Invariant Inference for Static Checking: An Empirical Evaluation

Jeremy W. Nimmer Michael D. Ernst

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

Abstract

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.

This paper describes an evaluation of the effectiveness of two techniques, one static and one dynamic, to assist the annotation process. We quantitatively and qualitatively evaluate 41 programmers using ESC/Java in a program verification task over three small programs, using Houdini for static inference and Daikon for dynamic inference. We also investigate the effect of unsoundness in the dynamic analysis.

Statistically significant results show that both inference tools improve task completion; Daikon enables users to express more correct invariants; unsoundness of the dynamic analysis is little hindrance to users; and users imperfectly exploit Houdini. Interviews indicate that beginning users found Daikon to be helpful; Houdini to be neutral; static checking to be of potential practical use; and both assistance tools to have unique benefits.

Our observations not only provide a critical evaluation of these two techniques, but also highlight important considerations for creating future assistance tools.

1. Introduction

Static analysis is a useful technique for detecting and checking properties of programs. Static analysis can reveal properties that would otherwise be detected only during testing or even after deployment. Static checking is valuable because the earlier in the development process that problems are identified, the less costly they are to correct. Simple static analyses like type-checking are widely applied and successful; analyses like theorem-proving and correctness-checking are still topics of research. The annotations that are checked by analyses such as type-checkers and theorem-provers are useful in their own right: they serve as a machine-verified form of documentation.

Static checking is not used in practice as often as might be desired, largely because of difficulty of use. Static checkers require explicit goals, and often also summaries of unchecked code. These annotations usually must be supplied by the programmer, a task that users find tedious, difficult, and unrewarding, and therefore often refuse to perform [FJL01]. Thus, users seem to view the cost of annotation as greater than the benefits of static checking. While it might be possible to increase the benefits of static checking, this paper considers the alternative of lowering costs by easing the annotation burden.

Automatic annotation of programs is a long-standing research goal, but it does not appear to be close to being solved. In fact, many researchers consider it harder to determine what property to check than to do the checking itself [Weg74, WS76, MW77, Els74, BLS96, BBM97]. Static tools for computing program properties are often stymied by constructs, such as pointers, that are common in real-world programming languages. Manipulating representations of the heap incurs gross runtime and memory costs, or else heap approximations introduced to control costs weaken results. Dynamic techniques, by contrast, have the fundamental limitation of unsoundness due to reliance on a specific test suite. Interaction with a user, or possibly another tool, is required in order to weed out properties that are not universally true from the proposed annotation set.

This research evaluates 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. Our experimental evaluation of users in the task of program annotation reflects the effectiveness of each tool. The experiments also examine how unsoundness affects users' performance by supplying the dynamic analysis with test suites of varying sizes (larger ones produce a more accurate dynamic analysis). The results indicate important considerations for creating future assistant tools.

ESC/Java performs modular checking, both verifying and relying on user-written annotations in order to guarantee the absence of run-time exceptions in a Java program. Houdini works by inserting annotations in addition to the ones written by the user, but removing them if they cannot be verified. The user never sees any of the Houdini-inserted annotations, but the effect is as if the user had written additional verifiable annotations: ESC/Java produces

Categories and Subject Descriptors: D.2.1 [Software Engineering] Requirements/Specifications; D.2.4 [Software Engineering] Software/Program Verification; F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs.
General Terms: Experimentation, Verification, Documentation.

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

fewer warnings. Daikon examines program executions and generalizes from run-time variable values to properties over those values. Properties that were true over the test suite are inserted into the program as annotations that a user may retain or delete, at the user's discretion.

In the experiment, 41 experienced programmers who were new to ESC/Java each annotated two programs for ESC/Java. Users were randomly assigned to programs and to annotation assistants.

In brief, the results suggest that both tools contribute to success, and neither harms users in a measurable way. Unsoundness did not hinder users: even very inaccurate dynamic analysis output produced from tiny test suites is better than no assistance. Additionally, Houdini helps users to express more properties in fewer annotations, and Daikon helps users express more true properties than strictly required, with no time penalty. However, users reported concerns with Houdini's speed and opaqueness and with Daikon's verbosity on poor test suites.

