On the Automatic Detection of Loop Invariants

Nicholas Deibel

February 25, 2002

Abstract

Loop invariants play a pivotal role in software verification. Since not all programs and the loops therein are annotated with their invariants, systems for automatically extracting loop invariants have been developed. In this paper, we will discuss the problem of finding loop invariants and the various methods that have been used for this problem. We will also propose potential avenues for future research in this area.

1 Introduction

Dijkstra once stated, "Testing can only show the presence of bugs, not the absence." This quote highlights the inherent difficulty of software verification. Simply throwing one test suite after another at a program might increase one's belief that a program is error-free, but it will never form a conclusive proof. The only exception is if the test suites completely cover all possible inputs. This is often too costly or outright impossible.

Instead, proving software correct is done (to some degree) automatically. In general, proving software correct is an undecidable problem (does a program halt is a potential correctness question one could ask). In an ideal situation, the verification process is semi-automatic. The programmer annotates the code with what he or she assumes to be correct method specifications and loop invariants. A program involving a theorem prover then checks these annotations. While this method is sound and works, the actual annotation process makes it impractical. One, the process of annotating code is tedious and time-consuming. Most programmers are hesitant to spend their time on such a task. In fact, most managers, for purely economical reasons, would prefer their employees to be coding rather than adding specialized comments to code. Another downside to annotation is how does one handle legacy code? Annotating unfamiliar code will certainly pose a greater challenge and will take time.

With this in mind, there is an increasing interest in discovering loop invariants, pre-conditions, post-conditions, etc. automatically. Various methods have been developed and applied for this task with varying levels of success. Interestingly enough, there has been a serious gap in the research of detecting loop invariants. Significant work was done in the 1970's for compiler technology. In order to optimize efficiency, compilers wanted to identify loop invariant code and move it outside of loop bodies. At this time, theorem provers and artificial intelligence work was still relatively young. Significant strides have been made in these areas, but it is not clear to what degree, if any, these advancements have been applied to system verification.

In this paper, we will focus on the methods, both past and current, used for detecting and verifying loop invariants. In Section 2, we will discuss important background material regarding this subject. Sections 3-6 will each cover an invariant-finding process that has shown success in practice. While this represents only a subset of the methods (currently and in the past) explored, we hope to show that successful detection can be achieved through noticeable different techniques. Finally, in section 7, we will propose ideas more akin to artificial intelligence search methods.

2 Core Concepts of Loop Invariant Detection

A *loop invariant* for a loop in a program is a proposition composed of variables from the program that is true before the loop, during each iteration of the loop, and after the loop completes (if it completes). For example, in figure 1, one can show that the statement (t >=0) is an invariant for the loop. Similarly, the statement $(i = div * j + t)$ is also an invariant. Intrinsically, one should see that the latter invariant delivers more knowledge about the loop. Neither invariant alone, though, is enough to prove the correctness of this simple division algorithm; both are required.

```
// PRE: i>0 && j>0
t := i
div := 0
while(t >= j) do
  div += 1
  t -= j
end
// POST: i=div*j+t && 0<=t<j
```

Figure 1: A simple loop program

To show that both invariants must be included, we will first introduce a formal logic system known as Hoare logic. Developed in 1969 by Hoare for proving programs correct, statements in this logic are of the form {P} S {Q}, where P and Q are predicates and S is a program. This statement reads as if P is true, then after the execution of S, Q will be true. A loop in this logic is written as: {P} while C do S {Q}, where C is the loop conditional (which we will sometimes refer to as the loop's guard). In Hoare logic, the three requirements of a loop invariant I are

```
P → I               // I is true when the loop starts
{C && I} S {I}      // I is true after a loop iteration
(~C && I )→ Q       // If the loop finishes, Q is true
```

Figure 2. The requirements of a loop invariant I for the Hoare statement: {P} while C do S {Q}

shown in figure 2. Under these rules, one should see that the proper loop invariant for the code in figure 1 must e $I = \{ t >= 0 \ \ \&\& \ \ i = div * j + t \}$.

This example, while simple, properly illustrates the challenges facing any loop invariant detector. The shear enormity of possible loop invariants to consider is enormous. Even if we limit ourselves to working with integers, a proposition like {t > ? } has infinitely possible settings. Furthermore, there might be variables in the code not required in the invariant. Plugging and checking every possible invariant is severely not feasible. We need, in general, an algorithm that easily cut its way through this large solution space. This is unfortunately impossible.

