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
http://sdg.lcs.mit.edu/daikon/

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 to assist the annotation process: inference via static analysis and inference via dynamic invariant detection. We quantitatively and qualitatively evaluate 33 users in a program verification task over three small programs, using ESC/Java as the static checker, Houdini for static inference, and Daikon for dynamic detection. With a well-constructed test suite, Daikon produces fully-verifiable annotations; therefore, we supplied Daikon with poor test suites to study its effectiveness in suboptimal circumstances.

Statistically significant results show that Daikon enables users to express more correct invariants; Houdini users do not take full advantage its capabilities; and both tools improve task completion. Interviews suggest 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 benefits.

1. Introduction

Static analysis is a useful technique for detecting and checking properties of programs. A static analysis can reveal properties that would otherwise have be detected only during testing or even deployment. This is valuable because the earlier in the development process that problems can be identified, the less costly they are to correct. Simple static analyses like type-checking are widely applied and successful; more complicated analyses like theorem-proving and correctness-checking are still topics of research. 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 desirable, largely because of cost in human time. Static checkers require explicit goals for checking, 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]. The annotation cost is so high that it is not offset by 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.

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. The cost of manipulating representations of the heap is so great that either runtime and memory increase unreasonably, or heap approximations introduced to control costs result in overly weak 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.

In this research, we consider techniques for easing the annotation burden by applying two annotation assistant tools, one static (Houdini) and one dynamic (Daikon), to the problem of annotating a Java program for the ESC/Java static checker. We performed an experiment to evaluate the tools' effectiveness in assisting users in the task of program annotation.

ESC/Java performs modular checking, both verifying and relying on user-written annotations in order to guarantee the lack 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 a additional verifiable annotations: ESC/Java produces fewer

warnings. Daikon examines program executions and generalizes from run-time variable values to properties over those values. Properties that were true over the entire test suite are inserted into the program as annotations which a user may retain or delete, at the user's discretion. We deliberately supplied Daikon with small test suites to test its performance in that situation, because with good test suites Daikon produces fully verifiable sets, so no user effort is required whatsoever [NE01b, NE01a].

In our experiment, 33 experienced programmers each annotated two programs for ESC/Java; users were randomly assigned to programs and to annotation assistants.

In brief, results suggest that both tools contribute to success, and neither harms users in a measurable way. Additionally, Houdini helps users to express more properties in fewer annotations, and Daikon helps users express more properties than required with no loss of time. However, users report concerns with Houdini's speed and opaqueness, and Daikon's verbosity.

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. Sections 4 and 5 report quantitative and qualitative results. Section 6 examines the results and concludes.

2. Background

This section provides details about the three tools used in our study: the ESC/Java static checker (Section 2.1), the annotation assistant Houdini (Section 2.2), and the dynamic invariant detector Daikon (Section 2.3), 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. ESC issues warnings about annotations that cannot be verified and about potential run-time errors. Its output may also include suggestions for correcting the problem or stylized counterexamples showing an execution path that violates 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/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, Det96, Eva96, NCOD97, LN98]. These other systems have different strengths and weaknesses than ESC/Java, but few have the polish of its integration with a real programming language.

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 follow from those, permitting users to write fewer annotations and end up with less cluttered, but still automatically verifiable, programs.

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 ESC/Java checker as a subroutine and removes unprovable annotations, until no more annotations are refuted. If even one required invariant is missing, then Houdini eliminates all other invariants 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. With the assistance of Houdini, programmers may only need to insert about one annotation per 100 lines of code.

2.2.1 Emulation

Houdini is not publicly available, so we were forced to reimplement 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 Houdini, except that it may 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.

2.3 Daikon

Daikon is a system for dynamically detecting likely program invariants [ECGN01, Ern00]. 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 (plus certain expressions over those variables) that are in scope at the program point. 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, then 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 [KR88], IOA [GL00], and Java [AGH00] languages. Daikon is available from `http://sdg.lcs.mit.edu/daikon/`.

