Understanding User Understanding: Determining Correctness of Generated Program Invariants

Matt Staats[†], Shin Hong[*], Moonzoo Kim[*], and Gregg Rothermel[† ⋆]

[†]Div. of Web Science and Tech.
KAIST
Daejeon, South Korea
staatsm@kaist.ac.kr

[*]Computer Science Department
KAIST
Daejeon, South Korea
{hongshin|moonzoo}@cs.kaist.ac.kr

[⋆]Dept. of Comp Science
University of Nebraska-Lincoln
Lincoln, NE, USA
grother@cse.unl.edu

ABSTRACT

Recently, work has begun on automating the generation of *test oracles*, which are necessary to fully automate the testing process. One approach to such automation involves *dynamic invariant generation*, which extracts invariants from program executions. To use such invariants as test oracles, however, it is necessary to distinguish correct from incorrect invariants, a process that currently requires human intervention. In this work we examine this process. In particular, we examine the ability of 30 users, across two empirical studies, to classify invariants generated from three Java programs. Our results indicate that users struggle to classify generated invariants: on average, they misclassify 9.1% to 31.7% of correct invariants and 26.1%-58.6% of incorrect invariants. These results contradict prior studies that suggest that classification by users is easy, and indicate that further work needs to be done to bridge the gap between the effectiveness of dynamic invariant generation in theory, and the ability of users to apply it in practice. Along these lines, we suggest several areas for future work.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Oracle generation, dynamic invariant generation, testing judgement

1. INTRODUCTION

Software testing involves two key components: *test inputs* and *test oracles*. Test inputs are executed against the system under test (SUT), and test oracles determine whether the SUT executes correctly. In testing research, significant effort has been expended on developing automatic methods for test input generation, resulting in many approaches (e.g., [13, 14, 18, 27]). Unfortunately, methods for automatically generating test oracles are less common. In the absence of such test oracles, automatic test generation approaches are limited to detecting a small subset of errors [2, 8].

ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA

Recent work has explored methods for automatically inferring invariants from program information [7, 9]. Such work promises (among other things) to help automate the construction of test oracles by allowing users to generate an invariant-based specification built on method pre and post conditions, and class invariants [23, 26, 30, 31]. Given invariants, we can automatically determine if program behavior is correct by monitoring for invariant violations, allowing automated testing. For example, in regression testing, test engineers can extract invariants from a previous program version, and check whether these invariants still hold on a new version.

Unfortunately, existing approaches for inferring invariants necessarily require human intervention for two reasons. First, invariants are intended to act as specifications, but are generated from the source code we wish to verify. Perfectly extracting what the program should do from what the program actually does is impossible. Second, many existing approaches are *dynamic* [9, 30, 31], and use only a finite number of program traces to generate "likely" invariants, rather than provably correct invariants.[1] For such approaches, generation of incorrect invariants appears to occur frequently: one study found that even when using large test suites, 0-60% of assertions inferred from real world Eiffel programs using dynamic generation are incorrect, with an average of 10% [26]. In either case, automated invariant generation may generate invariants that do not accurately capture the program's intended behavior, rendering them unsuitable for use as test oracles. It ultimately falls to human users to filter the resulting invariants, removing incorrect invariants while retaining correct invariants.

We believe that the ability of users to filter invariants is *crucial* to the practical use of automatic invariant generation. Consider the impact of user errors in the context of software testing. While existing work shows generated invariants can effectively describe program behavior [10, 26, 30], making them effective test oracles, if users discard correct invariants the ability to detect faults will decrease. Conversely, if users retain incorrect invariants, then the testing process can yield false error reports, and effort must be spent eliminating spurious errors.

Thus, if *user classification effectiveness* — defined as the percentage of invariants a user correctly classifies as correct or incorrect — is high in practice, then automatic invariant generation is a potentially effective method for generating automated test oracles, and existing results demonstrating the power of invariant generation may hold in practice. However, user classification effectiveness has, to this point, received little attention. Early work explored the ability of users to use dynamic invariant generation as an aid to static verification, concluding that distinguishing correct and incorrect invariants was "relatively easy" for users [23]. Unfortunately, this study did not rigorously consider user effectiveness, and it was

[1]This is often true of static approaches as well, due to abstraction.

conducted with static analysis tools, which are not always available.

In this work, we present the results of two empirical empirical empirical empirical empirical empirical empirical empirical studies with 30 human participants studying user classification effectiveness. The goal of this work was twofold: to determine user classification effectiveness for invariants generated using dynamic invariant generation, and to understand what factors lead to successful or unsuccessful classification. In each study, participants were given one of three Java classes with automatically generated invariants. Invariants were generated using *Daikon*, a dynamic inference tool with a strong body of supporting research [9, 10, 23, 26]. Participants were asked to determine, for each generated invariant, if the invariant was correct or incorrect with respect to the Java class. We then computed the user classification effectiveness for each participant, and analyzed the results to assess factors that impact effectiveness.

Our studies yield two key results. First, contrary to the implications of prior studies, users struggle to correctly classify invariants. On average, our study participants misclassified 9.1-39.8% of correct invariants and 26.1-58.6% of incorrect invariants. We found these results surprising, because not only did participants fail to detect incorrect invariants, they also frequently misclassified correct invariants as incorrect. Thus, dynamic invariant generation can be improved not only by reducing the number of incorrect invariants generated (as previously suggested by several authors [3, 30, 31]), limiting the need to manually detect incorrect invariants, but also by developing tools that help users understand why invariants are, in fact, correct.

Second, the factors that lead to invariant misclassification appear surprisingly subtle. Despite examining a large number of factors, we were unable to clearly determine why users perform poorly at the classification task. Factors we considered included program complexity, user ability in terms of GPA and years of experience, individual invariant complexity and composition, and several others. However, no single factor or set of factors strongly correlated with user classification effectiveness. We did note, however, that the ability to classify difficult invariants strongly correlates with a user's overall ability to classify invariants. This indicates that variation in the difficulty of classifying invariants is related to some set of factors impacting user effectiveness (i.e., variation is not due to chance), and methods for detecting and handling challenging invariants can be developed.

The specific contributions of this paper include:

- Two user studies investigating the ability of users to accurately classify invariants produced by dynamic invariant generation. The data derived from these studies is available for examination by other researchers.
- Evidence that users struggle to classify both correct and incorrect invariants, contradicting previous work.
- Analysis of the impact of several factors on user effectiveness, including program complexity, user-related metrics, and invariant-related metrics.
- Discussion of the results, specifically highlighting potential methods for improving the usability of dynamic invariant generation via discussing factors impacting effectiveness.

2. RELATED WORK

Significant work has been directed at automated test input generation techniques (see [8] for a survey), but the effectiveness of such techniques is limited by the absence of automated test oracles. In the absence of oracles, only "general" properties can be checked, such as null pointer dereferencing, array bound errors, and program crashes [1]. Recently, work has begun on automated support for test oracle construction, but such work (like invariant generation) requires user support [12, 28, 29].

Dynamic invariant generation was pioneered by Ernst et al. [11] in the form of Daikon, in response to the limitations of existing static methods. The goal of dynamic invariant generation is to infer method preconditions, postconditions, and case invariants by observing program traces. Several applications of dynamic invariant generation have been proposed; notably, the use of dynamic invariant generation to support automated software testing and program evolution has been frequently cited as a potential benefit [10, 19, 31, 32]. A number of improvements have been made to Daikon over the years [9, 25], with the current system the fourth version and, to the best of our knowledge, the only system freely available.