Blass and Gurevich, recently proved, [1], that as a consequence of Cook's completeness theorem there exists a program whose loop invariants are undecidable. Specifically, letting *N* be the set of natural numbers as well as the functions S(x)=x+1, D(x)=2x, and H(x)=x/2 (integer division), then there exists a program S with a single loop using the three variables x, y, and z such that $\{ x = y = z = 0 \} S \{ false \}$is correct in *N* but any proof uses an undecidable loop invariant. Their proof uses an interesting (in at least a theoretical aspect) reduction involving recursive functions on strings.

In one light, this is a very serious result. The program involved in this proof is simple in that it only uses three variables. Furthermore, the structure these programs work on is very simple. It only involves integers and the basic functions of successorship, doubling, and halving. With these two points, it is reasonable to surmise that there exist more complex programs whose loop invariants will also be undecidable.

These results, though, should be viewed in much the same way as an NP-completeness proof often is. While it is true that we will never have a perfect loop invariant finder for any program, this does not rule out the possibility of a large number of decidable invariants. In particular, consider the program considered in [1]. While the precondition is rather commonplace, the postcondition is strictly false, a situation not likely to occur in practice. True implies false is the "odd man" of conditional logic and often leads to some awkward situations in proofs. The undecidability of this particular loop invariant might just be a strange consequence of having a strictly false precondition. Without further evidence, we should probably view this is an extremely pathological case that is unlikely to occur in practice.

3 The Induction-Iteration Method

In the previous section, we saw the inherent difficulty in finding proper loop invariants. Thus, it comes as a surprising fact that there exists a purely iterative method that works well. Proposed originally by Suzuki and Ishihata for checking array programs, [7], this method is known as the *induction-iteration method*. The key concept to this method is finding the "weakest liberal precondition." We will notate as wlp(S,Q), where S is a program and Q a postcondition. A condition R = wlp(S,Q) if (i) Q is always true after S terminates (if S terminates) and (ii) no condition weaker than R satisfies (i). The key difference between the wlp and a weakest precondition is that we have no guarantee with a wlp that S halts.

The calculation of a wlp for a single loop is performed through back-substitution, starting with the postcondition of the loop. Formally, a recursive predicate W(i) is defined as

$$W(\ 0\ ) = \mathrm{wlp}(\ \text{loop-body}, Q\ )$$

$$W(\ i+1\ ) = \mathrm{wlp}(\ \text{loop-body}, W(i)\ )$$

The wlp of the loop is then defined as the conjunction of all W(i)'s. To calculate the wlp, one iterates over i till a W(i) is constructed that is strong enough to meet the wlp conditions. Further iterations are unnecessary as we are only looking for the weakest precondition. In [7], Suzuki and Ishihata proved that this would meet the three requirements for a loop invariant.

Figure 3 shows the pseudocode for their algorithm. The general idea is to find an L(j), where $L(j) = \Lambda_{j \geq i \geq 0} W(i)$, such that L(j) is true on entry into the loop and L(j) implies W(j+1). From the pseudocode, one can see that this approach does suffer from some inherent inefficiencies. Without the limit on the number of iterations, we have no guarantee that the algorithm will ever

```
Induction_Iteration() : SUCCESS | FAILURE {
i=0; Create formula W(0);
while (i < MAX_NUMBER_OF_ITERATIONS) {
    switch (Theorem_prover(($\Lambda_{i-1 \geq k \geq 0}$ W(k)) $\rightarrow$ W(i))) {
        TRUE: return SUCCESS;
        OTHERWISE:{ // try L(i)
            switch (Theorem_prover(wlp(<on-entry-to-loop>,W(i)))) {
                    TRUE: W(i+1)=wlp(loop-body, W(i));
                            i=i+1;
                    OTHERWISE: return FAILURE;
            }
        }
 }
}
}
```

Figure 3. Pseudocode for the basic induction-iteration algorithm. From [8].

halt. Suzuki and Ishihata discovered that with severely flawed programs, the algorithm could potentially enter an infinite loop. Another problem is that depending on the design of the theorem prover, the set of anded W(i)'s could create an exponentially large set of clauses. In practice, Suzuki and Ishihata were able to reduce this potential problem by cleverly designing their theorem prover. They were still limited to relatively simple programs, however.