Daikon can produce output in a variety of formats, including ESC/Java's annotation language (a variant of Java Modeling Language JML [LBR99, LBR00]). For the purposes of this research, we extended Daikon with a tool that automatically inserts its output into the program being analyzed as ESC/Java annotations. The resulting program can be run through ESC/Java (which may report warnings about unverifiable annotations or potential run-time errors).

3. Methodology

The section presents our experimental methodology and its rationale. Section 3.1 presents the participants' task. Section 3.2 describes participant selection and characteristics. Section 3.3 details our experimental design. Section 3.4 describes how the data was collected and 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 in sequence.

Before beginning, participants received a packet 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].

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

Figure 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" is the minimal number of annotations necessary to complete the task.

The instructions explained the programming task 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.

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 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. (Section 5 presents qualitative results from these interviews.)

3.1.1 Programs

The three programs used for this study were taken from a data structures textbook [Wei99]. Figure 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 specifically because they are relatively straightforward ADTs, and had some test suite included. The programs are not trivial for the annotation task, but are not so large as to be unmanageable. Since the annotations required for verifying absence of runtime errors overwhelmingly focus on class-specific properties, we expect results on

	Mean	Dev.	Min.	Max.
Years of college education	7.1	2.7	3	14
Years programming	11.6	5.2	4	20
Years Java programming	3.6	1.4	1	7

	Frequencies
Usual environment	Unix 57%; Win 15%; both 28%
Writes asserts in code	"often" 31%; less frequently 69%
... in comments	"often" 25%; less frequently 75%
Gender	male 88%; female 12%

Figure 2: Demographics of study participants. "Dev" is standard deviation.

small programs such as these to imply similar results for large programs.

3.2 Participants

A total of 39 users participated in the study, but six were disqualified, leaving data from 33 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 2 provides background information on the 39 participants. Participants were experienced programmers and were familiar with Java programming, but none had ever used ESC/Java before. Participants had at least 3 years of post-high education, and most were graduate students in Computer Science at MIT or the University of Washington.

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.

3.3 Experimental Design

3.3.1 Treatments

The experiment used four experimental treatments: a control group, Houdini, and two Daikon groups.

Control. Some participants were given the original program without any help from an annotation assistant and with only a minimal set of ESC/Java annotations already inserted in the program.

The minimal set of ESC/Java annotations that were provided to all groups 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. This boilerplate is easy to add automatically.

Houdini. This group was provided the same source code as the control group, but a version of ESC/Java enhanced with (our re-implementation of) Houdini. Participants did not have to do anything special in order to invoke it; for these users, it was automatically invoked (and a message printed) when the user ran `escjava`.

Program	Suite	NCNB LOC	Calls Stat.	Calls Dyn.	Prec.	Rec.
`DisjSets`	Tiny	23	5	389	0.65	0.57
	Small	28	5	1219	0.71	0.74
`StackAr`	Tiny	14	4	32	0.54	0.52
	Small	24	5	141	0.83	0.73
`QueueAr`	Tiny	16	4	32	0.37	0.44
	Small	44	10	490	0.47	0.56

Figure 3: Test suites used for Daikon runs. "NBNC LOC" is the non-comment, non-blank lines of code. "Stat" and "Dyn" are the static and dynamic number of calls to the ADT. "Prec" is precision, a measure of correctness, and "Rec" is recall, a measure of completeness.

Daikon$_{\mathbf{tiny}}$. This group received a program into which Daikon output had been automatically inserted as ESC/Java annotations.

The Daikon output was 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 3). We call these the "tiny" test suites, even though the term "test suites" is charitable.

These suites seem much less exhaustive than would be used in practice. Our rationale for using them is that users may not already have a good test suite available to them, 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.

These participants ran an unmodified version of ESC/Java. There was no sense also supplying Houdini to participants who were given Daikon annotations, since Daikon always produces all the invariants that Houdini might infer. Participants were not provided the test suite and did not run Daikon themselves. (Daikon took only a few seconds to run on these programs.)