The remainder of this paper is organized as follows. Section 2 provides background on the tools used in this study. Section 3 presents our methodology and describes the factors we measure and compare. Sections 4 and 5 report quantitative and qualitative results. Section 6 discusses the results, and Section 7 concludes.

2. Background

This section provides details about the three tools used in our study: the ESC/Java static checker, the annotation assistant Houdini, and the dynamic invariant detector Daikon, which can also act as an annotation assistant.

2.1 ESC/Java

ESC/Java [Det96, DLNS98, LN98] is an Extended Static Checker for 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. (Figure 1 gives an example of annotations for the `StackAr` class, one of the programs used in this study.) 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.

ESC/Java checks each method in isolation, assuming that all other annotations are correct. This permits checking different parts of a program independently and checking partial programs or modules. ESC/Java took 5–15 seconds to run on each program in our study.

ESC/Java is not sound; for instance, it does not model arithmetic overflow, it assumes that all loops are executed 0 or 1 times, and it permits the user to supply (unverified) assumptions. However, ESC provides a good approximation to soundness: in practice, it detects many potential problems and increases confidence in the program being checked.

There are many other tools besides ESC/Java for statically checking specifications [Pfe92, DC94, EGHT94, NCOD97]. These other

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

 private Object [ ] theArray;
 private int        topOfStack;

