support up to `size` objects"). In one sense, the polymorphic code has just one specification, with parameters supplied by the client at time of use. However, in another sense, we can think of the code as having multiple specifications, one for each client. The multiple specifications are similar, but will in at least the portions that were parameterized — each is an instantiation of the single, parameterized specification given specific values for the parameters. If an analysis can recover these instantiations of the generalized specification and apply them at the proper locations in the code, it is at least as effective as an analysis that recovers the generalized specification.

We illustrate this concept with a hypothetical example of a specification generated without context-sensitivity (Figure 6.4). `IntSet` uses a container `MyVector` to store its elements. Even though `IntSet` only stores non-null `Integer`s, `MyVector`'s specification does not reflect this property because it has multiple callers. Therefore, `IntSet` fails to verify because `MyVector.get` is not in general constrained to return a non-null `Integer`. However, we can specialize `MyVector`'s specification for `IntSet` by defining a specification field `MyVector.user0` that enables a context specific to `IntSet` (Figure 6.5). With this addition, both classes would fully verify.

We can generalize this example as follows. If some instances of a class `B` are used in a limited way from another class `A`, and those instances of `B` are only used from `A`, then we can specialize a version of `B`'s specification to `A`'s context, potentially enabling more proofs within `A`'s code. The specialization would itself be checked, but would only be employed when checking `A`'s code. We term these *polymorphic specifications*.

```
RatNum.div(PolyCalc.RatNum):::ENTER
  (<Called from RatPoly.divAndRem>)  ==>  (arg.numer != 0)
RatNum.mul(PolyCalc.RatNum):::ENTER
  (<Called from RatPoly.divAndRem>)  ==>  (this.denom >= 1)
```

Figure 6.2: Indications of representation properties, as found in partial output of context-sensitive invariant detection on `RatPoly`, using method-level granularity.

```
RatPoly.parse(java.lang.String):::ENTER
  (<Called from RatPoly.div>)  ==>  (polyStr.toString == "NaN")
RatPoly.scaleCoeff(PolyCalc.RatTermVec, PolyCalc.RatNum):::ENTER
  (<Called from RatPoly.negate>)  ==>  (scalar.denom == 1)
  (<Called from RatPoly.negate>)  ==>  (scalar.numer == -1)
RatTermVec.remove(int):::ENTER
  (<Called from RatPoly.divAndRem>)  ==>  (index == 0)
```

Figure 6.3: Indications of implementation inefficiencies, as found in partial output of context-sensitive invariant detection on `RatPoly`, using method-level granularity.

```
class IntSet {
  //@ invariant elts != null
  private MyVector elts;

  //@ requires elt != null
  public void add(Integer elt);

  //@ requires elts.store.length > 0
  //@ ensures \result != null        /**/
  public Integer min() {
    return (Integer) elts.get(0);    /**/
  }

  ...
}
```

```
class MyVector {
  //@ invariant store != null
  private Object[] store;

  public void add(Object elt);

  //@ requires index < store.length
  public Object get(int index);

  ...
}
```

Figure 6.4: Generated specifications *without* explicitly-contextual invariants. Even though `IntSet` only stores non-null `Integer`s, `MyVector`'s specification does not reflect this property because it has multiple callers. Therefore, the statements marked with `/**/` fail to verify: the first because `MyVector.get` may return null, the second because `MyVector.get` is not constrained to return an `Integer`.

```
class IntSet {
  //@ invariant elts.user0 == true
  //@ invariant elts != null
  private MyVector elts;

  //@ requires elt != null
  public void add(Integer elt);

  //@ requires elts.store.length > 0
  //@ ensures \result != null
  public Integer min() {
    return (Integer) elts.get(0);
  }

  ...
}
```

```
class MyVector {
  //@ ghost public boolean user0;

  //@ invariant store != null
  /*@ invariant user0 ==>
       (((store elements) != null) &&
        ((store elements).class == Integer))
  */
  private Object[] store;

  /*@ requires user0 ==>
       ((elt != null) &&
        (elt.class == Integer))
  */
  public void add(Object elt);

  //@ requires index < store.length
  /*@ ensures user0 ==>
       ((\result != null) &&
        (\result.class == Integer))
  */
  public Object get(int index);

  ...
}
```

Figure 6.5: Generated specifications *with* explicitly-contextual invariants. Once explicit context information is added (properties involving `user0`), verification succeeds. Only one change is made to `IntSet`; most additions are to `MyVector`'s specification. (Some `\forall` and `\typeof` notation has been abbreviated for clarity and space.)

6.3.2 Specialized representation invariants

The specialization of Section 6.3.1 is only necessary if the Bs used in A maintain an additional representation invariant. Normally, that invariant would only appear as a property scattered throughout A (e.g., all Bs returned from A have property P). Furthermore, the property would often be impossible to verify with a modular checker, since the inductive proof that B's specialized representation invariant is maintained is beyond its scope. However, by pushing the invariant into B, we can enable its proof via the `@invariant` annotation, and allow its consequences to appear as verifiable postconditions of B's methods. This is similar (in its effects) to inlining the code of B's methods while checking A; in essence, it lets us use a context-insensitive modular checker for context-sensitive static analysis.

The technique is not limited just to types or nullness, as in the `MyVector` example. Any property within the scope of the tools can be enforced, such as integer ranges, sorted-ness, etc. Though we use the term *polymorphic specification*, the technique is applicable to more than just standard type information.

The only requirement of the context used to specialize B is consistency: all instances of B that are used with context-dependent specification enabled must always be checked with that specification enabled. Otherwise, soundness is lost. With this constraint, though, any granularity (defined in Section 6.2.1) is acceptable. Our intuition says that a class-level granularity will be most useful, as was the case with `IntSet`. However, an even wider granularity, such as package-level, may also be useful. We suspect that using a may-alias analysis would suggest the proper scope. A private representation field can often be shown to be referenced by only one instance of its defining class, in which case class-level granularity is appropriate. On the other hand, if data is shared within a package, an instance may be aliased by multiple classes in that package, which suggests that package-level granularity would be required.

6.3.3 Algorithm

We propose implementing the checking of explicitly contextual specifications in the following way.

1. Run Daikon with context sensitivity enabled at the class granularity. The generated specification for a class T may contain explicitly-contextual implications whose predicate is of the form `<Called from Class C>`.

2. Construct a set S_{all} of pairs whose elements are every observed substitution of T and C as given in the previous step.

3. Perform an analysis to form S_{ok}, a safe subset of S_{all}. (The safety requirement is that any time a refined specification of T is utilized, those instances of T are always checked with the refined specification.) As one choice, select all pairs where C has at least one field of type T and all fields of type T are not aliased outside of C.

4. For each pair in S_{ok}

   (a) Add a boolean specification-only field `userC` to T's specification.

   (b) Replace `<Called from Class C>` in T's specification with `userC`.

   (c) For each T-typed field `f` in C, add an object invariant `@invariant this.f.userC == true`.

These steps create context-independent specifications that may be verified in the same way as any other specification.

6.4 Discussion

We have presented practical techniques to implement both the generation and checking of automatically-generated specifications of polymorphic code, a situation that arises in most larger software projects.

Our generation extension provides a new way to induce Daikon to produce implications. Previous techniques depend on discovering mutually exclusive properties when inference is complete. Our technique of splitting based on call-site provides a necessarily exclusive property that can be used to form implications.

Our checking extension shows how to use a modular checker to check context-dependent properties. While the context-dependent properties were useful in studying `RatPoly` even without being checked, providing a way to validate the specification may enable more users to take advantage of it.

Chapter 7

Scalability

Daikon operates offline and in batch mode, by first reading all data from a trace file into memory, and then processing it. For large or long-running programs, trace files may be consume too much space or take too long to read from disk, or Daikon may have insufficient memory to hold all data. To analyze larger or longer-running programs, Daikon should be modified to operate *incrementally*, processing one sample at a time, without requiring that all data be available in memory, and operate *online*, running concurrently with the program under test. This chapter proposes a technique to improve the scalability of Daikon by taking advantage of properties of program point structure. By improving Daikon's use of processor and memory resources, we enable online and incremental operation, permitting analysis of larger and longer-running programs.