Several other dynamic invariant generation approaches have been created. Agitator is a commercial product incorporating dynamic invariant generation in the spirit of Daikon [3]. Tillmann et al. created Axiom Meister for the .NET platform [30], and Wei et al. developed AutoInfer for Eiffel programs [31]. Each approach is conceptually the same as that of Daikon, but employs various extensions in order to improve performance. Note that each of these methods is unsound, and thus like Daikon some generated invariants are incorrect (81.3-100.0% for Axiom Meister [30], and 88.0-98.0% for AutoInfer [31]). Thus the key issue in our study, the need for users to understand generated invariants, remains.

Few studies on the effectiveness of dynamic invariant generation exist. Polikarpova et al. explore the effectiveness of Daikon's generated invariants versus programmer-written invariants [26]. They use several programs written in Eiffel and find that, for large test suites, 40-100% of inferred assertions are correct, with an average of 90%; users must detect the remaining incorrect invariants.

To the best of our knowledge, Nimmer and Ernst performed the only other user study on dynamic invariant generation [23]. In this study, the effectiveness of 33 participants (mostly MIT or University of Washington graduate students) at performing static program checking when aided by Daikon and the ESC/Java static analysis tool was explored [6]. Quantitative results indicate that using Daikon improved the likelihood of success (i.e., verifying program correctness) in the study for two of three case examples used. Qualitative results indicate that removing incorrect invariants generated by Daikon was usually easy, with only invariants poorly understood by ESC/Java being problematic. Our study differs in the context (software testing), the absence of external tools to detect incorrect invariants, and the chief metric used (percentage of invariants correctly classified). Our results show that users do struggle to detect incorrect invariants, and often misclassify correct invariants.

3. STUDY DESIGN

The goal of our study was to examine one key question: *how effective are users at classifying automatically generated invariants as correct or incorrect?* Our interest in this question is motivated from a testing perspective, as automatic invariant generation has been proposed as a method for constructing test oracles. This perspective motivates much of our study design. For example, static analysis tools (as used in [23]) were not used by participants, as we cannot assume their availability when testing. Additionally, the presentation of material to participants describes invariant generation in a testing context, and determining invariant correctness is separate from test input creation, as would likely occur if automated test input generation tools were applied.

Given this overall goal, we formulated several research questions (*RQ*). First, we considered the problem of classifying correct and incorrect invariants. When classifying and removing invariants,

Table 1: Case Study Artifact Information

	KAIST Case Study		
	StackAr	**Matrix**	**PolyFunction**
NCSS	35	135	150
# of Class Methods	9	23	22
Avg. Method Cycl. Compl.	1.89	5.70	6.64
Total # of Invariants	85	127	124
% of Correct Invariants	64.7%	81.9%	79.0%
% of Incorrect Invariants	35.3%	18.1%	21.0%
	KNU Case Study		
NCSS	35	122	110
# of Class Methods	9	21	18
Avg. Method Cycl. Compl.	1.89	5.62	5.89
Total # of Invariants	85	88	84
% of Correct Invariants	64.7%	79.5%	83.3%
% of Incorrect Invariants	35.3%	20.5%	16.7%

two types of errors exist: removing correct invariants and retaining incorrect invariants. As noted previously, both types of errors have negative consequences, but lead to different results. We wished to know the frequency of these two types of errors. Second, we considered which factors may contribute to mistakes by users. In the sole prior dynamic invariant generation user study, all case examples were relatively small data structures [23]. We were interested in whether the size and complexity of the case examples considered could impact the ability of users to correctly classify invariants. Finally, we considered how user effectiveness might vary across individual invariants. In particular, we were interested in whether some invariants were particularly hard to classify, and if so, why? Such insights would be valuable when considering how to improve the usability of dynamically generated invariants.

We thus explored the following research questions:

RQ_1 How effective are users at classifying correct and incorrect invariants?

RQ_2 How is user classification effectiveness influenced by program complexity/size?

RQ_3 How does the difficulty of classifying invariants vary?

3.1 Experiment Overview

To address our research questions we conducted two separate studies. The first study was conducted at the Korea Advanced Institute of Science and Technology (KAIST), in Daejeon, South Korea, with 11 graduate students, as part of an advanced software engineering course. The second study was conducted at Kyungpook National University (KNU) in Daegu, South Korea, with 20 undergraduate students, as an optional special seminar included within a software engineering course. The backgrounds of the participants varied considerably, ranging from those with several years of industrial experience to undergraduates.

Both studies involved the same basic steps. We prepared by applying Daikon to generate invariants for three case examples, and then determined the correctness of each generated invariant via random testing and manual examination. In each study, (1) we presented information related to invariants and the tasks to the participants, (2) we assigned each participant a single case example, after which (3) each participant manually generated program invariants (to instill an understanding of the program), and finally (4) each participant classified the automatically generated invariants.

The experiment setup and artifacts used for both studies are similar, but not identical due to differing time constraints (discussed in Section 3.5). In the remainder of this section, we note differences in the experiment procedure where they exist.

3.2 Java Case Examples

We used three Java case examples. *StackAr* is a stack class originally used in user studies of *Daikon* [23]. *Matrix* is a class representing a matrix, found in the JAMA linear algebra package developed by The MathWorks and the National Institute of Standard and Technology (NIST) [17]. *PolyFunction* is a class representing a polynomial function, and is part of the Math4J package [22]. Table 1 provides further details on these artifacts. Measurements given in the table were collected using JavaNCSS [21]. Non-Commented Source Statements (NCSS) measures roughly the number of statements ending in ";".[2]

We selected these particular case examples for several reasons. First, they are small enough (given some adjustments described in the following paragraph) that participants in pilot studies were able to first understand the case example, and then classify all invariants given to them in a reasonable period of time. Second, the *Matrix* and *PolyFunction* case examples are, in terms of number of lines and cyclomatic complexity, more complex than case examples used in prior studies. Finally, they differ in size and nature; *StackAr* is a relatively small, well-known data structure while the other two examples are larger and primarily algebraic computations.

In pilot studies, we found that the *Matrix* and *PolyFunction* case examples were too large to be used in a reasonable time frame. We therefore slightly reduced both case examples for use in our studies.[3] For the *Matrix* case example, where several similar functions exist (e.g., multiple versions of "minus" and "plus" operations), only one version was retained. Furthermore, mathematical operations that required a strong understanding of linear algebra (e.g. matrix factorization) were removed. For the *PolyFunction* case example, multiple constructors with different parameters (i.e., `long`, `int`, and `double`) were reduced to one constructor, and functions requiring a strong understanding of calculus were removed.

3.3 Applying Dynamic Invariant Generation

We used the *Daikon* invariant generation system in this study for several reasons. First, Daikon is the most mature and well documented freely available toolset, with a large body of related research [9, 10, 23, 26]. Second, Daikon supports Java, which is familiar to our study participants. Third, Daikon's approach has been adopted by commercial tools, specifically Agitator [3], and thus represents a tool likely to be used in practice. Finally, previous empirical work by several authors, including the only user study conducted using dynamic invariant generation, suggests that Daikon is an effective invariant generation tool [9, 23, 26].

Daikon requires test suites to generate invariants, and the selection of these suites represents an additional study factor. Previous work indicates that the test suites selected impact the quantity and quality of generated invariants [9, 23, 26]. Unfortunately, each additional test suite results in a different set of invariants to be considered. In these studies, to avoid potential learning effects (noted in [23]), we prefer that participants not perform the task multiple times, and our pool of participants is limited. We therefore used one test suite for each system.