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

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

 ...
}
```

Figure 1: Object invariants and two method specifications of the annotated `StackAr.java` file [Wei99].

systems have different strengths and weaknesses than ESC/Java, but few have the polish of its integration with a real programming language. ESC is available from `http://research.compaq.com/SRC/esc/`.

2.2 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 postulated 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].

2.2.1 Emulation

Houdini was not available to us, 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.

2.3 Daikon

Daikon is a system for dynamically detecting likely program invariants [ECGN01]. Daikon discovers likely invariants from program executions by running the program, examining the values that it computes, and detecting patterns and relationships among those values. The system reports properties that hold over execution of an entire test suite (which is provided by the user).

The potential invariants are generated by instantiating, at each procedure entry and exit, each of a set of three dozen invariant templates. Daikon's candidate invariants are richer than those of Houdini; additionally, Daikon can output implications and disjunctions. The templates are filled in with each possible subset of variables that are in scope at the program point (plus certain expressions over those variables). Although there are many potential invariants, testing is efficient because most potential invariants are falsified quickly and need not be tested thereafter.

The output is further improved by suppressing invariants that are not statistically justified, that are implied by other invariants in the output, or that involve variables that can be statically proved to be unrelated [ECGN00].

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. When a reported invariant is not universally true for all possible executions, it indicates a property of the program's context or environment or a deficiency of the test suite, which can then be corrected. The Daikon invariant detector is language independent, and currently includes instrumenters for the C, C++, IOA, and Java languages. Daikon is available from `http://pag.lcs.mit.edu/daikon/`.

Daikon can produce output in a variety of formats, including ESC/Java's annotation language (a variant of JML, the Java Modeling Language [LBR99, LBR00]). Daikon includes a tool that automatically inserts Daikon's output into the target program. The resulting annotated program can be run through ESC/Java, and ESC/Java may report warnings about unverifiable annotations or potential run-time errors.

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 users. We evaluate performance based on the time users took; whether they succeeded; precision and recall, two standard measures of partial success; and other factors (Section 3.4).

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

Figure 2: 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.

Section 3.1 presents the participants' task. Section 3.2 describes participant selection and demographics. Section 3.3 details the experimental design. Section 3.4 describes what data was collected and how it was analyzed.

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 (chosen from three possibilities) in sequence.

Before beginning, participants received a packet [Nim02, App. A] containing six 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 [Wei99]. 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 and quick reference.

The instructions explained the 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. 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.

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 (see Section 5).

3.1.1 Programs

The three programs used for this study were taken from a data structures textbook [Wei99, pp. 78, 89-90, 269-272]. Figure 2 gives some of their characteristics.

We selected three programs for variety. The `DisjSets` class is

	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 3: Study participant demographics. "Dev" is standard deviation.

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 (quite a few users failed to complete it in one hour), 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 (see Section 6).

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 written 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 3 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 authors' research group.

Participants reported their primary development environment, whether they write assert statements in code (never, rarely, sometimes, often, usually, or 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.

3.3 Experimental Design

3.3.1 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 ESC/Java 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,

		NCB	Calls		Coverage		Invariants		
Program	Suite	LOC	Stat.	Dyn.	Stmt.	Bran.	#	Prec.	Rec.
`DisjSets`	Tiny	23	5	389	0.67	0.70	20	0.65	0.57
	Small	28	5	1219	1.00	0.90	28	0.71	0.74
	Good	43	13	11809	1.00	1.00	31	0.94	0.97
`StackAr`	Tiny	14	4	32	0.60	0.56	26	0.54	0.52
	Small	24	5	141	0.64	0.63	29	0.83	0.73
	Good	54	14	2783	0.96	0.94	36	1.00	0.95
`QueueAr`	Tiny	16	4	32	0.68	0.65	46	0.37	0.44
	Small	44	10	490	0.71	0.71	51	0.47	0.56
	Good	66	10	7652	0.94	0.88	53	0.74	0.75

Figure 4: Test suites used for Daikon runs. "NCB 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. "#" is the number of annotations proposed by Daikon, according to the semantic measure of Section 3.4.2. "Prec" is precision, a measure of correctness, and "Rec" is recall, a measure of completeness; see Section 3.4.3.

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. Figure 2 gives the minimal number of such invariants.

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 write enough annotations so that Houdini could complete annotating 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 4). 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 4). We call these the "tiny" test suites, but the term "test suites" is charitable.

Variable	Domain
Independent variables	
Annotation assistant	none, Houdini, Daikon$_{T,S,G}$
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 variables	
Success	yes, no
Time spent	up to 60 minutes
Final written answer	set of annotations (Figure 7)
Nearest verifiable answer	set of annotations (Figure 7)

Figure 5: Variables measured and their domain (set of possible values). We also analyze computed values, such as precision and recall (Section 3.4.3).