Daikon$_{\mathbf{small}}$. 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 which varied some values and added a few more method calls. In order to construct this suite, one author limited himself to 3 minutes of wall clock time for each of `DisjSets` and `StackAr`, and 5 minutes for `QueueAr`, in order to simulate low-cost testing methodology; see Figure 3. As in the case of Daikon$_{\text{tiny}}$, use of these suites measures performance when invariants are detected from an inadequate test suite. We call these the "small" suites.

Past research has shown that, given an adequate test suite, Daikon produces fully-verifiable annotation sets for these programs [NE01b, NE01a] (except for one missing annotation required by `StackAr`'s client). Those adequate test suites take about half an hour to produce.

3.3.2 Assignment of treatments

There are a total of 96 possible experimental configurations: no participant annotated the same program twice so

Variable	Domain
Independent	
Annotation assistant	none, Houdini, Daikon
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)
Nearest verifiable answer	set of annotations (Fig. 5)

Figure 4: Variables studied (Section 3.4.1), and their domain (set of possible values). We also compute derived variables, such as precision and recall (Section 3.4.3).

there are six choices of program pairs; there are four possible treatments for the first program; and there are four possible treatments for the second program. Participants could be assigned the same treatment on both trials.

In order to reduce costs, we ran only a subset of the 96 configurations. We assigned the first 32 participants to configurations using a randomized partial factorial design, then assigned the last participant randomly to one of the remaining configurations. (Participants who were disqualified had their places taken by subsequent participants, in order to preserve balance.)

3.4 Analysis

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

3.4.1 Quantities Measured

We are interested in studying what factors affect a user's performance in a program verification task. Figure 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 independent variables in order to identify other factors which have an effect, or to rule out other possibilities and 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 measures of success. 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 (in part) 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 the variables in Figure 4 were measured. The annotation assistant, program, and experience are derived from the configuration. The other independent variables were reported by the participant. 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 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 which must be present in all answers. Removing annotations was also straightforward: incorrect statements cause warnings from ESC/Java, so the statements may be easily identified and removed. The most significant risk is declining to add an annotation which 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.

Lexical annotation measurements count the textual lines of annotations in the source file. Essentially, it 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.

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 syntactic choices of users, and it accounts for unexpressed but logically derivable properties.

To measure the semantic set, we created specially-formed calling code for 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 which meets the condition, and one call which breaks it. If the first passes but the second fails, the precondition is present. Similar techniques exist for modifies clauses and postconditions.

This automatic grading system ensures that our measurements are reliable and unbiased, and enables reproducibility. The grading system may break down if users write bizarre or unexpected annotations. These mistakes are not common, and we corrected them by hand.

3.4.3 Computed Values

From the quantities directly measured (Figure 4) we computed additional variables to to focus on more specifically useful values, and to compare results across differing programs.

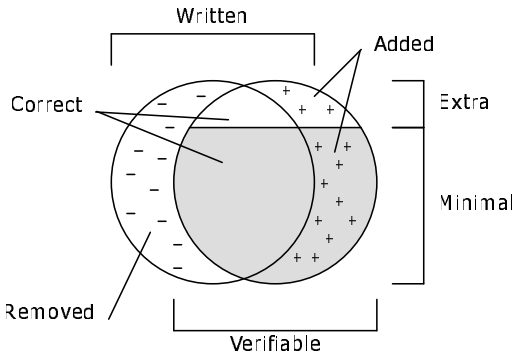

Figure 5: Visualization of written and verifiable sets of annotations. The left circle represents the set of invariants written by the user; the right circle represents the nearest verifiable set. The overlap is correct invariants, while the fringes are additions or deletions. In general, the nearest verifiable set is not necessarily the smallest verifiable set. See Section 3.4.2 for details.

Figure 5 presents a visualization of 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. Unverifiable annotations were removed ("Removed"), while other annotations may have been added ("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 ("Extra"). The minimal set does not depend on the user's written annotations.

From these measurements, we compute several values, all of which were computed automatically.

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, 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, the higher the recall. We measure recall to determine how many necessary statements a user wrote. Additionally, in this study, recall is 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.