There are several possible methods for selecting test inputs, including manual generation and several forms of automatic generation. In previous studies, existing test suites developed by the class authors were sometimes used [26], but unfortunately not all case examples selected have such test suites available. Rather than

[2]Raw data, including case examples, is available at `http://pswlab.kaist.ac.kr/data/`.

[3]The *Matrix* and *PolyFunction* case examples were reduced twice: once to fit within the time available in the KAIST study, and again to fit within the smaller time available in the KNU study.

manually generate such test suites and risk introducing our own biases into the study, we used automatic test input generation via the *Randoop* test input generation tool [24]. Randoop uses feedback-directed random test generation, and was selected because it is a mature test input generation tool for Java.

For each case example, we generated 1,000 random test inputs. We chose this number because the resulting test suites, when used with Daikon, yield a set of invariants such that the percentage of correct invariants is reasonable, but incomplete enough that incorrect invariants are still relatively common. In our studies, 64.7-83.3% of generated invariants are correct, which is within the range reported in previous empirical studies [26] when using actual systems. The number of invariants generated is listed in Table 1.

Daikon produces three types of invariants: method preconditions, method postconditions, and class invariants. In this study we removed method preconditions. Determining the correctness of method preconditions is problematic – a precondition conceptually represents "intended" method behavior, which a tester cannot possibly know. (Daikon is agnostic concerning "intended" behavior; users are not.) That said, simply instructing participants to assume preconditions were correct was also unsatisfactory, as several preconditions were clearly false and in pilot studies made understanding the postconditions (and the task itself) quite confusing.

Daikon can produce invariants in several formats. For this study, all generated invariants used the Java Modeling Language (JML) [5]. JML was selected for two reasons. First, the format is easy to understand, and uses mostly Java syntax. Second, there exist tools for JML – notably, it is possible to compile Java with JML extensions to detect violations of invariants during testing.

3.4 Determining Invariant Correctness

Once the invariants were generated, we needed to determine whether each invariant was correct or incorrect, in order to evaluate classifications made by each participant. Initially, we had hoped to use one of the many static analysis tools developed for determining the correctness of JML invariants in Java classes [4] (including more recent tools). Unfortunately, this was not possible: for every tool tested (roughly 10), the tool was almost completely undocumented, or did not support constructs used in the source or invariants, or did not compile/run at all.

Consequently, we could not prove invariants correct. Thus, we employed two automated approaches to try to falsify invariants. These approaches were conducted using the help of JML's invariant-aware compiler, allowing us to create test inputs (manually or automatically) that signal faults in the generated invariants. First, we applied Randoop using 100,000 test inputs (far more than the 1,000 used to generate the invariants). Second, a different, manually written random test generation harness was written for each case example, and then applied for a long period of time (24 hours). This manual harness was written to overcome some deficiencies we noted in Randoop's generation. For any remaining invariants, three of the authors manually examined each one, attempting to develop a test input capable of violating the invariant, or failing that trying to understand why the invariant was correct. Invariants that we could not falsify were accepted as correct. The percentages of correct and incorrect invariants are given in Table 1.

3.5 Experiment Procedure

Both studies began with the administration of a background questionnaire gathering information about participants' experience with invariants, GPA, and so forth. We then gave a classroom presentation (1) explaining the concept of dynamic invariant generation; (2) overviewing JML; (3) giving examples of automatic invariant generation, with explanations of what a correct and incorrect invariant is; (4) describing the tasks to be done; and (5) providing example tasks for the participants.

In the presentation, the motivation for invariant generation was explained to be regression testing. While invariant generation has been suggested as suitable for testing in general, it is possible to generate a correct invariant that, due to errors in the program, can be violated. This complicates a user study, as the user must be able to judge whether invariants match the intended program behavior. However, in regression testing users can assume the program is correct as given, and the user need only determine whether the invariants match the program's current behavior. We thus avoid questions of what the program does versus what the program "should" do; the program should do what it already does.

Following the presentation, each participant was assigned one of the three case examples at random, and given the tasks. Once the tasks were complete or the allotted time expired, exit surveys were administered, gathering information concerning participants' confidence in the task, their opinion of usefulness of generated invariants, and so forth.

In both studies the participants were asked to perform two tasks. First, they were given the case example without instrumentation, and asked to write five invariants in the source code. This was done to provide time in which they could study the entire Java class without examining generated invariants. Second, they were given the case example annotated with JML invariants, and asked to classify each invariant as *correct* or *incorrect* with respect to the source code. They were also given a list of special Java functions used by the generated JML invariants (extracted from Daikon's source code). They were encouraged to classify each invariant, but were allowed to leave invariants unclassified, for reasons of time or uncertainty. These unclassified invariants were categorized as "unknown".

Both tasks were conducted using paper printouts of the source code, rather than in front of computers. This was due to a lack of sufficient number of computers for the KNU study and a desire to maintain comparability between studies. Source code was formatted to print neatly for each case example.

Case study differences: When conducting the KAIST study, two 90 minute classroom sessions were available, while for the KNU study only one 90 minute classroom session was available. The KAIST study was conducted first, and using lessons learned we were able to pare down the experiment to require only one classroom session. The chief difference between the KAIST study is the length of the presentation given to students. The KAIST study was conducted as part of class material, and so a broader introduction to invariant generation and regression testing was given.

Other differences between the two studies are: (1) the presentation at KAIST was given in English for 90 minutes, while at KNU it was given in Korean for 30 minutes; (2) the first task was allotted 25 minutes at KAIST, but only 20 minutes at KNU; (3) the second task was allotted 60 minutes at KAIST, but only 35 minutes at KNU, (4) the KAIST study used larger case examples with more methods (and thus more invariants) than the KNU study; (5) the KAIST study placed a greater emphasis on applying invariant generation to regression testing than the KNU study.

3.6 Threats to Validity

External: In this study, our participants are students, most of whom are not professional software developers. In practice, however, developers vary in skill sets, and we find no reason that automatic invariant generation should be useful only to experienced developers. Furthermore, in our study, indicators of programmer

skill such as professional experience, education level, and GPA did not correlate with user effectiveness.

Our case examples are unfamiliar to the participants and time was limited. In practice, however, developers are often expected to maintain code that is not theirs and are typically time limited. Furthermore, our case examples were chosen to be simple enough to be quickly understood, and in pilot studies participants understood them in roughly half the time allotted. Additionally, participants were given time specifically to understand the code (20-25 minutes writing manual assertions). Additionally, the invariants were generated using test inputs from Randoop. In practice, user-written tests could be used, though the percentage of incorrect invariants we observed is similar to that reported in previous studies [26].

Each of the three case examples involves relatively simple Java classes chosen due to limited class time, though we expect the simple nature of the classes should ease the participants' tasks. Our current understanding of user classification effectiveness in this context is limited; thus we believe that establishing user effectiveness with small, easily understood programs, and basing later, larger studies on these results, is a prudent approach.

Internal: Our measurement of user effectiveness is obtained by comparing participants' classifications against our own classifications. As noted, while we can be sure incorrect invariants are incorrect, our own classifications for invariants believed to be correct are unprovable; thus, our measurements of user effectiveness may not be 100% accurate. Nevertheless several authors with unlimited time studied each invariant, and large amounts of random testing were performed for each system. Furthermore, the variance seen in the results (discussed in Section 4.3) indicates that even if our results are slightly incorrect, our conclusions are unlikely to change.

Construct: We have chosen to measure the usefulness of invariants in relation to the effectiveness of users in classifying them as correct or incorrect. Another option for measuring usefulness is to consider it in the context of specific software engineering tasks. Further studies could explore this avenue.