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$_{small}$: This group received a program into which different 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, one 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$_{tiny}$, 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$_{good}$: 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 one author about half an hour to produce.

3.3.2 Assignment of treatments

There are 150 possible experimental configurations: six program pairs, five treatments for the first program, and five treatments for the second program. No participant annotated the same program twice, but participants could be assigned the same treatment on both trials. Assigning a (potentially) different treatment for each program avoids conflating subjects with treatments and also permits users to compare multiple treatments (see Section 5).

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. (Disqualified participants were replaced, in order to preserve balance.)

3.4 Analysis

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

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 6: Distinction between lexical and semantic annotations (Sec. 3.4.2). The last annotation is implied because `arg` is not declared to be modified across the call.

3.4.1 Quantities Measured

We are interested in studying what factors affect a user's performance in a program verification task. Figure 5 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.

3.4.2 Measurement Techniques

This section explains how we measured the variables in Figure 5. 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 involved 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 while making as few edits as possible to the user's answer, until there were no warnings. A potential source of error is that we missed a nearer answer. However, most edits were straightforward: some annotations must be present in all answers, and ESC/Java warnings indicate incorrect user annotations. The most significant risk is declining to add an annotation that would prevent removal of others that depend on it. We were careful to avoid this risk.

We count annotations both lexically and semantically (Figure 6). 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 may differ from the lexical count if an annotation expresses multiple properties, or if a user writes essentially the same annotation twice or writes an annotation that is implied by another one. The semantic set of annotations 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 grading code. The method instructs ESC/Java to assume certain conditions and check others; the specific conditions depend on the property being checked. For

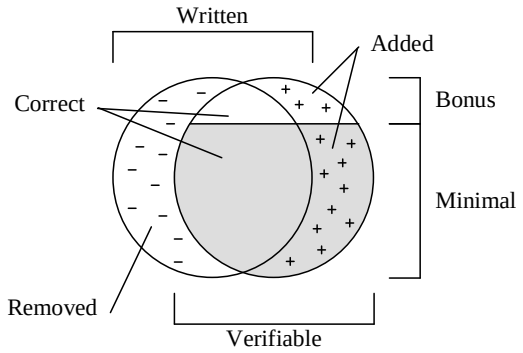

Figure 7: Visualization of written and verifiable sets of annotations. The left circle represents the set of invariants written by the user; the right circle represents the nearest verifiable set. The overlap is correct invariants, while the fringes are additions or deletions. In general, the nearest verifiable set (the right circle) is not necessarily the smallest verifiable set (the shaded region). See Sections 3.4.2 and 3.4.3 for details.

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 and corrected the (infrequent) mistakes. Thus, the automatic grading system ensures that our measurements are reliable, unbiased, and reproducible.

3.4.3 Computed Values

From the written and verifiable sets of annotations, we computed additional values that reflect users' success and that enable comparing results across different programs.

Figure 7 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 semantic annotations. 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 precision and recall (for all users) and unnecessary and boost (for Houdini users only). Precision and recall are the standard measures used in information retrieval [Sal68, vR79], of which our problem is an instance.

Precision measures the correctness of a user's statements. Precision 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 7, the higher the precision.

Recall indicates how many necessary statements a user wrote. Recall 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 7, the higher the recall. In this study, recall was a good measure of a user's progress in the allotted time, since recall varied more than precision.

Bonus indicates how many additional properties the user expressed. Bonus is 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 7, the larger the bonus.

Effects predicted by *Treatment*

Depend. var.	None	Houd.	D_{tiny}	D_{small}	D_{good}	Confidence
Success (¬Q)	36%	71%				$p = 0.03$
		(*83%*)	(*54%*)	(*67%*)	(*100%*)	
Recall	72%	86%				$p = 0.02$
		(*88%*)	(*81%*)	(*86%*)	(*94%*)	
Bonus	1.25		1.47		1.75	$p < 0.01$
	(*1.25*)	(*1.26*)	(*1.44*)	(*1.49*)		

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
Time (¬Q)	49	43	$p = 0.08$

Figure 8: 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, and the indistinguishable means are given in parentheses. Variables are explained in Section 3.4, and effects are detailed in Section 4. (¬Q means that the result holds only if `QueueAr` data, for which no users were successful, is omitted.)

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 (the last `@invariant` in Figure 1), and that `push` and `pop` operations may modify only the top of the stack, instead of any element of the stack (the last `@ensures` in Figure 1).

Unnecessary measures the unnecessary effort expended by Houdini users. It is the fraction of the explicitly written properties that Houdini would have inferred, had the user omitted them.

Boost measures the information added by Houdini. Boost is the fraction of semantic information in the user's answer that was contributed by Houdini; we measured this by grading with and without Houdini enabled. Boost indicates the degree to which Houdini users are able to achieve verification while writing fewer properties. Boost is reduced when unnecessary increases, since Houdini no longer contributes the unnecessarily-written properties. For a single program, boost is correlated with unnecessary, but across different programs, they may have different relationships.

3.4.4 Other Values

We studied the statistical significance of all 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, whether the user used Windows, etc. None of these factors was statistically significant.

4. Quantitative Results

This section presents quantitative results of the experiment, which are summarized in Figure 8. Each subsection discusses one dependent variable and the factors that predict it.