7.1 Staged inference

As described in Section 2.2 (page 7), the Daikon invariant detector infers invariants over a trace database captured from an instrumented version of the target program. Daikon operates in batch mode, by first reading all samples into memory, and then using multiple passes over the samples to infer invariants. The multiple passes permit optimizations because certain invariants are always true or false, or certain derived variables are undefined. By testing the strongest invariants in earlier passes, the weaker invariants or certain derived variables may not need to be processed at all. For example, if $x = 0$ always holds over an earlier pass, then $x \geq 0$ is necessarily true and need not be instantiated, tested, or reported on a later pass. Similarly, unless the invariant $i < \mathrm{theArray.length}$ holds, the derived variable $\mathrm{theArray}[i]$ may be non-sensical.

While operating in passes, Daikon also treats each program point independently. Therefore, data from one program point may be discarded before the other points' data is processed.

However, processing data in passes prevents Daikon from operating on traces where the amount of the data exceeds available memory. Currently, programs that run (uninstrumented) for longer than about two minutes create enough data to reach this limit. To analyze larger or longer-running programs, Daikon must operate incrementally, so that not all data is required to be available at once.

The naive solution would be to simply forgo the optimizations noted in the first paragraph of this section. However, such an approach is so computationally expensive as to be infeasible. Some optimization is required so that fewer candidate invariants are instantiated and tested. Section 7.3 presents a technique that provides such an optimization, but we first review important terminology.

7.2 Terminology

Section 2.2 described how Daikon operates, but here we more carefully define its terminology, to give a foundation for the rest of this chapter.

A *program point* is slightly more general than just a specific location in the program. Instead, it represents a specific scope (set of variables) and its associated semantics. For example, consider a program point associated with the pre-state of a method. Its scope is all fields of the class and any arguments to the method. Its semantics are that every time the method is called, a snapshot of all pre-state within scope is taken. For a program point associated with the object invariants of a class, its scope is all fields of the class, and its semantics are that every time the any public method is called, snapshots of all pre-state and post-state within scope is taken.

A *sample* is the snapshot of program state taken for a specific program point.

A *variable* is really an expression associated with a given scope (a program point). It may be a simple field reference (such as `this.x`), may involve array indexing or slicing (such as `this.myArray[x..y]`), or may involve other compound expressions.

A *derived variable* is a variable whose value is not provided in a sample, but is instead computed as a function of other variables after the fact. It is computed by Daikon as opposed to the front end. For example, array slices are derived from the full array given in the sample.

7.3 Variable ordering

To improve the performance and usability of Daikon, we propose that program points should no longer be processed inde-

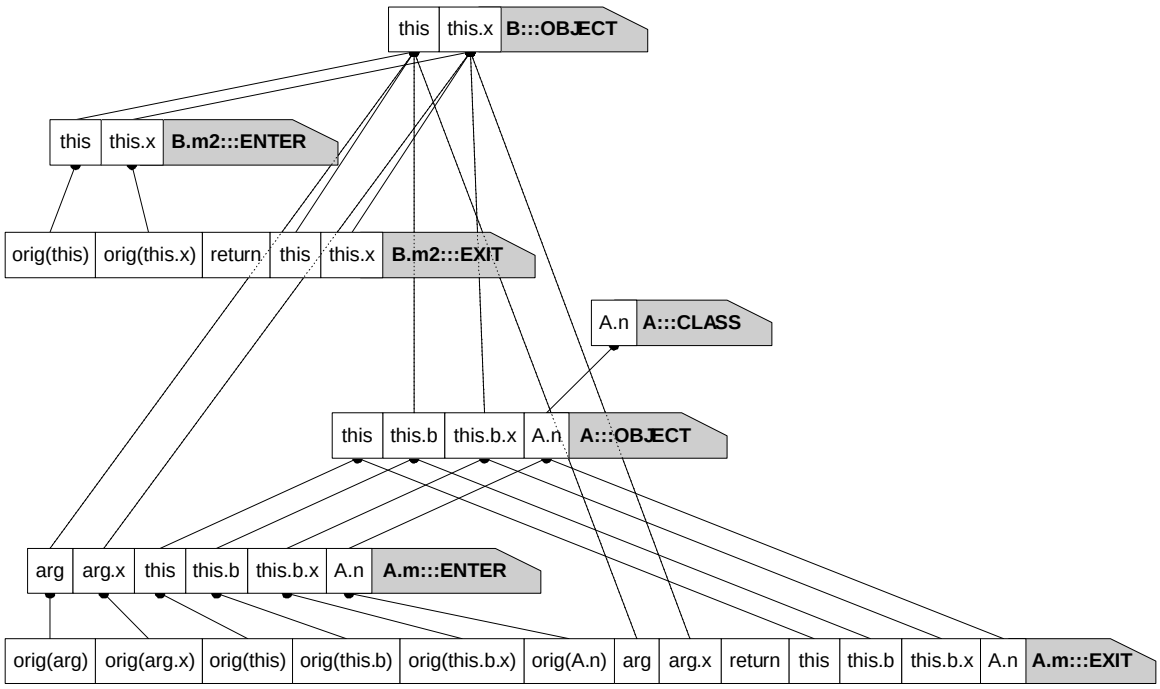


Figure 7.2: Flow relationship between variables for the code shown Shaded areas name the program point, while unshaded boxes represent variables at that program point. Lines show the partial ordering $\sqsubseteq_D$ described in Section 7.3, with a nub at the lesser end of the relation. (For instance, `arg` $\sqsubseteq_D$ `orig(arg)` in the lower left corner.) Relations implied by transitivity of the partial order are not explicitly drawn.

```
public class A {
  public static int n;
  private B b;
  public int m(B arg);
}
```

```
public class B {
  private int x;
  public int m2();
}
```

Figure 7.1: Example declarations for two simple Java classes.

pendently. Instead, we relate variables from all program points in a partial order.

The relationship that defines the partial order $\sqsubseteq_D$ is "sees as much data as". If variables X and Y satisfy X $\sqsubseteq_D$ Y, then all data seen at Y must also be seen at X — X sees as much data as Y. As a consequence, the invariants that hold over X are a subset of those that hold over Y, since any data that contradicts an invariant over Y must also contradict the same invariant over X.

Figure 7.2 shows the partial order formed by $\sqsubseteq_D$ for the example classes of Figure 7.1. Consider the relationship between `B:::OBJECT` and `B.m2:::ENTER`. First, recall that all data from method entries must also apply to the object invariants. (In other words, object invariants must always hold upon method entry.) Therefore, we have $\texttt{this}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this}_{\texttt{B.m2:::ENTER}}$ and $\texttt{this.x}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this.x}_{\texttt{B.m2:::ENTER}}$. The same holds true for method exits: $\texttt{this}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this}_{\texttt{B.m2:::EXIT}}$. Finally, note that $\texttt{this}_{\texttt{B.m2:::ENTER}} \sqsubseteq_D \texttt{orig(this)}_{\texttt{B:::EXIT}}$, since any pre-state data associated with a method exit must have been seen on entry.

For reasons similar to ones that relate `B`'s variables across program points, the relationships that contribute to the partial order are as follows.

Definition of `orig()`.

Variables on `ENTER` points are $\sqsubseteq_D$ the corresponding `orig()` variables at all `EXIT` and `EXCEPTION` program points.

Object invariants hold at method boundaries.

Instance variables from the `OBJECT` program point are $\sqsubseteq_D$ the corresponding instance variables on all `ENTER`, `EXIT`, and `EXCEPTION` program points.

Object invariants hold for all instances of a type.

Instance variables from the `T:::OBJECT` program point are $\sqsubseteq_D$ the corresponding instance variables on instrumented arguments and fields of type `T`. (For example, see $\texttt{arg}_{\texttt{A.m:::ENTER}}$ and $\texttt{this.b.}_{\texttt{A:::OBJECT}}$ in Figure 7.2.)

Subclassing preserves object invariants.

Instance variables from the `T:::OBJECT` program point are $\sqsubseteq_D$ the same instance variables on subclasses or non-static inner classes of `T`.

Overriding methods may only weaken the specification.

Argument(s) to a method `m` are $\sqsubseteq_D$ argument(s) of methods that override or implement `m`, by the behavioral subtyping principle.

7.4 Consequences of variable ordering

As shown in the previous section, the partial ordering of variables implies that when invariants hold true over variables at certain program points, those invariants also must hold true at lower (as drawn in Figure 7.2) program points. For instance, if we have $\texttt{this.x}_{\texttt{B:::OBJECT}} \geq 0$, then we also know that $\texttt{arg.x}_{\texttt{A.m:::ENTER}} \geq 0$.