Conclusion: We have conducted two studies with 11 and 19 students. Each case example was thus given to 3-7 students, depending on the study. This is a relatively small number of users. However, for the KNU study statistical power was sufficient to show an effect between case examples,[4] and the number of classifications made by each user (85-127 invariants each) makes it easy for patterns in invariant difficulty to be visible. All statistical methods we employ are non-parametric. Thus conclusions we make concerning the existence of relationships between variables are sound, relying on few assumptions.

4. RESULTS

For each user's assessment of an invariant, the invariant is either correct or incorrect, and the user may judge the invariant to be either correct, incorrect, or unknown. Thus for each assessment of an invariant, six basic results can occur. For brevity, we denote each classification as $I_X Y$, where I is an invariant of correctness X (true or false) classified as Y (true, false or unknown). For each user, we present the percentage of invariants that fall into each category, for each case example / study pairing, in Figure 1. Percentages are normalized for correct and incorrect invariants.[5] The results are partitioned between correct and incorrect generated invariants. These figures capture the effectiveness of users at classifying generated invariants. Note that only two categories represent correct judgements by the users: $I_T T$ and $I_F F$ classifications.

Table 2: Percentages of Invariants Correctly Classified by Participants

KAIST Study					
	Correct Invariants				
	Min	**Mean**	**Median**	**Max**	**Std. Dev.**
StackAr	89.0%	90.9%	90.9%	92.7%	1.48
Matrix	48.0%	75.9%	89.4%	96.1%	18.7
PolyFunction	63.2%	71.4%	74.4%	79.5%	6.16
	Incorrect Invariants				
StackAr	30.0%	55.5%	40.0%	96.6%	29.3
Matrix	60.8%	73.9%	73.9%	91.3%	11.0
PolyFunction	23.0%	46.1%	53.8%	57.6%	13.5
KNU Study					
	Correct Invariants				
	Min	**Mean**	**Median**	**Max**	**Std. Dev.**
StackAr	45.4%	74.5%	74.5%	96.3%	17.0
Matrix	40.0%	69.0%	71.4%	95.7%	17.3
PolyFunction	31.4%	68.3%	75.7%	94.2%	20.5
	Incorrect Invariants				
StackAr	20.0%	41.4%	36.6%	83.3%	18.2
Matrix	50.0%	60.1%	66.6%	66.6%	6.74
PolyFunction	7.14%	54.7%	78.5%	85.7%	34.4

In Table 2, we present user classification effectiveness as measured by the percentage of invariants correctly classified, listing mean, median, minimum and maximum user effectiveness, along with the standard deviation for user effectiveness (in percentage points). Values are divided by study, case example, and invariant correctness. We discuss our results in the context of our research questions in the remainder of this section.[6]

4.1 RQ1: User Effectiveness for Correct and Incorrect Invariants

As Figure 1 shows, both correct and incorrect invariants are frequently misclassifed by users. For the KAIST study, average user effectiveness for invariants ranged from 71.4%-90.9% and 41.6%-73.9% for correct and incorrect invariants, respectively. For the KNU study, average user effectiveness ranged from 68.3%-74.5% and 41.4%-60.1% for correct and incorrect invariants, respectively.

In both studies and for each case example, users seem to do better at classifying correct invariants. This matches our expectations; we believed that users, presented with automatically generated invariants, would either mentally generate a correct counterexample, or accept the invariant. To test this, we proposed the following hypothesis and null hypothesis:

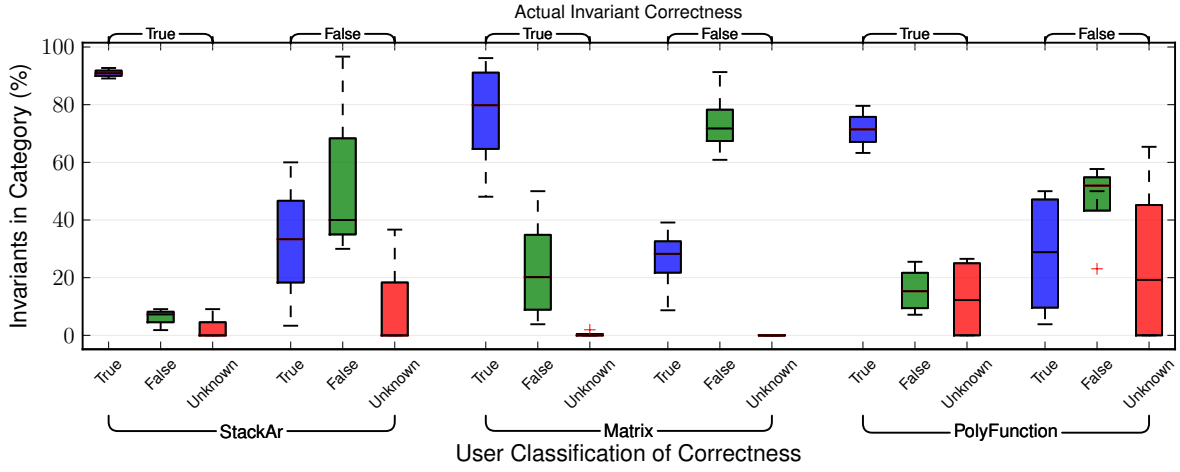
H_1**:** Users are more effective at classifying correct invariants than incorrect invariants.

H_{1_0}**:** The percentage of correct classifications by users for incorrect and correct invariants are drawn from the same distribution.

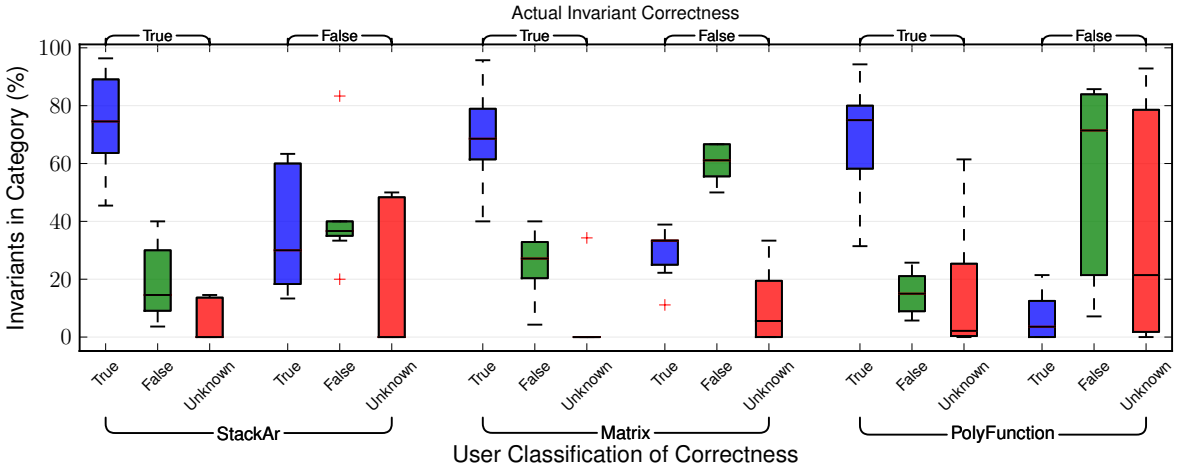
[4] 20 subjects were used for the KNU study, with at least 6 per case example. Using a permutation test and case examples X and Y with 6 subjects each, we have 12 choose 6 = 924 possible resamplings. If the test statistic for the original population $<=$ the test statistic for 5% or less of the resampled populations (46), there is a statistically significant difference at $\alpha = 0.05$.

[5] In the figure, each box represents the values falling between the first and third quartiles. All data points greater or less than the median by 1.5 * IQR (interquartile range) are marked as outliers (plus signs). The star represents the mean and the line in the box represents the median.