We analyzed all of the sensible combinations of variables listed in Section 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 substantial absolute differences in means are not reported here, though they would be under the methodology of some other studies. For instance, 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.

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 both expected and intended, since some programs are more amenable to a given analysis, and we selected the three programs to be of varying difficulty. While we did not expect `QueueAr` to have no successful users, it permitted us to study unsuccessful 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 assistants were statistically indistinguishable. However, the Daikon$_{\text{good}}$ users always succeeded, which was not the case for any other tool.)

These results suggest that some programs are difficult to annotate, whether or not either Daikon or Houdini is assisting. `QueueAr` requires unusually complex invariants (see Figure 9 and Section 6). Furthermore, for more easily-expressible invariants, tool assistance improves the success rate by a factor of two.

4.2 Time

We measured time to determine what factors may speed or slow a user. Perhaps evaluating Daikon's suggested annotations — particularly wrong ones — 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 predicts 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.

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 unsound 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, no treatment group had 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 5.3.

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 86% when any tool was used. 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. In each case, the effects among tools were statistically indistinguishable.

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.

4.5 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 Daikon$_{\text{good}}$ treatment had a mean bonus of 1.75. Users with Daikon$_{\text{tiny}}$ or Daikon$_{\text{small}}$ had a mean bonus of 1.47, while users with Houdini or no tool had a mean of 1.25.

For successful users, the treatment also predicted bonus information ($p < 0.01$). (The verifiable set for unsuccessful users includes annotations that they did not write; consider the top three $+$'s in Figure 7, which might be necessary to avoid creating a larger number of $-$'s.) Daikon$_{\text{good}}$ users had a mean ratio of 1.71, Daikon$_{\text{tiny}}$ and 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 improves the program's documentation and may be helpful for other tasks, such as verifying correctness (as opposed to mere lack of runtime exceptions).

4.6 Unnecessary

For Houdini trials, we measured the fraction of the answer that the user unnecessarily wrote. (The computation is explained in Sec-

tion 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 mean unnecessary percentage for all Houdini users was 26%, and no factor predicted it overall. However, if only object invariants are considered, both program and experience were predictors. `DisjSets` users had a mean unnecessary percentage of 5%, in line with the overall mean. However, `StackAr` and `QueueAr` users had a joint percentage of 42%. Furthermore, users on the first trial had a mean percentage of 9% while users on the second trial had 42%. Surprisingly, second-time users were *more* likely to write object invariants that would have been inferred by Houdini.

These results suggest that users in our study (who have little Houdini experience) may not understand what annotations Houdini may infer, and frequently write inferable invariants. This effect increases with program difficulty; perhaps struggling users tend to write more unnecessary invariants. The effect also occurs on the second trial; users who are more familiar with annotating programs write more unnecessary ones, indicating the omissions on the first trial may have been accidental. Alternately, perhaps users find that explicitly written object invariants help them reason about code, whereas they paid less attention to method specifications.

Related qualitative results are presented in Section 5.2.

4.7 Boost

We measured the relative amount of information Houdini added to verification to examine whether Houdini users are able to make use of it as claimed — by achieving verification while using fewer written properties.

The mean boost for all Houdini users was 35%, and no factor predicted it overall. However, if we distinguish object and method invariants, both program and experience predicted boost. On method annotations, `StackAr` users had a mean boost of 10%, `QueueAr` users had a mean boost of 32%, and `DisjSets` users had a mean boost of 52%. On object invariants, users on the first trial had a mean boost of 66% while users on the second trial had 29%.

The effect of the program on boost can be explained by the fraction of the required annotations that are in Houdini's grammar; `StackAr` had relatively few, while `DisjSets` had many. The effect of experience is largely explained by the increase in unnecessary annotations on the second trial — when the user writes annotations that Houdini could have inferred, Houdini's contribution is less.

These results suggest that Houdini users on average receive one third of their answer from the tool, that Houdini's grammar has a noticeable effect on its contribution, and that unnecessary annotations from the user temper Houdini's effectiveness.

5. Qualitative Results

This section presents qualitative results gathered from exit interviews conducted after each user finished all tasks.

5.1 General

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

Users reported it was not efficient to annotate the program incrementally, in response to ESC/Java feedback. 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 not to a statistically significant degree). 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. Our experimental setting, in which users were asked to complete a task as quickly as possible, may have encouraged an incremental approach.

Users reported difficulty in eliminating ESC/Java warnings. Users said that ESC/Java's suggested fixes were obvious and unhelpful. The `exsures` annotations were particularly troublesome. Despite being explicitly told that exceptional post-conditions refer to post-state values of the variables, many 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 over pointers, 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.

5.2 Houdini

A total of 14 participants used Houdini for at least one program. Three users had positive opinions, five were neutral, and six were negative.