Daikon's implementation could take advantage of this fact by only instantiating, testing, and reporting invariants at the most general place they could be stated. For instance, if an invariant always holds over an object's fields, it would only exist at the `OBJECT` program point (instead of each method's `ENTER` and `EXIT` points), and would only need to be tested once per sample.

The implementation could locate all invariants over a set of variables V at a program point P by forming the closure of V at P using the partial ordering, and taking the union (conjunction) of the invariants present at each point in the closure.

However, for this technique of minimal invariant instantiation to work, both the samples and the invariants must flow through the partial order in a specific way, as explained in the next two sections.

7.5 Invariant flow

The observations above lead to the following proposal for invariant instantiation.

1. At the start of inference, instantiate invariants only where one or more of the variables used to fill in the invariant template has no predecessor in the $\sqsubseteq_D$ partial order. That is, a set of n variables V should be used to fill an n-ary template only if $\forall x \exists v \in V : x \not\sqsubset_D v$.

2. When an invariant is falsified during inference, copy it "down" to the nearest program point(s) where every variable used by the invariant is less in the partial ordering (nearest meaning that there must be no intermediate choice). That is, a falsified invariant over a set of source variables A should be copied to destination sets B when all variables in B are at the same program point and when $\forall a \in A : (\exists b \in B : a \sqsubseteq_D b) \wedge (\neg \exists x : a \sqsubset_D x \sqsubset_D b)$.

3. Treat equality specially, using only one of the equal variables. For example, if $\texttt{x} = \texttt{y}$ then instantiate $\texttt{x} > \texttt{z}$, but not $\texttt{y} > \texttt{z}$. If $\texttt{x} = \texttt{y}$ is falsified, duplicate all invariants over x (replacing x with y), and also instantiate invariants relating x and y.

One positive consequence of this approach is that methods defined in interfaces will have invariants reported over their arguments, even though no samples can ever be taken on interfaces directly. For example, if every implementation of an interface's method is called with a non-null argument, Daikon will report this property as a requirement of the interface, instead of as a requirement of each implementation.

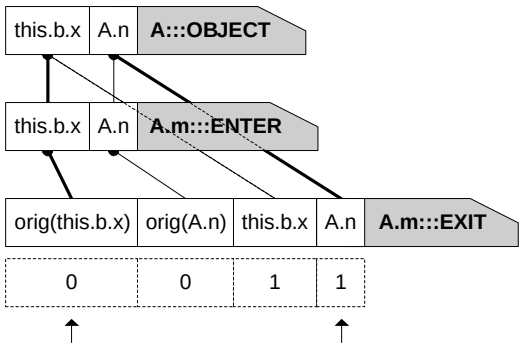


Figure 7.3: Example indicating the need for path information in sample and invariant flow, as described in Section 7.7. A portion of Figure 7.2 is reproduced, along with a potential sample (0,0,1,1). Given only that sample, the invariant $\texttt{this.b.x} = \texttt{A.n}$ at `A::OBJECT` should hold. However, if the sample flows as indicated by the bold links of the partial order, the invariant would be incorrectly falsified. Therefore, the path taken is important.

7.6 Sample flow

In the invariant flow algorithm, invariants flow down as they are falsified. This property suggests a corresponding flow algorithm to process samples.

1. Identify the exact program point where the sample was drawn from.

2. Form the closure of program points that have any variable filled in by following the relations upward in the partial order.

3. Feed the sample to each of these program points in a topological order. A sample is fed to a point after it has been fed to all points where a variable is greater. Therefore, any falsified invariants are always coped to lower program points before the sample is fed there to falsify them.

7.7 Paths

For both invariant and sample flow, the path taken through the partial order is important. For example, consider Figure 7.3. Given only this data, Daikon should report that $\texttt{this.b.x} = \texttt{A.n}$ at `A::OBJECT`. However, since we have $\texttt{this.b.x}_{\texttt{A:::OBJECT}} \sqsubseteq_D \texttt{orig(this.b.x)}_{\texttt{A.m:::EXIT}}$ and $\texttt{A.n}_{\texttt{A:::OBJECT}} \sqsubseteq_D \texttt{A.n}_{\texttt{A.m:::EXIT}}$, the values for `orig(this.b.x)` and `A.n` would falsify the invariant. The problem is that the two paths through the partial order are different — they traverse different program points.

To address this problem, we annotate each edge in the partial order with some nonce. A pair of variables `<A1,A2>` is together related to `<B1,B2>` by the partial order if the path from `A1` to `B1` follows the same nonces as the path from `A2` to `B2`. The nonces must be chosen so that sets of variables from two program points that are related due to the same item

from the list starting on page 36 must share the same nonce. In terms of Figure 7.2, parallel or nearly-parallel lines from one program point to another will have the same nonce.

7.8 Tree structure

An important property of the technique presented in this chapter is that an invariant only appears at the one place where it may be most generally stated. This implies that each variable has at most one parent, so the partial ordering forms a forest of variables.

However, at least one situation violates this constraint. With multiple inheritance (due to interfaces), a method's specification could be governed by multiple interfaces, so its arguments would have multiple parents in the partial order. Furthermore, depending on the implementation of conditional program points, a variable at a conditional program point could have multiple parents. For instance, a field at a conditioned method program point might have as parents both the unconditional version of itself from the unconditional method program point, and the conditioned version of itself from the object program point.

To solve this, we could reword "an invariant only appears at the *one place* where it may be most generally stated" to *minimal number of places*. The implementation would have to take into account the non-tree nature of the partial order when flowing samples and variables.

7.9 Conclusion

We have described a technique to improve the scalability of Daikon by organizing program point structure to embody knowledge of a program's structural semantics. This technique enables Daikon to use processor and memory resources more efficiently, because invariants that are known to be true need not be instantiated, tested, or reported. This technique helps enable online and incremental operation, permitting analysis of larger and longer-running programs.

Chapter 8

Related Work

This is the first research we are aware of that has dynamically generated and statically verified program specifications, used such information to investigate the amount of information about program semantics available in test runs, or evaluated user effectiveness in using dynamically detected specifications to verify programs.

The component analysis techniques, however, are well-known: much work has been done with a specific static or dynamic analysis (Section 8.1). The Houdini tool is notably similar to our research (Section 8.2). Finally, specifications generated from Daikon have been used for purposes beyond the applications explored in this work (Section 8.3).

8.1 Individual analyses

While ours is the first work to evaluate the combination of static and dynamic analyses, the two component techniques are well-known.

8.1.1 Dynamic analyses

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.

8.1.2 Static analyses

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].

8.1.3 Verification

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.

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.

8.2 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.

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. This is a limitation of the Daikon prototype, not of the technique of dynamic invariant detection. An appropriate re-engineering effort is currently underway, with its approach driven in part by insights gained in through this research.

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. With the assistance of Houdini, programmers may only need to insert about one annotation per 100 lines of code.

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 was not publicly available until shortly before publication, so we could not 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 explicitly in place of Houdini's inferred invariants. 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.

8.3 Applications

In this work, we evaluate the accuracy of Daikon's specification generation, and measure its effectiveness when used to assist a program verification task. The surprising accuracy of the results suggest that other uses of the generated specifications will have utility. Indeed, other researchers have usefully utilized specifications generated from Daikon.

Generated specifications enable proofs over languages other than Java. Our colleagues 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. In preliminary experiments [NWE02], 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.

Generated specifications also suggest program refactorings — transformations of a program to improve readability, structure, performance, abstraction, maintainability, or other characteristics [KEGN01]. It is advantageous to automatically identify places in the program that are candidates for specific refactorings. When particular invariants hold at a program point, a specific refactoring is applicable. Preliminary results are compelling: Kataoka's tool identified a set of refactorings that the author of the codebase had not previously identified or considered, and their recommended refactoring are justified in terms of run-time properties of the code that must hold for the refactoring to be correct.

Generated specifications are also useful for generating or improving test suites. Harder [Har02] presents the specification difference technique for generating, augmenting, and minimizing test suites. The technique selects test cases by comparing Daikon-generated specifications induced by various test suites. A test case is considered interesting if its addition or removal causes the specification to change. The technique is automatic, but assumes the existence of a test case generator that provides candidate test cases. The technique compares well with branch coverage: it performs about as well for fault detection, and slightly better for augmentation and minimization. However, the real benefit is in its combination with other techniques, such as branch coverage. Each technique is better at detecting certain types of faults.