[6] In our statistical analyses, we do not consider *unknown* classifications. Statistics such as user effectiveness and invariant difficulty are based on firm answers by the user. Note that while some users failed to classify several invariants, most classified the vast majority of invariants. This can be seen in the low median for *unknown* classifications (0.0% - 13.7%).



(a) KAIST Study



(b) KNU Study

Figure 1: Classification of invariants by users

H_{1_0} states that user effectiveness for correct and incorrect invariants are drawn from the same distribution; i.e., users are equally likely to make $I_T T$ and $I_F F$ classifications. To test this hypothesis, we can use a non-parametric test for evaluating whether two sets of numbers are drawn from the same population. In this case, each user provides the percentage of correct and incorrect invariants correctly classified, and therefore the data for evaluating this hypothesis is paired, with 11 pairs of data for the KAIST study and 19 pairs of data for the KNU study.[7] We applied the two-tailed paired permutation test for each study [20], resulting in a p-value of 0.038 for the KAIST study and 0.004 for the KNU study. In both cases, we reject the null hypothesis at $\alpha = 0.05$. Further, given the higher than average user effectiveness for correct invariants observed for all case examples, we conclude that H_1 is supported.

We were surprised at the number of $I_T F$ classifications in both studies. We expected that users would rarely, if ever, generate incorrect counterexamples, and would therefore rarely reject correct invariants. Thus with respect to RQ_1, we find that while our hypothesis is supported, users appeared to struggle with both recognizing invariants as correct, and identifying incorrect invariants (as we expected). We discuss implications of this in Section 6.

[7]One participant in the KNU study given the *Matrix* case example declined to complete the study. His answers were not included in the data analysis.

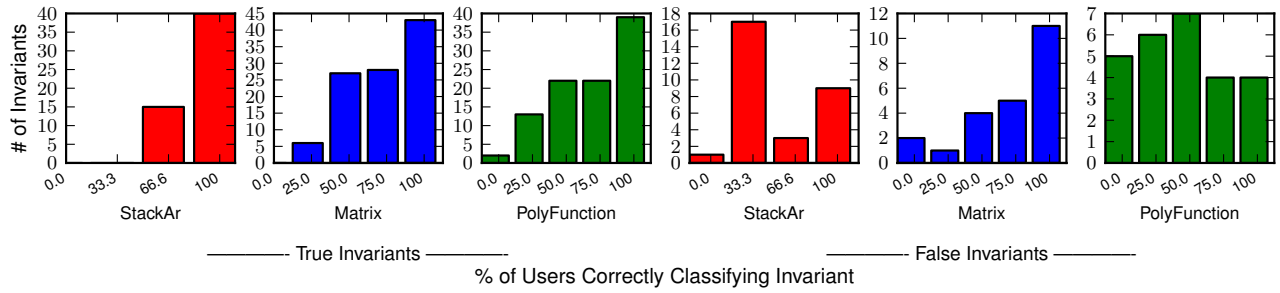
4.2 RQ2: Impact of Program Complexity on User Effectiveness

In Table 2, we can see from the mean and median user effectiveness that differences exist in user effectiveness between case examples, particularly for the KAIST study. However, as indicated in Figure 1 and Table 2, the differences between the minimum and maximum user effectiveness *within* a case example (and even the interquartile ranges) are often quite large – larger than the differences between case example means and medians. This is particularly true for incorrect invariants, which have standard deviations of 6.74 to 34.4 percentage points.

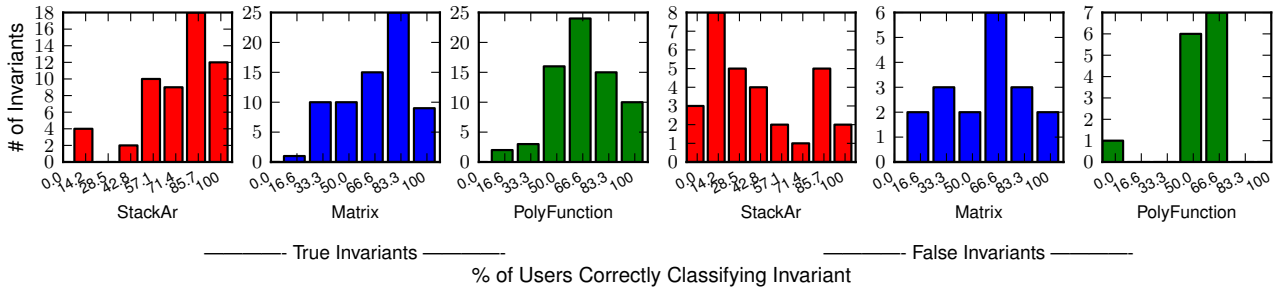
To further study the impact of program complexity, we again employed statistical hypothesis testing. The number of users was insufficient in the KAIST study to conduct statistical analysis between case examples (3-4 students per case example). However, the number of users was sufficient in the KNU study to conduct hypothesis testing (5-6 students per case example). Accordingly, we formulated the following hypothesis and null hypothesis:

H_2**:** Users are more effective at classifying invariants for smaller, less complex programs than for larger, more complex ones.

H_{2_0}**:** The percentage of correct classifications for two case examples X and Y (possibly of different size) are drawn from the same distribution.



(a) KAIST Study



(b) KNU Study

Figure 2: Invariant difficulty

We evaluate this hypothesis for each pair of case examples for the KNU study. We again use a permutation test (non-paired, in this case), resulting in the p-values shown in Table 3. As the values show, we cannot reject the null hypothesis at the $\alpha = 0.05$ level for any pair of case examples. We therefore conclude that, for the KNU study, we do not have sufficient evidence to conclude that H_2 is supported. Furthermore, while we did not have enough users to conduct statistical hypothesis testing in the KAIST study (insufficient power), the large spread in user effectiveness within each case example also indicates that H_2 is not well supported.

Table 3: H_{2_0} p-values

	Stack / Poly	Stack / Matrix	Matrix / Poly
True Invariants	0.59	0.59	0.98
False Invariants	0.42	0.06	0.65

We thus conclude that H_2 is not supported. With respect to RQ_2, the explanatory value of program structure is — contrary to our initial belief — not very strong; thus, there must exist other factors that more strongly impact the effectiveness of users in classifying invariants. We discuss our search for such factors in Section 5.

4.3 RQ3: Invariant Difficulty Variance

Figure 2 presents the percentage of user judgements that we believe are correct, divided between correct and incorrect invariants, for all case examples and studies. The right-hand side of each subfigure represents the invariants that all users correctly classify (100%), and the left-hand side represents invariants that all users incorrectly classify (0%). Based on our own experiences, we expected that invariants would tend to be either obviously correct or incorrect and thus very easy to classify, or very difficult to classify and thus misclassified by most users. In other words, we expected a bi-modal distribution in each subfigure, with most invariants classified correctly by almost all users (near 100%, "easy") or very few (near 0%, 'hard"), and few in between.

The figures show, however, that our expectation was not met for any combination of study, case example, or invariant correctness. While a large number of invariants were classified correctly by all users — sometimes more than 50% of invariants — particularly in the KAIST study, there is a long, uneven drop to 0%. Furthermore, invariants that no users correctly classified are relatively rare: no more than six invariants in any case example / study.

With respect to RQ_3, we see that there exists a large middle ground in invariant difficulty between very easy and very hard. Accordingly, it does not appear that there is some specific subset of generated invariants that all users struggle with; we cannot solve the problem simply by ceasing to generate invariants of troublesome type X. We must instead identify potential factors related to failure, and attempt to model their effect on user effectiveness.