Dependent variable	Independent variable
Success	Program to be verified Without `QueueAr`: Any tool (+109%)
Time	Program to be verified If successful: Second trial (+26%)
Precision	Program to be verified
Recall	Program to be verified Without `QueueAr`: Any tool (+25%)
Density	Houdini (+57%)
Bonus	Daikon (+17%)

Figure 6: For each dependent variable, the independent variables that predict performance, and their numerical improvement, if appropriate. Variables are explained in Section 3.4, and effects are discussed in Section 4. All major statistically justified results are shown above.

We created a fraction from this set by dividing its size by the number of properties inferable by Houdini. 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 unshaded arc in Figure 5, the larger the bonus. We measured bonus to judge the total amount of annotations the user expressed in a program-independent way.

3.4.4 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 at all, etc. None of these factors were statistically significant.

4. Quantitative Results

This section presents quantitative results of the experiment, which are summarized in Figure 6. 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 large absolute differences in means are not reported here, even though in the absence of a statistical analysis, the effects might appear to be clear. 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 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 70% for `DisjSets`, 48% 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 predicts success ($p = 0.07$). Users with no tool succeed 33% of the time, while users with either Daikon or Houdini succeed 69% of the time (the effects of the assistants were statistically indistinguishable).

These results suggest that programs such as `QueueAr` that require complicated object invariants are difficult to annotate, whether or not either Daikon or Houdini is assisting. Furthermore, for less complex programs, tool assistance improves success by more than 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 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`, 52 for `StackAr`, and 60 for `QueueAr`. Furthermore, if the user was successful, then experience also predicted time ($p = 0.02$). Successful first-time users averaged 43 minutes, while successful second-time users averaged 34.

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 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 87%.

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.

Related qualitative results are presented in Section 5.3.3.

4.4 Recall

We measured recall, the fraction of the necessary annotations that are written by the user, 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 achieve more progress in the time allotted.

As expected, recall was predicted by the program under test ($p < 0.01$). Mean recall was 95% for `DisjSets`, 84% for `StackAr`, and 63% for `QueueAr`.

Recall was not universally affected by treatment. However, if the `QueueAr` trials are removed, recall is helped by any tool ($p < 0.01$). Users with no tool had a mean recall of 76%, while users with any tool had a mean recall of 95%.

For unsuccessful users, only data for `StackAr` trials showed an effect ($p < 0.01$). Daikon users had a mean recall of 83%, while users with no tool had mean recall of 48%. (No Houdini users failed, so the Houdini treatment cannot be judged for unsuccessful users. Overall, Houdini users had success rates indistinguishable from those of Daikon users; see Section 4.1.)

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 more properties with fewer written annotations.

The only factor that predicted the density was treatment ($p < 0.01$). Houdini users had a mean density of 1.63 semantic properties per written statement, while non-Houdini users had a mean of 1.04. Daikon was indistinguishable from the null treatment.

4.6 Redundancy

For Houdini trials, we measured the redundancy level of users' annotations in relation to the annotations that Houdini may infer (the redundancy computation is explained in Section 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 ($p = 0.10$). Users on the first trial had a mean redundancy of 18%, while users on the second trial had a mean redundancy of 55%. Surprisingly, second-time users were more likely to write annotations that would have been inferred by Houdini.

Overall, users redundantly wrote half 51%of the available method annotations. For object invariants, though, users wrote more redundant annotations as program difficulty increased (17% for `DisjSets`, 31% for `StackAr`, and 60% for `QueueAr`).

These results suggest that users with 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 5.2.3

4.7 Bonus

We measured the relative size of a user's verifiable set of annotations compared to the minimal 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 ($p < 0.01$). Daikon users had a mean ratio of 1.47, while users with Houdini or no tool had a mean of 1.26.

Since the verifiable set for unsuccessful users includes annotations that they did not write, examining the same measurements for successful users is informative. For successful users, the treatment also predicted bonus information

($p < 0.01$). Daikon users had a mean ratio of 1.50, while others had a mean of 1.22.

These results suggest that Daikon users express a broader range of verifiable properties, with no harm to time or success at the given task. The extra properties were not needed for the task studied in this experiment, but may be helpful for other, similar tasks.