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.

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.

Users' main complaint was that Houdini was too slow, even though our re-implementation was faster than the original could have been. 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 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 to us.)

5.3 Daikon

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, some users appreciated object invariants because they were more difficult to discover than method annotations.

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. (These contained 37%–83% useful annotations; see Fig-

ures 4 and 2.) 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.

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.

5.4 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.

6. Threats to Validity

We have carefully evaluated how static and dynamic assistance tools affect users in a verification task. 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. (In many cases substantial 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 9: 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.

The subjects in our experiment had no previous experience with the tools. Experienced users may be affected by tool assistance in different ways, 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 would be too time-consuming. We mitigated the effect of inexperience by giving users a tutorial that took them step-by-step through an verification exercise exactly like the one they would perform. We also inserted boilerplate and obscure annotations, permitting users to focus on the interesting part of the task. Thus, the subjects were much more effective than a typical beginning user would have been, but nonetheless our results might not generalize to non-novices.

Each volunteer subject spent a total of three hours, which restricted the size of the programs we could study. Our results may not generalize to larger programs, particularly if larger programs have more complex invariants. That no users succeeded on `QueueAr` may also seem to support this argument. We chose `QueueAr` as an unusually difficult example for the user study; it guarantees that unused storage is set to null via the invariants shown in Figure 9. These invariants were difficult for users to generate, particularly because of ESC/Java's syntax limitations; furthermore, Houdini and Daikon do not suggest such complicated properties. However, larger programs do not necessarily require such complex invariants. 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. Other experiments [NE02] 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; none of these annotations was complex. By contrast, `QueueAr` required one annotation per 1.7 lines. (Houdini's authors say that with its assistance, programmers may only need to insert about one annotation per 100 lines of code [FL01].) Finally, both Daikon and ESC/Java are modular (operate one class at a time): the invariants detected by Daikon and required for ESC/Java verification are local to one class. This modularity limits the difficulty for a user and suggests the tools will scale to larger programs.

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. We have evaluated two annotation assistance tools that lower the costs of static checking (Houdini does static analysis, and Daikon does dynamic analysis). We also investigated the effect on users of unsoundness in dynamic analysis output, resulting from test suites of varying quality.

The results demonstrate that assistants need not be perfect. Some Daikon output contained numerous incorrect invariants (see Figure 4), but it 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 test suites were used. Users effortlessly discard poor suggestions and readily take advantage of correct ones.

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

Another conclusion is the importance of relatively complete generation of candidate annotations. 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. Neither tool (and no set of test suites) fared statistically significantly better in terms of success or time. However, except in `QueueAr`, users given Daikon's output from a modest test suite ($\text{Daikon}_{\text{good}}$) always succeeded, whereas 36–83% of other users succeeded. Furthermore, Daikon users achieved a more complete specification, which serves as documentation and can be useful for other tasks. 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, hiding inferred annotations confused users (leading to unnecessary annotations). A user interface that allows users to toggle annotations might help.

In sum, both assistants increased users' effectiveness in completing a 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 information expressed by the user, and was effective even in the presence of limited test suites.

Acknowledgments

We thank the volunteer study participants for their time and feedback. We also thank the members of the Daikon group — particularly Alan Donovan, Michael Harder, and Ben Morse — for their contributions to this project. The presentation was improved by comments from Steven Wolfman and the anonymous referees. Vibha Sazawal provided statistical consulting. This research was supported in part by NSF grant CCR-0133580 and a gift from NTT.

References

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

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

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

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

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

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001. A previous version appeared in *ICSE*, pages 213–224, Los Angeles, CA, USA, May 1999.

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

[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.

[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.

[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`.

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

[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.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[Nim02] Jeremy W. Nimmer. Automatic generation and checking of program specifications. Technical Report 852, MIT Lab for Computer Science, June 10, 2002. Revision of author's Master's thesis.

[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.

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

[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.

[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.