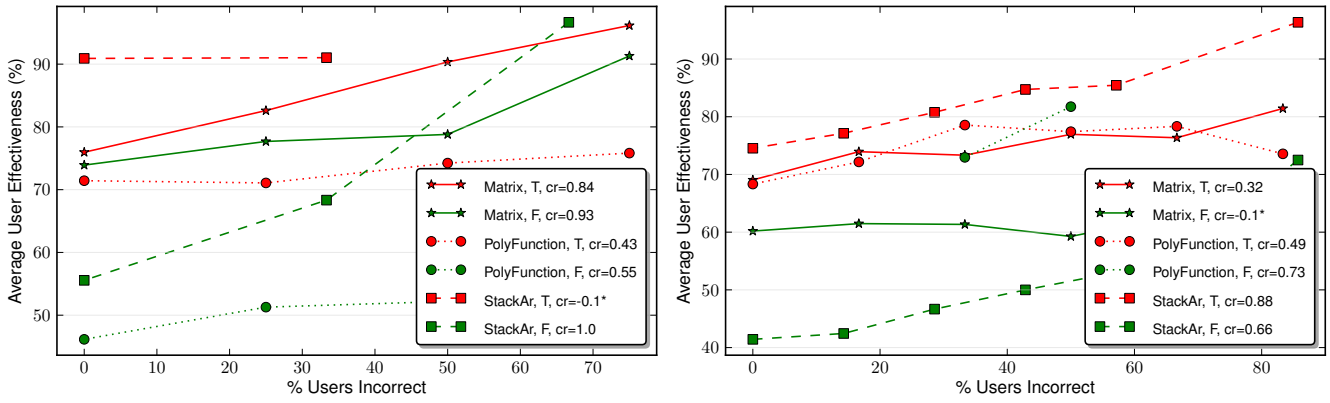
5. DISCUSSION

In our two studies, we find the results differ in degree — the KAIST participants tend to outperform the KNU participants and have less variation between users, for example — but contain similar patterns. We therefore draw the same two core conclusions for both studies. First, contrary to the implications of previous studies, users struggle to correctly classify invariants (RQ_1). Depending on the case example, users on average misclassified up to 58.6% of incorrect invariants, and misclassified up to 31.7% of correct invariants. Second, it is unclear why users struggle – we noted that the difficulty of classifying invariants varies (RQ_3), and our original hypothesis related to program complexity is unsupported (RQ_2).

Thus we need to improve our ability to apply invariant generation, but it is unclear how. In this section, we examine other factors potentially related to user effectiveness and invariant difficulty. In the next section, we discuss the implications of our results.

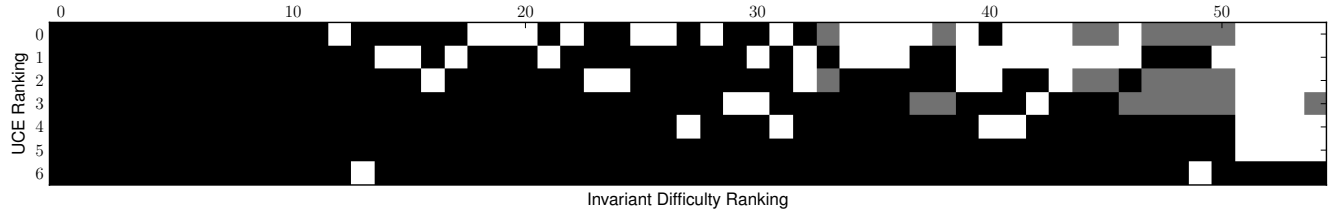
5.1 Establishing a Pattern of Difficulty

One of the primary goals in this study is to identify factors related to user effectiveness. However, we would first like to establish that such factors exist. To do this, we begin by determining whether a relationship exists between user effectiveness and the difficulty of classifying invariants (as measured by the percentage of users misclassifying them). In other words, we would like to know whether "hard" and "easy" invariants actually exist, and whether "good"

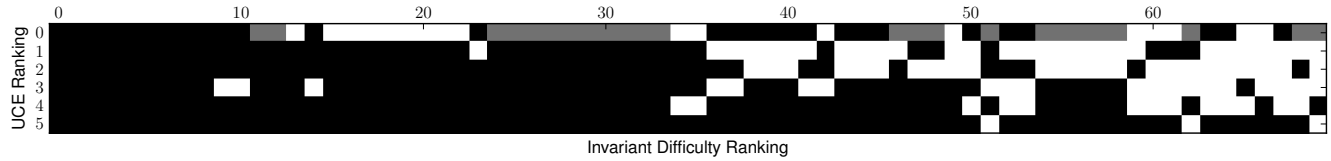


(a) KAIST Study (b) KNU Study

Figure 3: Relationship between user effectiveness and invariant difficulty (* = not statistically significant)



(a) KNU Study, StackAr Case Example (Correlation 0.88)



(b) KNU Study, Matrix Case Example (Correlation 0.32)

Figure 4: User Effectiveness Matrix. 0 = Easiest invariant / Least effective user. UCE = User Classification Effectiveness.

users are those who can classify "hard" invariants. In the absence of such a relationship, we must conclude that differences in user effectiveness are not invariant-specific, and we will probably not be able to identify factors related to user effectiveness.[8]

In other words, we would like to establish whether the percentage of users misclassifying an invariant is positively correlated with the average overall effectiveness of users who correctly classify the invariant. We thus formed the following hypothesis:

H_3 The correlation between the percentage of users misclassifying an invariant I, and the average overall effectiveness of all users detecting I, is positive.

To test this hypothesis, we measured this correlation (using the Spearman rank correlation adjusted for ties [20], a non-parametric measure of correlation) for all combinations of studies / examples / invariant correctness, resulting in Figure 3. As the figure shows, the correlation between invariant difficulty and average user effectiveness is positive for most combinations, with statistical significance at $\alpha = 0.05$ (correlations not statistically significant are labelled with a *). Indeed, the correlation tends be quite high, with statistically significant correlations ranging from 0.32, indicating low correlation, to 1.0, indicating perfect correlation, with most correlations near or above 0.5, indicating moderate correlation [16].

[8]Consider a game in which players memorize lists of numbers. We expect that while no number is easier to remember, some individuals have better memories. Similarly, users may vary in their ability to understand invariants, without invariants varying in difficulty.

To illustrate the implications of this, in Figure 4 we have plotted a matrix representing the invariants classified by each user for two case examples in the KNU study (both using true invariants). Black squares represent correct classifications, white squares represent misclassifications, and grey squares represent "unknown" (i.e., non-classifications). We have ranked the invariants by difficulty and ranked the users by average effectiveness.

In this figure, higher numbers represent more effective users and more difficult invariants. For example, for the *StackAr* case example, for the 50th most difficult invariant, participants 4-6 (the best participants) correctly classify the invariant, while participant 1 misclassifies it and participants 0 and 2-3 do not classify it; only the most effective participants can correctly classify the invariant. Conversely, for the 20th most difficult invariant, only the least effective participant cannot correctly classify the invariant.