Generated specifications are also effective for anomaly and bug detection. In [Dod02], Dodoo reports that techniques to discover predicates for implications in Daikon discover about 30% of the possible fault-revealing invariants supported by Daikon's grammar, and that about 30% of the actual reported invariants are fault-revealing. This suggests that programmers may only have to examine a few invariants before finding one that points to the location of a fault. In [ECGN01], Ernst et al. demonstrate that programmers performing a maintenance task on C code were able to use Daikon's output to both reveal a preexisting bug, and avoid introducing new ones. Other similar dynamic analyses are also effective for similar tasks: [RKS02] uses Daikon along with other tools, while [HL02] reimplements a subset of Daikon that operates online alongside the program under test.

Chapter 9

Future Work

Possibilities for future research fall into a few broad categories: improvements to the evaluation (Section 9.1), improvements to the techniques (Sections 9.2 and 9.3), and improvements to the implementation (Section 9.4).

9.1 Further evaluation

Additional evaluation of the ideas proposed in this work is one of the most valuable lines of future research.

One worthwhile experiment would be a study of what factors contributed to the success of the accuracy experiment in Chapter 4. Multiple factors could be varied to explore their relevance to the results, including the programs under test, the test suites used to generate invariants, and Daikon settings, such as thresholds for its statistical tests.

Similarly, insight could be gained into the accuracy of context-sensitivity (Section 6.2) by enabling the context-sensitive analysis and performing accuracy experiments similar to Chapter 4, but using programs that have more context-sensitive properties. Examining measurements of redundancy, precision, and recall while varying the kind of the context-sensitivity enabled could lend insight into the effects of sensitivity on machine verification.

A case study similar to the modification to `replace` described in [ECGN01] would be informative. In contrast to the controlled experiment of Chapter 5, a case study provides qualitative results regarding the overall utility of generated specifications in the software life-cycle, instead of quantitative measures of a one-time task. Furthermore, a case study could explore larger programs, which may be different or more interesting than smaller programs, but the results would only be anecdotal.

Finally, we should address performance of automatic tools such as Daikon. How do cost metrics (time and space requirements) correlate with characteristics of the program under test (textual size or language), characteristics of the test size (calls or coverage), or settings of Daikon (grammar of invariants, granularity of context, or statistical thresholds)? Measuring of these factors would help predict tool performance in situations not yet explored.

9.2 Specification generation

Even though Daikon was successfully used as a specification generator in this work, improvements to its analysis could produce even better results.

Properties that are difficult to obtain from a dynamic analysis may be apparent from an examination of the source code. For instance, properties enforced by language semantics are not well-utilized by Daikon. Properties such as inheritance, overriding, visibility, and immutable fields could both permit faster inference (by eliminating testing of necessarily-true invariants) and more useful output (by avoiding repeating information obvious to programmers).

Alternatively, Daikon could focus on code or properties that stymie a static analysis. Properties that require detailed knowledge of the heap or inter-procedural reasoning are often beyond the capabilities of static tools, but may be within Daikon's reach. For instance, statically detecting the fact that a `Vector`-typed argument method never contains nulls might require analysis of all source code in the program, but a dynamic analysis could simply observe all calls to the method and report whether any nulls were observed.

9.3 Context sensitivity

The extensions of Chapter 6 provide many opportunities for further exploration.

One way to improve the system would be to integrate the invariant information into a profile-viewer that helps the user to visualize the call graph. The viewer could display invariants from certain control flow edges on the edges themselves, while context-insensitive properties could be associated with the nodes of the graph. This may be a convenient way to browse the output. Furthermore, if standard profiler output (such as call frequencies or elapsed times) is also shown, programmers can better relate frequent or expensive operations with the conditions under which they occur.

An important next step in the checking of context-sensitive specifications is to automatically generalize across multiple callers and discover the parameter(s) of the underlying polymorphic specification. For instance, in the example of Section 6.3, we might want to specify `MyVector` in terms of two parameters: what element type it holds, and whether nulls

may be stored. Comparing the structure of multiple context-dependent specifications may provide a way to achieve this result. Presenting just the generalized specification to users may be more understandable, or may ease its checking.

The partitioning of data suggested by context information could be combined with other partitioning techniques [Dod02] for use by machine learning (statistical) methods to form more complicated predicates for implications. Context-sensitivity provides an important first step towards producing useful and relevant predicates for implications, but its combination with machine learning may be even more useful.

Finally, future work could consider different degrees or kinds of context sensitivity. We have proposed examining callers at the line, method, class, or package granularity, but other useful groupings of callers may be useful. Furthermore, the sensitivity could extend to more than just the immediate caller. Perhaps interesting distinctions would be created by using the most recent n methods or classes on the stack, or using the most recent call not from the callee's class itself.

9.4 Implementation

Finally, Daikon would benefit from improved scalability. It requires that trace data to be written to disk, which prohibits analysis of large programs, and that all data is available at once, prohibiting online operation alongside the program. Furthermore, performance with context-sensitivity also degrades approximately linearly with the average number of callers per method.

Work is underway to create an implementation of Daikon that runs online, and that is more efficient at considering predicates for implications. (Chapter 7 presented a part of this design.) This would remove the need to write disk-based trace files, and permit the evaluation of larger programs.

Chapter 10

Conclusion

This thesis has demonstrated that program specifications may be accurately recovered from program source code by a combination of dynamic and static analyses, and that the resulting specifications are useful to programmers.

We retrieve a specification in two stages: the first is a *generation* step, where a specification is unsoundly proposed; the second is a *checking* step, where the proposal is soundly evaluated. Our approach is advantageous because the generation step can take advantage of the efficiency of an unsound analysis, while the checking step is made tractable by the postulated specification. We expect our techniques to improve programmer productivity when applied to verification, testing, optimization, and maintenance tasks.

We have performed two major experiments to evaluate our approach. An assessment of the accuracy of the dynamic component of specification generation showed that generated specifications scored over 90% on precision and recall, indicating that the Daikon invariant detector is effective at generating consistent, sufficient specifications. An assessment of the usefulness of the generated specifications to users showed that imprecision from the dynamic analysis is not a hindrance when its results are evaluated with the help of a static checker.

We have also proposed techniques to improve the sensitivity of our analyses when applied to polymorphic code. We suggest how to account for context-sensitive properties in both the generation and checking steps, and show how such information can assist program understanding, validate test suites, and form a verifiable specification.

Given that our techniques have been successful within our domain of investigation, we reflect on broader lessons that can be gleaned from our results.

Incomplete answers are better than nothing. Incomplete answers are usable, as long as their contribution outweighs the effort involved in their use. If our system used a different checker, such as one oriented more to theorem-proving, reports of errors might be less localized, thus obscuring their root cause. By using a modular checker, users may have found it (relatively) easier to be successful when only part of the answer was provided.

Imprecise answers are better than nothing. Even inaccurate output can be used effectively in practice, especially when tools assist in separating the good from bad. Many researchers have traditionally presupposed that complete soundness is required when dealing with specifications, and any unsound technique would be disastrous. In fact, unsound program analysis techniques are justifiably useful for program development.

Interaction is advantageous. Programmer interaction is acceptable and useful. We are not writing a compiler, but tools to be used actively by programmers. Attempting to create a system that solved every problem noted in the accuracy experiment (Chapter 4) on its own would be doomed to failure. Allowing programmer involvement enables success.

Intent matters. One factor in our success is that both tools used in our system were designed for use by working programmers using a practical language. This increases the likelihood that the tools' vocabulary and semantics are well-matched, and that programmers can take advantage of their capabilities. We suspect that integration of tools with similar characteristics will also succeed, but static and dynamic tools not meant for direct use by programmers may not.

Integration is key. Users are able to effectively use a combination of sound and unsound tools. Each tool by itself has serious weaknesses, but the two together address each other's weaknesses and enhance each other's strengths.

Acknowledgments

Portions of this thesis were previously published at the First Workshop on Runtime Verification [NE01], at ISSTA 2002 [NE02a], and at FSE 2002 [NE02b]. The first two works draw mainly from Chapter 4, while the last draws mainly from Chapter 5.