5. Qualitative Results

This section presents qualitative results gathered from exit interviews conducted after each user finished all tasks. Section 5.1 briefly covers general feedback. Section 5.2 describes experiences with Houdini and Section 5.3 describes experiences with Daikon.

5.1 General

While the main goal of this paper 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.

5.1.1 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 (although there are not enough values to statistically justify a conclusion). Additionally, a few users who worked incrementally for the whole experiment reported that an initial attempt at writing relevant annotations at the start would have helped. Notably, all users who were given Daikon annotations for program decided to work incrementally.

5.1.2 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, and that they wanted more useful help. 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, or 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. A common suggestion was to display concrete variable values that would have caused the exception.

5.2 Houdini

Users' descriptions of experiences with Houdini help examine 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.

5.2.1 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.

5.2.2 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.

5.2.3 Slow and confusing

The six negative opinions provide the insight into Houdini's weaknesses. The 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 was supported by the numerical results above having to do with redundancy. Some users wished that the annotations inferred by Houdini could have been shown to them upon request, to aid in understanding what properties already present.

5.3 Daikon

5.3.1 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.

5.3.2 Overload

About a third of the Daikon 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 other said the size was a persistent problem.

5.3.3 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 the truthfulness of a correct annotation (or the absence of 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 Daikon 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 syntax hurdles could be overcome, or if a large system already existed and needed to be checked.

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 would rather build a larger test suite rather than annotate 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. Discussion

Static checking is a useful software engineering practice. It can reveal errors that would otherwise have be detected only during testing or even deployment. Participants in this study recognized its advantage, but for some, the costs outweigh the benefits.

We have evaluated two assistance tools both quantitatively and qualitatively. Both tools aided success by a factor of two. Furthermore, neither tool hurt users in any quantitative measure.

Houdini had both benefits and drawbacks. Participants appreciated some aspects of its behind-the-scenes invariant inference. For instance, removing the need to write certain properties was helpful, as was less annotation clutter in the source code. However, Houdini's process can be misunderstood, and can be too slow at times. (The un-emulated version may be even slower.) Additionally, new users may not be helped as much as experienced users, since they do not view the inferred invariants.

Daikon, too, had both benefits and drawbacks. New users appreciated the inserted annotations as training wheels, and all users ended up writing bonus properties. Users appreciated the suggestions, but were sometimes unhappy with the volume of suggestions produced by the deficient test suites. These results were obtained by using Daikon on (intentionally) very poor test suites. With better test suites (which Daikon may help produce), it should be even more effective at helping users.

These results suggest a few important characteristics of an annotation assistant:

- Assistants should provide helpful and realistic examples for new users — users enjoy learning by example. Even an assistant which reduces total workload still needs to ease the leaning curve for the remainder.
- Reducing total workload is valuable. When users do not have to think of as many annotations on their own, they are more successful.
- When users are shown a starting set of annotations, the set should not be too large or verbose. In the study, picking through the set is not difficult, but was not enjoyable for users.
- A permanent (final) set of annotations should not clutter the code; inferring properties is helpful. However, always hiding annotations is confusing. This suggests a user interface that allows toggling of annotations at the direction of the user.
- Assistants must be fast. During the annotation process, users are not willing to wait for a minute or two while one source file is processed.
- Assistants need not be perfect. Daikon's output contained numerous incorrect invariants (see Figure 3), but Daikon did not slow down users and helped them write more correct annotations.

Acknowledgments

We thank the members of the Daikon group — particularly Alan Donovan, Michael Harder, and Ben Morse — for their contributions to this project. Steve Wolfman made helpful comments on a draft of this paper. This research was supported in part by NSF grants CCR-9970985 and CCR-6891317.

References

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Addison Wesley, Boston, MA, third edition, 2000.

[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 Work-*

shop 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.

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

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

[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.

[GL00] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 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`.

[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.

[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.

[NE01a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation and checking of program specifications. Technical Report 823, MIT Lab for Computer Science, August 2001.

[NE01b] 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.

[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.

[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.

[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.

[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.