As the figure shows, for the *StackAr* case example (correlation of 0.88) relatively effective participants consistently are those that correctly classify difficult invariants. This indicates that there is some factor related to these invariants that makes them consistently challenging to users. However, for the *Matrix* case example (correlation 0.32) the pattern is less clear. Participants who are relatively ineffective sometimes correctly classify apparently challenging invariants, while participants who are relatively effective sometimes fail to classify apparently easy invariants. In this instance, it appears that the differences between user effectiveness are less related to the actual invariants than in other instances. Other case examples / study pairings have behavior between these two extremes.

```
@ ensures (this.topOfStack == -1)  <==>  (\result == null);
@ ensures (this.topOfStack >= 0)  <==>  (\result != null);

  public Object top( )
  {
    if( isEmpty( ) )
      return null;
    return theArray[ topOfStack ];
  }
```

Figure 5: StackAr Pattern 1 Incorrect Invariants Example

5.2 Statistical Analysis

With our pattern established, we began our search for relevant factors by applying several statistical analyses. First, we measured the Spearman correlation between user effectiveness, invariant difficulty, and several metrics related to invariant complexity [20]. The metrics include: the existence of each operator (e.g., implication, and, plus, etc.); the number of operators used (i.e., the size of the invariant); the number of unique operators used; the number of Daikon-specific methods used; the number of unique methods used; the number of variables referenced; the number of class variables referenced. Additionally, several metrics related to the participants as drawn from the user entry and exit surveys were also applied, including user GPA, years of programming experience, and confidence concerning invariant understanding (estimated by the participant). Correlations were measured at all possible levels, ranging from the aggregate results across all participants, down to the level of individual case examples from separate studies subdivided between correct and incorrect invariants.

In all cases, no correlation was observed to be higher than 0.3, and no correlations were consistent across studies or examples. This indicates that no strong relationship between a single factor and user effectiveness exists. We then attempted to find a relationship between combinations of factors and user effectiveness, applying both linear regression (for all exploratory variables) and logistic regression (for binary exploratory variables), again at all possible levels. Again, these techniques yielded no strong predictive relationship.

5.3 Manual Analysis

Following our initial analysis, we conducted a manual examination of each invariant to try to identify qualitative reasons users misclassified invariants. In particular, we were interested in incorrect generated invariants that a high percentage of users (>80%) tended to misclassify. Each author independently conducted this task, and several patterns were identified as occurring for such invariants.

Pattern 1: Non-exclusive use of a specific return value for signalling exceptional states. For some methods, exceptional states are signalled by returning specific values (e.g., 0, -1, `null`) instead of using Java exceptions. However, it is sometimes possible to return this value in non-exceptional (but often rare) circumstances. This can result in invariants being generated that initially appear correct, but can be violated.

For example, consider the code fragment from the `StackAr` case example shown in Figure 5. In this method, `null` is used to indicate that a usage error has occurred: reading the top of an empty stack. This behavior is captured by Daikon, resulting in the invariants listed (when `topOfStack` is -1, the stack should be empty). However, it is also possible to directly push a `null` value onto the stack, and later read it, thus violating these invariants.

Pattern 2: Direct, unchecked manipulation of class variable values. Some classes allow class member variables to be directly manipulated by `set` methods. Such methods remove class enforcement of data constraints in exchange for improvements in performance, with the intent that the user externally enforce such constraints. As with *Pattern 1*, this can result in the generation of invariants that capture the intended class invariants, but that can be violated through use of direct manipulation of class member variables via `set` methods.

```
/*@ ensures pairwiseEqual(this.A, A);  */

  public Matrix (double[][] A, int m, int n) {
    this.A = A;
    this.m = m;
    this.n = n;
  }
```

Figure 6: StackAr Pattern 3 Incorrect Invariants Example

For example, in the `Matrix` example, many of the invariants generated by Daikon capture the intended relationship between the array size, and the number of columns and rows in the matrix. However, several methods allow the user to directly set internal state variables without enforcing consistency checks for array size and row and column sizes. Such methods, when used in the right sequence — i.e., directly before another method that has invariants capturing array/column/row invariants — and with the right inputs, can inviolate many of these invariants. In both studies, for most (> 80%) of the *Matrix* invariants fitting this pattern, users correctly noted that the invariants were incorrect, but for a handful of instances users did overlook such possibilities.

Pattern 3: Potentially unexpected behavior of invariant utility functions. As noted in Section 3, participants were given copies of the functions defined by Daikon to define invariants. Most of these functions are straightforward, capturing, for example, the ordering of elements or equality of all the elements. However, many of the methods define comparisons against a `null` object — even equality — as always `false`. Without a careful examination of the functions, this behavior can be overlooked; this can be particularly troublesome for invariants that seem trivially true.

For example, consider the example invariant drawn from the *Matrix* example in Figure 6. This invariant captures an obvious property of this constructor: the elements of `A` are the same elements of `this.A`, as both point to the same object. However, `pairwiseEqual(null,null)` is defined as false, and thus when calling `Matrix(null, 0, 0)` — a nonsensical, but valid input — we violate this invariant.

Based on this analysis, we can see that for these studies, the invariants users consistently misclassify are those related to unexpected behaviors. These include unusual function semantics, in the case of the Daikon-specific methods, unusual (or poorly designed) exception signalling, and unexpected changes to member variables. Generalizing, we believe that in such scenarios, the user's preconceptions are violated. In the case of unusual invariant semantics, the user believes the invariant correct because their understanding of it is flawed. In other cases, the user's own preconceptions concerning how the program *should* work align with the invariant inference tool's conclusions, but corner cases violate both.

We can therefore infer that when using dynamic invariant generation, special care should be taken to (1) use easily understood invariant semantics, and (2) be aware of corner-cases in the applications where the generation process may infer reasonable, possibly intended or desirable properties that can be violated in certain circumstances. Unfortunately, while such guidelines may capture the worst offenders, as noted in Section 4.3, most invariants are not so universally difficult. In the next section, we discuss the implications of this for future work.

6. IMPLICATIONS

These studies were conducted to evaluate one crucial aspect of dynamic invariant generation: the ability of users to classify gen-

erated invariants. The results indicate that dynamic invariant generation in its current form requires refinement. In this section, we outline the implications of our results for future work.

While our study has been conducted from the perspective of software testing, the aspect studied is applicable to any context in which users must understand the the results of dynamic invariant generation: namely, any context in which dynamic invariant generation aids software verification. Thus while our results are framed in the context of testing, we believe they apply to other contexts.

6.1 Crack the Code

In both studies, we found a surprisingly high level of variance in terms of invariant difficulty. We originally believed that invariants would be either easy or difficult to classify, but in practice, for most case examples, the majority of invariants fall into the middle ground, with a moderate percentage of users (20-80%) misclassifying the invariants. This is unfortunate, because while we have identified a small class of invariants that consistently cause problems, there is no easy "off switch" fix for all invariants.

Given this, ideally, this study should result in a model of invariant difficulty: an explanation for why and when users find the task challenging. Such a model could be used to help users to focus efforts on tricky to classify invariants, or operate as a blueprint for future improvements to the technique. Unfortunately, our statistical analyses did not yield such a model, and our manual analysis indicated only a couple of scenarios that led to consistently misclassified invariants. While our studies have demonstrated the need for improvements, we have not "cracked the code" concerning *why* using existing techniques is challenging.

In Section 4.3, however, we did note that invariants can often be ordered by difficulty, with difficulty related to user effectiveness. We believe this relationship is encouraging. While we have not been able to identify why some invariants are more difficult than others, the fact that the relationship exists suggests that future work, possibly with larger sample sizes, may uncover why.

6.2 Generation with Explanation and Better Invariant Filtering

Our results indicate that users frequently misclassify both correct and incorrect invariants. Irrespective of why users make mistakes, improvements to help them understand correct invariants, and do a better job removing incorrect invariants (ideas suggested, but not empirically supported, in [3] and [31]) are warranted.