I have been extremely lucky to have Michael Ernst as my advisor. Michael is committed to helping his students succeed, and I have been a happy beneficiary of his invaluable guidance, both technical and otherwise. He is always available and works to involve his students in all aspects of research, from brainstorming to writing, and implementation to presentation. From his example, I have come to appreciate and enjoy research. I truly cannot imagine a better graduate advisor.

I also owe thanks to my colleagues in the Program Analysis Group — particularly Nii Dodoo, Alan Donovan, Lee Lin, Ben Morse, Melissa Hao, Mike Harder, and Toh Ne Win — for their contributions and suggestions. Mike Harder in particular has provided excellent critical review of my ideas, competent assistance in the engineering of our tools, and a good dose of common sense.

Alan Donovan was a collaborator for the work in Section 6.2.

I am indebted to Chandra Boyapati, Felix Klock, Stephen Garland, Tony Hoare, Rachel Pottinger, Gregg Rothermel, Steven Wolfman, and anonymous referees for their helpful suggestions on previous papers describing this research. Their feedback has significantly improved my investigation and presentation.

As my teachers and colleagues in teaching, Michael Ernst, Daniel Jackson and John Guttag have provided inspiration and insight into the joys of both studying and teaching software engineering. Teaching has enabled me to better understand the fundamental ideas, and taught me how to present my ideas clearly.

I thank the 6.170 staff members who designed, wrote, and documented some of the programs evaluated in Chapter 4, and the programmers who volunteered for the study in Chapter 5 for their time and comments.

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

My musical activities have provided the perfect counterbalance to my technical life. I thank my directors Fred Harris, Larry Isaacson, Tom Reynolds, Rob Rucinski, and the late John Corley for creating a wonderful musical environment for me to enjoy, and my instructors Edward Cohen, Tele Lesbines, George Ruckert, and Pamela Wood for opening my eyes and ears to the world of music.

Finally, I thank my parents and grandparents for their unwavering support and encouragement. Even though they joke about not understanding anything I've written, they have always encouraged me to do what I love, and I'm glad that I have.

Appendix A

User study information packet

This appendix contains the information packet that was given to participants in the user study of Chapter 5 (page 20). The formatting has been slightly altered, but the content remains the same, except for the author's contact information on page 49, which has been removed for publication. Also, contrary to what is stated on page 46, we did not actually provide printouts of the noted documents to study participants.

Finally, we acknowledge that the source code, diagrams, and explanatory text on pages 51–56 are originally from [Wei99]. In particular: the text on page 52 is reproduced from pages 269–272 of [Wei99]; the figure on page 52 is reproduced from page 272 of [Wei99]; the text on page 54 is reproduced from page 89 of [Wei99]; the figure on page 54 is reproduced from pages 89–90 of [Wei99]; and the text on page 56 is reproduced from page 78 of [Wei99].

INTRODUCTION

Thank you for assisting us with this experiment. This document contains background information, instructions, and reference information.

MOTIVATION

We are interested in the use of automated tools to check that a program does not crash with unexpected runtime errors. By evaluating your experience verifying two sample programs, we will be able to quantitatively judge our approach, and your individual comments will help us better understand its merits and deficiencies. Finally, you will have an opportunity to learn about exciting program analysis tools.

EXPECTATIONS

It is important for you to understand our expectations before you start. First, and most importantly, you are under no pressure to complete this experiment, even after you have started — **you may terminate the experiment at any time**. Also, no information about your personal performance will ever be publicly revealed.

We do **not** expect that everybody will complete all the tasks within the given time bounds — the tasks are (intentionally) of widely varying difficulties.

Finally, we have no expectations about your own performance. We are not evaluating your abilities — rather, we are evaluating tools for programmers. Problems you encounter are likely to indicate shortcomings in the tools, and your experiences will help us to evaluate and improve them.

LOGISTICS

- First, log in (via ssh) to the machine `geyer.lcs.mit.edu`, using the username and password you were given. This (temporary) account is exclusively your own — you are free to edit or upload files to your liking (use scp to transfer files). You should check that you are able to run Emacs (or your editor of choice). The files and directories mentioned in this document are located relative to your home directory.
- Next, spend no more than 20 minutes reading this documentation and studying the example below. You should read until you reach the horizontal line before the EXPERIMENT section.
- You will then have two programming tasks to complete, each of which will be limited to an hour at most. (Some tasks may take substantially less time; others may not be complete when the hour runs out.)
- Finally, we will conduct a brief exit interview to review your experiences.

More details are presented the the sections below.

In total, you will spend less than 2.5 hours. Please allow for an uninterrupted block of time to work on this project. Feel free to take short breaks, but please do not read email, etc. while you are working.

THE EXTENDED STATIC CHECKER FOR JAVA

ESC/Java is a tool that statically checks certain program properties. Users express the properties via source code annotations, similar to assertions. ESC reads the source code (and annotations) and warns about annotations which might not be universally true. ESC also warns about potential runtime exceptions such as null dereferences and array bounds errors.

The annotations are called “pragmas” by ESC. In general, ESC supports annotations (pragmas) anywhere in the source code. However, in this experiment, you will only use pragmas that specify properties of an abstract data type. Specifically, you will write a representation invariant (object invariant) and method specifications (preconditions, postconditions, and modification targets).

ESC is a modular checker: it reasons about code by examining one method at a time. Therefore, ESC both checks and depends on annotations. For instance, if an annotation describing the return value of an observer method is missing, ESC may not be able to prove a result about a method which calls that observer.

The next section gives an example of an annotated program. While reading it, you may wish to refer to the user’s manual and quick reference for ESC/Java. We have given you a printout of these, but you may also read them online.

User’s Manual:
`http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-2000-002.html`

Quick Reference:
`http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-2000-004.html`

FIXED-SIZE SET

`FixedSizeSet` is an boolean-array-based implementation of a set of the integers 0-7. You can find the source code in the `~/example/` directory. The `FixedSizeSet` class contains the implementation of the set, while the `FixedSizeSetCheck` class contains a few testing routines which perform operations on the set. We have annotated `FixedSizeSet` in the same way we will ask you to annotate two other data structures — this is an example of the "finished product" we will ask you to produce.

1. First, confirm that the program passes through ESC/Java without any errors. Run ESC on the sources:

   ```
   [study@geyer ~]% cd example
   [study@geyer example]% escjava FixedSizeSet.java FixedSizeSetCheck.java
   ```

   After a few seconds, ESC finishes with output that includes timing information and the word "passed" for each of the methods in `FixedSizeSet`. This indicates a successful verification. You may also run ESC/Java on a single file, if you only want to verify one class.

   (If ESC/Java is crashing with a message like "unexpected exit from Simplify", it is likely that you have nonsensical or contradictory invariants. Try removing annotations until the problem disappears.)

2. Now, remove the first annotation and see what happens: change the line

   ```
   /*@ invariant bits != null */
   ```

   by removing the `@`. This changes the line from an annotation pragma to a regular comment, which ESC ignores.

3. Run the checker again:

   ```
   [study@geyer example]% escjava FixedSizeSet.java
   ...
   FixedSizeSet.java:34: Warning: Possible null dereference (Null)
       bits[n] = true;
   ...
   ```

   ESC reports five warnings. The first one (reproduced above) states that the expression `bits[n]` in the `add` method may cause a `NullPointerException`. ESC needs to know that the array is never null to prove correct operation, and a programmer examining the source could easily determine this is true. However, since ESC checks each method in isolation, programmers must write the non-nullness condition into the representation invariant.

4. Put back the first annotation (by adding the `@` again) to restore the example to its original state.

5. Look at the annotations on the contains method and notice the use of the `\result` notation. In ESC, `\result` is a variable which represents the result of the method. Accordingly, `\result` may only be used in `@ensures` pragmas. Additionally, notice the use of == in-between two boolean values. This is a way of writing an "if and only if" (a bi-implication), where each side implies the other.

6. Look at the annotations on the union method and notice the use of the `\old(...)` notation.

   ```
   /*@ ensures bits[0] == (\old(bits[0]) || other.bits[0]) */
   ```

   The use of `\old` tells ESC to interpret the expression in the pre-state (i.e. on entry to the method). For instance, the above pragma for union states that after the call, the 0^{th} bit be set if either the 0^{th} bit was set in the pre-state, or if other had the 0^{th} bit set. Variables described in `\old` must also be listed in the `@modifies` clause, or ESC will issue a caution.

7. To see how an omission in `FixedSizeSet` can affect calling code, disable this annotation from the add method of `FixedSizeSet`:

   ```
   /*@ ensures bits[n] == true */
   ```

   Then, run escjava on `FixedSizeSetCheck.java`, which produces the following warning:

   ```
   FixedSizeSetCheck: checkAdd() ...
   ------------------------------------------------------------------------
   FixedSizeSetCheck.java:33: Warning: Possible unexpected exception (Exception)
   ```

   With the annotation missing, ESC/Java cannot verify that the set contains '3' just after it has been inserted, so the `Check` code fails. Replace the disabled annotation.

8. Now, examine the similar method, whose signature is shown here:

   ```
   public boolean similar(FixedSizeSet other) throws RuntimeException
   /*@ ensures other != null */
   /*@ exsures (RuntimeException) (other == null) */
   ```

Note the new clause: `@exsures`. Exsures is used to talk about postconditions for exceptional exits. Specifically, **if** the procedure exits with an exception, **then** the expression in the exsures clause must be true. Similarly, `@ensures` are expressions which must hold on non-exceptional exits.

The `@exsures` semantics might seem counterintuitive, since we often would say "if (expression) then Exception", whereas ESC uses "if Exception then (expression)". However, the combination of `@exsures` and `@ensures` can have the same effect. If the expressions in `@ensures` and `@exsures` are exhaustive (they cover all possibilities), then a caller can determine what will happen. For instance, in the example above, other can be either null or non-null, so either the `@ensures` expression must hold, or the `@exsures` expression must hold - they are exhaustive. Therefore, the caller is able to know exactly when an exception will occur.

As always, you may talk about either pre-state or post-state values in both `@ensures` and `@exsures` annotations (#6 above). Be sure to use `\old` if a variable is modified and you want pre-state; it is an easy point to miss.

9. Consider the second requires clause of the fillDigits method.

   ```
   /*@ requires \typeof(digits) == \type(Object[]) */
   ```

   This invariant states that the runtime type of the array is `Object[]` (instead of some subclass); this allows ESC to prove that an `ArrayStoreException` won't happen during assignment. (Recall that the run-time and compile-time of an array may differ.) Section 3.2.4 of the ESC manual describes this in more detail.

10. The pragmas involving `@spec_public` and the `owner` field are baseline annotations needed by ESC. You do not have to understand their specifics, as they will always be provided for you in this experiment.

11. To ensure you understand `FixedSizeSet`'s annotations, spend a few minutes reading over the other annotations and/or experimenting with adding or deleting annotations. Consult the ESC/Java quick reference or manual if you have questions about any of the annotations.

PROGRAMMING TASK

The study consists of two experiments, each one hour long. For each experiment, the programming task is as follows.

Two classes will be presented — an abstract data type (ADT) and a class which calls it. You will create and/or edit annotations in the source code of the ADT. Your goal is to enable ESC/Java to verify that neither the ADT nor the calling code may ever terminate with a runtime exception. That is, when ESC/Java produces no warnings or errors **on both the ADT and the calling code**, your task is complete.

TOOLS

For each of the two experiments, you may encounter one of three scenarios.

In the first scenario, you will receive un-annotated source code for the program (except for baseline annotations mentioned in #8 above), and will use only ESC/Java.

In the second scenario, you will receive un-annotated source code for the program (except for baseline annotations mentioned in #8 above), but will use the Houdini tool as part of ESC/Java. Houdini is a behind-the-scenes annotation assistant. When you run ESC/Java, Houdini steps in and guesses likely annotations, which are fed into ESC/Java along with your source code. If a guessed annotation fails to verify, it is simply ignored and has no effect on your checking. The guessed invariants are not shown to the user. By guessing likely invariants and always supplying them for you behind the scenes, Houdini allows you to write fewer annotations in the source code itself. Houdini guesses annotations of this form:

```
integers   :  variable cmp [-1, 0, 1, array.length, variable]
              (cmp is = ≠ > ≥ < ≤)
references :  variable != null
array      :  first variable elements of array are non-null
```

In the third scenario, you will receive source code with many annotations already present (in addition to the annotations mentioned in note #8 above). These annotations were produced by the Daikon tool, which infers program invariants from actual program executions. The properties were true for a small number of executions of the program, and may be true in general. However, since the executions did not include all possible inputs, **some of the provided annotations may not be universally true**. You may use these annotations as a starting point in your programming task. You are permitted to edit or delete them, and to add new annotations.

The third scenario is easy to distinguish, since you will already see annotations when you begin. For the first and second scenarios, you can tell if Houdini is enabled by running ESC/Java on the source and watching for "Houdini is generating likely invariants" as the first line of output.

GUIDELINES

- Ensure that both the ADT and calling code pass escjava.
- Ensure that you don't spend more than 60 minutes on each half.

- Do not modify the calling code or the ADT's implementation at all — you should only add or edit **annotations** in the **ADT**.
- Do not use any unsound pragmas provided by ESC/Java (such as `@nowarn`, `@assume`, or `@axiom`).
- As you work to complete this task, you may have further questions. If you have practical questions, such as how to invoke ESC/Java, or how to state a certain property, feel free to ask us. However, in order not to invalidate the results of the experiment, we will not answer questions about the task itself, such as why a certain annotation fails to verify.
- You may contact me (Jeremy) by visiting NE43-525, calling *[snip]* (w), *[snip]* (h), emailing *[snip]*, or zephyring *[snip]*.

EXPERIMENT

When you are ready to begin, please start by answering these questions:

```
What login name were you given (studyXX)? _____________

How many years of post-high-school education have you had? _____________

How many years have you been programming? _____________

How many years have you been programming in Java? _____________

When you are programming, do you primarily work with tools for:
Circle one: Windows, Unix, or Both?

Do you write assert statements (in code) when you are writing code?
Circle one: Never, Rarely, Sometimes, Often, Usually, Always

Do you write assertions in comments when you are writing code?
Circle one: Never, Rarely, Sometimes, Often, Usually, Always

Have you ever used ESC/Java before? _____________
```

Now, please start with the program in the directory `~/experiment1/` and perform the task described in the PROGRAMMING TASK section above. See the README file found alongside the source, and photocopies from a textbook that we provide you, for additional information about the program's implementation. Spend at most one hour on this program. As you begin, and when you are done, make sure you record the amount of time you spent in the space below.

```
Experiment 1 start time: ___________________

Experiment 1 stop time: ___________________

Experiment 1 elapsed time: ___________________
```

Do not edit the first program again — leave it unchanged as you continue on to the second.

Next, take the program in the directory `~/experiment2/` and perform the same task. Again, note your start, stop, and elapsed times in the space below. Spend at most one hour on this program.

```
Experiment 2 start time: ___________________

Experiment 2 stop time: ___________________

Experiment 2 elapsed time: ___________________
```

When you are done, please contact us for a brief exit interview. You may contact us via any of the methods listed in the section above. If you are at LCS, simply stopping by room 525 may be easiest. If you are off-site, send me an email and I will call you.

We prefer an oral interview, and you may also find it more convenient, but if you would rather write out your answers, you may answer the written interview questions below. In either case, please do one of these immediately after finishing, while the experience is fresh in your mind.

Also, feel free to make comments below about your experience with ESC, Houdini, or any pre-annotated source code you were given. We are interested in hearing your experiences, comments, and suggestions. (We will also examine your completed work in the `~/experiment1` and `~/experiment2` directories.)

EXIT INTERVIEW

(Again, most participants will do an oral interview instead of answering these questions, but you may write your answers if you prefer. Feel free to use additional sheets if necessary.)

Did you write all your annotations first, then check them, or incrementally add annotations and check? How much time did you spend after the first ESC/Java run? (Essentially, describe your mode of operation while performing this task). Was the approach the same for both halves; how did it differ?

Did you use cut-and-paste or write annotations from scratch? Furthermore, did you refer back to previous work during later work, (e.g. review the example during experiment 2 to find out about exsures)?
What did you find especially hard (or especially easy)?

How much of your effort was struggle learning ESC idiosyncrasies; how much was thought-provoking exploration of the program? (Stated another way: describe how much time was relatively spent on figuring out syntax or what could be stated, compared to trying to reason about the program's behavior or causes of warnings.)