Currently, dynamic invariant generation does not provide a user-readable argument for why each invariant has been generated. Invariant generation is entirely a black-box process, taking program traces as input and producing program invariants as output. However, if invariant generation were to provide a method for better understanding why each invariant was generated, the user could potentially (1) more quickly understand why an invariant is correct, thus preventing them from trying to falsify it, and (2) identify shortcomings in the argument, thus improving the process of identifying incorrect invariants. Daikon already (necessarily) internally forms such an argument (as must other inference engines); it may be possible to translate this argument into something human readable. Other options include developing methods of summarizing and presenting paths taken during invariant generation and highlighting shortcomings in the submitted traces. (Such work could borrow from existing work on summarizing test traces and counterexamples [15].)

Additionally, the current approach to invalidating invariants in Daikon (and other tools) can be extended to use more powerful techniques. Recall that dynamic invariant generation relies on data traces generated from the program of interest, and thus a variety of potentially useful sources of information and analysis techniques are left unused. These include: program coverage information (e.g., methods covered, paths explored); static analysis techniques, particularly those capable of falsifying program assertions [13]; and dynamic analysis techniques, such as symbolic/concolic execution [27]. By applying such techniques, we may reduce the number of incorrect invariants generated.

6.3 Quantify the Impact of User Error

As noted in Section 4.1, there are two potential types of invariant classification errors, $I_T F$ and $I_F T$. The first type of error is potentially self-correcting for cautious users: when the user attempts to demonstrate the invariant can be violated, they will find the invariant is actually correct. Nevertheless, such errors are still a problem, as they cost the user time when applying the tool. The second type of error is more serious in the context of testing; incorrect invariants will be used, potentially leading to false positives later in the testing process, possibly much later. These false positives may lead to several negative outcomes, including: mistaking false positives for true positives, causing false "errors" to be corrected; loss of time in determining that an error is in fact a false positive, a potentially difficult task for any testers not intimately involved in the development or invariant generation process; and loss of trust in invariants generated for testing, leading to discontinuation of tool use.

In these studies, we have found that both of these types of errors occur frequently: on average, 9.1%-31.7% of correct and 26.1%-58.6% of incorrect generated invariants are misclassified. Unfortunately, while we can broadly categorize the impact of user error — slowdowns in the process of generating invariants, false positives in testing, lost of trust, etc. — this impact has not yet been quantified in this study or, to the best of our knowledge, any other.[9]

We believe that understanding this is key to understanding the practicality of dynamic invariant generation. While prior studies have mechanically quantified the benefits of invariant generation, this work casts doubt on the ability of users to make practical use of the invariants. In future work, understanding how user error negatively impacts the testing process (the time required to determine whether invariants are correct, the time required to fix false positives in testing) is necessary to understand whether dynamic invariant generation is a cost effective method of oracle generation.

7. CONCLUSIONS

In this work, we have conducted two studies to evaluate one crucial aspect of dynamic invariant generation: the ability of users to understand and correctly classify generated invariants. Our results indicate that users frequently make mistakes. These results run contrary to previous work, and call into question the ability of users to effectively use generated invariants.

Further analysis indicates that while some patterns in user effectiveness and the difficulty of classifying invariants do exist, the relationship between invariant difficulty and other factors is subtle. Factors such as program complexity, invariant size, the number of class variables referenced, etc., are not strongly correlated. However, manual analysis did indicate that certain corner cases related to unusual method semantics and poor exception handling can result in tricky invariants misclassified by nearly all users.

Based on these results, we have made several suggestions for future work that we believe may help bridge the gap between dynamic invariant generation techniques and user ability, including: refining our understanding of invariant complexity, likely through

[9]Nor has it been quantified in related areas, such as the application of static analysis tools, where similar issues can arise.

additional studies; extending existing techniques to support users in understanding why invariants are correct and reducing the number of incorrect invariants generated; and quantifying the impact of user error on the effectiveness of invariant generation in practice.

Testing research has long focused on test input generation, with most evaluations largely automated. As we move towards supporting users in the entire testing process, including test oracles, we must be aware of the limitations of users. Evaluations must quantify not only the potential effectiveness of the test oracles generated, but also the ability of users to apply them. We believe this study represents an initial step towards this for dynamic invariant generation.

8. ACKNOWLEDGEMENTS

We thank Yunja Choi for discussions concerning experimental design, and for allowing us to conduct the experiment at Kyungpook National University. We also thank the participating students at KAIST and Kyungpook National University. This work is supported in part by: the World Class University program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007); the National Science Foundation through awards CNS-0720757 and by the Air Force Office of Scientific Research through award FA9550-10-1-0406; the ERC of MEST/NRF of Korea (Grant 2012-0000473).

9. REFERENCES

[1] L. Baresi and M. Young. Test oracles. In *Tech. Report CIS-TR-01-02, Dept. of CIS, U. of Oregon*, 2001.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *Int'l Symp. Softw. Test. Analy.*, pages 169–180, 2006.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *Int'l J. Softw. Tools Tech. Transfer*, 7(3):212–232, 2005.

[5] Y. Cheon and G. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Int'l Conf. on Softw. Eng. Res. Pract.*, pages 322–328, 2002.

[6] D. Cok and J. Kiniry. ESC/java2: Uniting ESC/Java and JML. *Constr. Analy. Safe, Secure, Interop. Smart Devs.*, pages 108–128, 2005.

[7] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Int'l Conf. Softw. Eng.*, pages 281–290, 2008.

[8] J. Edvardsson. A survey on automatic test data generation. In *Conf. Comp. Sci. Eng.*, pages 21–28, 1999.

[9] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Comp. Prog.*, 69(1-3):35–45, 2007.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Trans. Softw. Eng.*, 27(2):99–123, Feb. 2001.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Int'l. Conf. Softw. Eng.*, pages 449–458, June 2000.

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Int'l Symp. Softw. Test. Analy.*, pages 147–158, 2010.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Softw. Eng. Notes*, 24(6):146–162, Nov. 1999.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. *Prog. Lang. Des. Impl.*, 2005.

[15] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *Comp. Aided Verif.*, pages 318–321, 2004.

[16] J. Guilford and B. Fruchter. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, New York, 1956.

[17] J. Hicklin, C. Moler, P. Webb, R. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A Java matrix package, 2005. URL: http://math.nist.gov/javanumerics/jama.

[18] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Softw. Eng. J.*, 11(5):299–306, 1996.

[19] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Int'l. Conf. Softw. Maint.*, pages 736–743, Nov. 2001.

[20] P. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley, 2007.

[21] C. Lee. JavaNCSS-a source measurement suite for Java, 2005. URL: http://javancss.codehaus.org/.

[22] Math4J: A Java numerics package, 2005. URL: http://math4j.com/.

[23] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Int'l Symp. Softw. Test. Analy.*, pages 232–242, July 2002.

[24] C. Pacheco and M. Ernst. Randoop: Feedback-directed random testing for Java. In *O.O. Prog. Sys. Apps.*, pages 815–816, 2007.

[25] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Symp. Found. Softw. Eng.*, pages 23–32, Nov. 2004.

[26] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Int'l Symp. Softw. Test. Analy.*, pages 93–104, 2009.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Int'l. Symp. Found. Softw. Eng.*, pages 263–272, 2005.

[28] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Int'l Conf. Softw. Eng.*, 2012.

[29] M. Staats, M. Whalen, and M. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Int'l Conf. Softw. Eng.*, pages 391–400. IEEE, 2011.

[30] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. *Formal Meth. Softw. Eng.*, pages 717–736, 2006.

[31] Y. Wei, C. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Int'l Conf. Softw. Eng.*, pages 191–200, 2011.

[32] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Auto. Softw. Eng.*, 2003.