Do you have any suggestions for improving the way the tool(s) work? (What did you expect or want to see which you didn't?) Did you use or appreciate the execution path information (branches taken) reported by ESC/Java?

Did the provided annotations help with the process? (If you were provided any annotations.)

Describe your qualitative experience with provided annotations?

If you were provided annotations for experiment 1, were they helpful to use for experiment 2?

Are there situations in your own work where you would consider using ESC/Java (or a similar tool if you don't write Java code)? If so, under what circumstances? If you were provided help from Daikon or Houdini, (how) would that affect your decision?

Before starting the experiment, were you already familiar with the notions of representation invariants, preconditions, and postconditions?

Please use this space for any further comments.

```
/**
 * Disjoint set class.
 * Does not use union heuristics or path compression.
 * Elements in the set are numbered starting at 0.
 * @author Mark Allen Weiss
 */
public class DisjSets
{
  /*@ spec_public */ private int [ ] s;
  /*@ invariant s.owner == this */

  /**
   * Construct the disjoint sets object.
   * @param numElements the initial number of disjoint sets.
   */
  public DisjSets( int numElements )
  {
    s = new int [ numElements ];
    /*@ set s.owner = this */
    for( int i = 0; i < s.length; i++ )
      s[ i ] = -1;
  }

  /**
   * Union two disjoint sets.  For simplicity, we assume root1 and
   * root2 are distinct and represent set names.
   *
   * @param root1 the root of set 1.
   * @param root2 the root of set 2.
   **/
  public void unionDisjoint( int root1, int root2 )
  {
    s[ root2 ] = root1;
  }

  /**
   * Union any two sets.
   * @param set1 element in set 1.
   * @param set2 element in set 2.
   **/
  public void unionCareful( int set1, int set2 )
  {
    int root1 = find(set1);
    int root2 = find(set2);
    if (root1 != root2)
      unionDisjoint(root1, root2);
  }

  /**
   * Perform a find.
   * Error checks omitted again for simplicity.
   * @param x the element being searched for.
   * @return the set containing x.
   **/
  public int find( int x )
  {
    if( s[ x ] < 0 )
      return x;
    else
      return find( s[ x ] );
  }

}
```

In this chapter, we describe an efficient data structure to solve the equivalence problem. The data structure is simple to implement. Each routine requires only a few lines of code, and a simple array is used. The implementation is also extremely fast, requiring constant average time per operation.

Recall that the problem does not require that a find operation return any specific name; just that finds on two elements return the same answer if and only if they are in the same set. One idea might be to use a tree to represent each set, since each element in a tree has the same root. Thus, the root of the tree can be used to name the set. We will represent each set by a tree. (Recall that a collection of trees is known as a *forest*.) Initially, each set contains one element. The trees we will use are not necessarily binary trees, but their representation is easy, because the only information we will need is a parent link. The name of a set is given by the node of the root. Since only the name of the parent is required, we can assume that this tree is stored implicitly in an array: each entry `s[i]` in the array represents the parent of element i. If i is a root, then `s[i] = -1`. In the forest in Figure 8.1, `s[i] = -1` for $0 \leq i < 8$. As with binary heaps, we will draw the trees explicitly, with the understanding that an array is being used. We will draw the root's parent link vertically for convenience.

To perform a union of two sets, we merge the two trees by making the parent link of on tree's root link to the root node of the other tree. It should be clear that this operation takes constant time. Figures 8.2, 8.3, and 8.4 represent the forest after each of `union(4,5)`, `union(6,7)`, `union(4,6)`, where we have adopted the convention that the new root after the `union(x,y)` is `x`. The implicit representation of the last forest is shown in figure 8.5.

A `find(x)` on element `x` is performed by returning the root of the tree containing `x`.

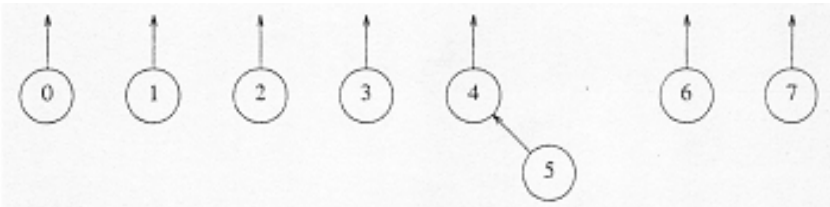


Figure 8.2 After union(4,5)

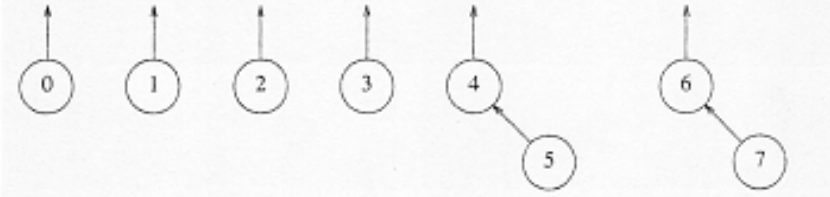


Figure 8.3 After union(6,7)

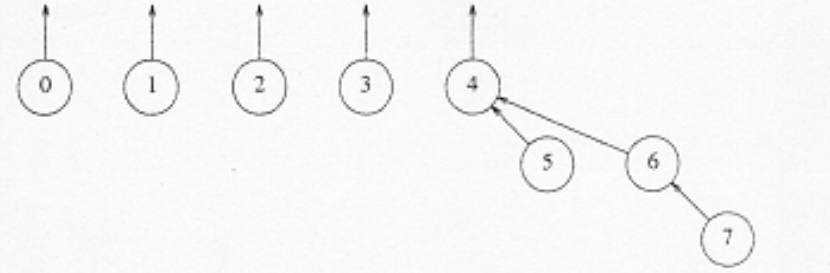


Figure 8.4 After union(4,6)

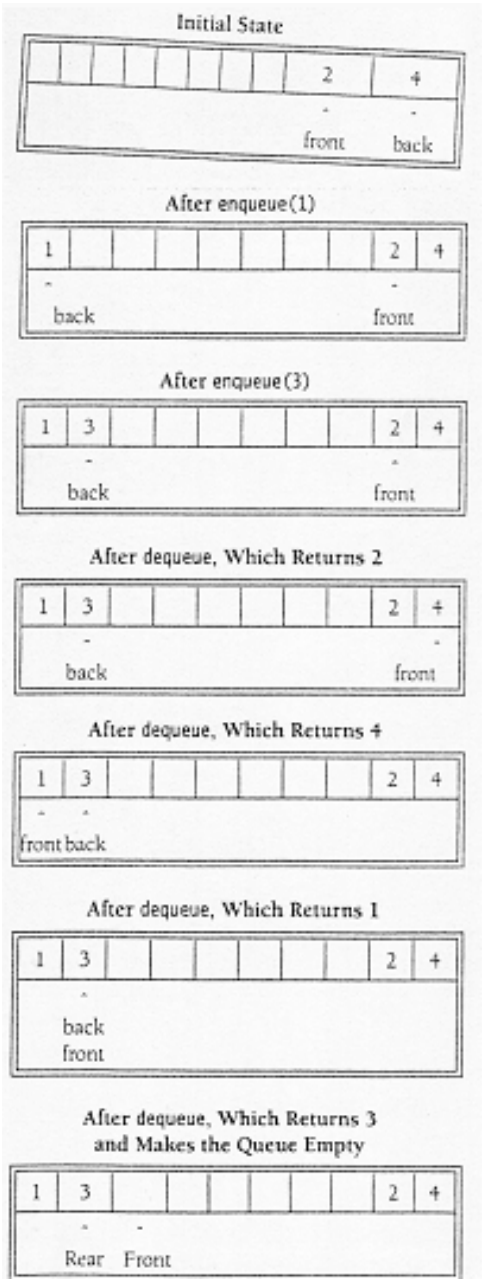
−1	−1	−1	−1	−1	4	4	6
0	1	2	3	4	5	6	7

```
/**
 * Array-based implementation of the queue.
 * @author Mark Allen Weiss
 **/
public class QueueAr
{
  /*@ spec_public */ private Object [ ] theArray;
  /*@ invariant theArray.owner == this */
  /*@ spec_public */ private int        currentSize;
  /*@ spec_public */ private int        front;
  /*@ spec_public */ private int        back;

  /**
   * Construct the queue.
   **/
  public QueueAr( int capacity )
  {
    theArray = new Object[ capacity ];
    currentSize = 0;
    front = 0;
    back = theArray.length - 1;
    /*@ set theArray.owner = this */
  }

  /**
   * Test if the queue is logically empty.
   * @return true if empty, false otherwise.
   **/
  public boolean isEmpty( )
  {
    return currentSize == 0;
  }

  /**
   * Test if the queue is logically full.
   * @return true if full, false otherwise.
   **/
  public boolean isFull( )
  {
    return currentSize == theArray.length;
  }

  /**
   * Make the queue logically empty.
   **/
  public void makeEmpty( )
  {
    currentSize = 0;
    front = 0;
    back = theArray.length - 1;
    java.util.Arrays.fill(theArray, 0, theArray.length, null);
  }

  /**
   * Get the least recently inserted item in the queue.
   * Does not alter the queue.
   * @return the least recently inserted item in the queue, or null, if empty.
   **/
  public Object getFront( )
  {
    if( isEmpty( ) )
      return null;
    return theArray[ front ];
  }

  /**
   * Return and remove the least recently inserted item from the queue.
   * @return the least recently inserted item in the queue, or null, if empty.
   **/
  public Object dequeue( )
  {
    if( isEmpty( ) )
      return null;
    currentSize--;
    Object frontItem = theArray[ front ];
    theArray[ front ] = null;
    if ( ++front == theArray.length )
      front = 0;
    return frontItem;
  }

  /**
   * Insert a new item into the queue.
   * @param x the item to insert.
   * @exception Overflow if queue is full.
   **/
  public void enqueue( Object x ) throws RuntimeException
  {
    if( isFull( ) )
      throw new RuntimeException( "Overflow" );
    if ( ++back == theArray.length )
      back = 0;
    theArray[ back ] = x;
    currentSize++;
  }

}
```

The operations should be clear. To enqueue an element `x`, we increment `currentSize` and `back`, then set `theArray[back] = x`. To dequeue an element, we set the return value to `theArray[front]`, decrement `currentSize`, and then increment `front`. Other strategies are possible (this is discussed later). We will comment on checking for errors presently.

There is one potential problem with this implementation. After 10 `enqueues`, the queue appears to be full, since `back` is now at the last array index, and the next `enqueue` would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The simple solution is that whenever `front` or `back` gets to the end of the array, it is wrapped around to the beginning. The following figures show the queue during some operations. This is known as a *circular array* implementation.



```
/**
 * Array-based implementation of the stack.
 * @author Mark Allen Weiss
 **/
public class StackAr
{

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

  /**
   * Construct the stack.
   * @param capacity the capacity.
   **/
  public StackAr( int capacity )
  {
    theArray = new Object[ capacity ];
    /*@ set theArray.owner = this */
    topOfStack = -1;
  }

  /**
   * Test if the stack is logically empty.
   * @return true if empty, false otherwise.
   **/
  public boolean isEmpty( )
  {
    return topOfStack == -1;
  }

  /**
   * Test if the stack is logically full.
   * @return true if full, false otherwise.
   **/
  public boolean isFull( )
  {
    return topOfStack == theArray.length - 1;
  }

  /**
   * Make the stack logically empty.
   **/
  public void makeEmpty( )
  {
    java.util.Arrays.fill(theArray, 0, topOfStack + 1, null);
    topOfStack = -1;
  }

  /**
   * Get the most recently inserted item in the stack.
   * Does not alter the stack.
   * @return the most recently inserted item in the stack, or null, if empty.
   **/
  public Object top( )
  {
    if( isEmpty( ) )
      return null;
    return theArray[ topOfStack ];
  }

  /**
   * Remove the most recently inserted item from the stack.
   * @exception RuntimeException if stack is already empty.
   **/
  public void pop( ) throws RuntimeException
  {
    if( isEmpty( ) )
      throw new RuntimeException( "Underflow" );
    theArray[ topOfStack-- ] = null;
  }

  /**
   * Insert a new item into the stack, if not already full.
   * @param x the item to insert.
   * @exception RuntimeException if stack is already full.
   **/
  public void push( Object x ) throws RuntimeException
  {
    if( isFull( ) )
      throw new RuntimeException( "Overflow" );
    theArray[ ++topOfStack ] = x;
  }

  /**
   * Return and remove most recently inserted item
   * from the stack.
   * @return most recently inserted item, or null, if
   * stack is empty.
   **/
  public Object topAndPop( )
  {
    if( isEmpty( ) )
      return null;
    Object topItem = top( );
    theArray[ topOfStack-- ] = null;
    return topItem;
  }

}
```

Array Implementation of Stacks

An alternative implementation avoids links and is probably the more popular solution. The only potential hazard with this strategy is that we need to declare an array size ahead of time. Generally this is not a problem, because in typical applications, even if there are quite a few stack operations, the actual number of elements in the stack at any time never gets too large. It is usually easy to declare the array to be large enough without wasting too much space. If this is not possible, we can either use the linked list implementation or use a technique, suggested in exercise 3.29, that expands the capacity dynamically.

If we use an array implementation, the implementation is trivial. Associated with each stack is `theArray` and `topOfStack`, which is -1 for an empty stack (this is how an empty stack is initialized). To push some element `x` onto the stack, we increment `topOfStack` and then set `theArray[topOfStack] = x`. To pop, we set the return value to `theArray[topOfStack]` and then decrement `topOfStack`.

Notice that these operations are performed in not only constant time, but very fast constant time. On some machines, `push`es and `pop`s (of integers) can be written in one machine instruction, operating on a register with auto-increment and auto-decrement addressing. The fact that most modern machines have stack operations as part of the instruction set enforces the idea that the stack is probably the most fundamental data structure in computer science, after the array.

One problem that affects the efficiency of implementing stacks is error testing. Our linked list implementation carefully checked for errors. As described above, a `pop` on an empty stack or a `push` on a full stack will overflow the array bounds and cause an abnormal termination. This is obviously undesirable, but if checks for these conditions were put in the array implementation, they would be likely to take as much time as the actual stack manipulation. For this reason, it has become a common practice to skimp on error checking in the stack routines, except where error handling is crucial (as in operating systems). Although you can probably get away with this in most cases by declaring the stack to be large enough not to overflow and ensuring that routines that use `pop` never attempt to `pop` an empty stack, this can lead to code that barely works at best, especially when programs are large and are written by more than one person or at more than one time. Because stack operations take such fast constant time, it is rare that a significant part of the running time of a program is spent in these routines. This means that it is generally not justifiable to omit error checks. You should always write the error checks; if they are redundant, you can always comment them out if they really cost too much time. Having said all this, we can now write routines to implement a general stack using arrays.

A stack class, `StackAr` is shown, partially implemented, in Figure 3.42. The remaining stack routines are very simple and follow the written description exactly (see Figs 3.43 to 3.47). Notice that in both `pop` and `topAndPop` we *dereference* (that is, make null) the array reference to the object being removed. This is not required, since it will

Bibliography

[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.

[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. Revised March 17, 2000.

[Els74] Bernard Elspas. The semiautomatic generation of inductive assertions for proving program correctness. Interim Report Project 2686, Stanford Research Institute, Menlo Park, CA, July 1974.

[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.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL*, pages 193–205, January 17–19, 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.

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

[MW77] James H. Morris, Jr. and Ben Wegbreit. Subgoal induction. *Communications of the ACM*, 20(4):209–222, April 1977.

[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.

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, July 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, November 2002.

[Nel80] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Palo Alto, CA, 1980. Also published as Xerox Palo Alto Research Center Research Report CSL-81-10.

[NWE02] Toh Ne Win and Michael Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Lab for Computer Science, May 25, 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.

[Rya59] T. A. Ryan. Multiple comparisons in psychological research. *Psychological Bulletin*, 56:26–47, 1959.

[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.

[Ser00] Silvija Seres. ESC/Java quick reference. Technical Report 2000-004, Compaq Systems Research Center, October 12, 2000. Revised by K. Rustan M. Leino and James B. Saxe, October 2000.

[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.

[WS76] Ben Wegbreit and Jay M. Spitzen. Proving properties of complex data structures. *Journal of the ACM*, 23(2):389–396, April 1976.