The induction-iteration algorithm was later enhanced by Xu, et al. as described in [8]. While the main focus of their work was in performing safety checking on machine code, they found this method to be easily adapted for working on machine language programs. In doing so, they identified and addressed several potential drawbacks to the method. We will briefly mention a few of these:

1. First, the method does not naturally extend to working with nested loops. One cannot naively test that the W(i) of the inner loop is true upon entry into the inner loop. Calculations occurring in the outer loop might lead to W(i)'s that pass this test but are not invariant over the loop. Instead, they use the current L(j) of the outer loop and verify that L(j) implies W(i). This blocks bad W(i)'s from slipping by.
2. Conditionals within the loop body can wreck havoc with constructing L(j). In particular, they can degrade L(j) enough that it can never converge to being a loop invariant. The conditionals just add additional clauses. To work around this problem, $\mathrm{wlp}(\mathrm{loop\text{-}body}, W(i-1))$ is written in disjunctive normal form and each disjunct is tried for W(i). At least one of these is guaranteed to suffice. To make this pass over the disjuncts more efficient, a heuristic, breadth-first strategy is used.
3. To limit the exponential blow-up in clauses that could occur in the theorem prover, L(j)'s are condensed and simplified as much as possible.

In practice, Xu, et al. found that the loop invariant part of their verification algorithm performed fairly well. It is an automatic process that works well in practice on their problem domain. Interestingly enough, they found that limiting the number of iterations to three sufficed for most programs. This might just be unique to the programs they investigated. The loops in machine code might be significantly easier than those found in higher-level languages. Regardless, their work also found the induction-iteration method to be a major bottleneck in their verifier. They propose several improvements for reducing the cost of calling the theorem prover, including caching previous calculation results in the theorem prover.

The iteration-induction method is an old method for discovering loop invariants that is finding new use. However, the iterative process of building the invariant is not an intelligent search. It blindly searches for the next best potential solution. When an L(j) is found to not be an invariant, there is information available. Using the knowledge as to why it failed could be used to direct the search.

4 Using Critics and Failed Proof Attempts

As mentioned in the previous section, one possible way to improve the search for invariants is to utilize knowledge learned from failed attempts. Using failed proofs in this way has been extensively studied in theorem-proving work. The general idea is known as *proof planning*. A proof plan is essentially a tree representation of a proof of a particular theorem. Each node of the tree is some tactic or method to be used to prove the theorem. The plan essentially guides the theorem prover towards the correct proof, sometimes making mistakes and having to correct them. To improve the performance of proof plans, *critics* are used. Critics are additional methods that can "patch" a bad proof. For example, if a proof tries to use a rule that requires three preconditions to be true, but only two are, an appropriate critic is called. This critic will do something to make the step valid, either telling the prover it's at a dead end or better, proving a means for the prover to acquire the third precondition.

Ireland and Stark have made considerable steps in the application of proof planning to loop invariant detection, [6]. Their method utilizes a proof approach known as *rippling,* a heuristic used often in guiding inductive proof plans. To understand rippling, assume you have a statement (the target) and a rule that you want to apply to the statement. Currently, the rule cannot be applied to target because target is not in some proper form (i.e., the rule insists on left-first associativity: (A+B)+C and not A+(B+C)). This proper form is the goal statement. Rippling takes advantage of syntactical similarities between the target and goal statements to apply allowed rewrite rules to the target. These rewrite rules, sometimes called "wave rules," are derived from given definitions and lemmas. Eventually, rippling propagates through the target, resulting in the desired goal statement. This propagation has often been described as a "wave-front" moving across the statement.

Ireland and Stark's work showed that rippling loop invariant proofs are syntactically similar enough to inductive proofs to allow the application of rippling. This is important in that previous work on rippling created a large family of critics associated with this method. These critics that can guide and enhance a proof plan for inductive proofs can thus do the same for invariant proofs. This all falls naturally in line with the idea of proof planning. Each time an incorrect invariant is found, critics are called that use this failed attempt to correct the invariant. Ireland and Stark showed that the way rippling operates allows the discovery of invariants from incorrect invariants. Essentially, to find a loop invariant, a guess is made and rippling is applied.

In applying this approach to relatively simple loops (i.e. exponentiation and summation programs), this rippling approach performs quite well. Significant improvement was made when they used the postcondition as the initial guess for the invariant. Using the postcondition is a likely first guess for a human trying to find the invariant, so this is intuitively good. One important observation they noted was that multiple critics could often discover the invariant, but certain critics required less overhead (branching, number of rewrite rules, etc.) in the theorem prover. This opens up the new question of deciding which critics to apply when.

While this method has shown initial promise, it needs to be applied to larger programs. In particular, their work does not appear to consider dealing with nested loops. It does seem reasonable that rippling could be applied from the innermost loops out. Regardless of how future tests perform, their work is important. Ireland and Stark, researchers in artificial intelligence, utilize a well-known technique in theorem proving instead of just generically throwing a theorem prover at the problem. While there are no statistics as of yet to compare the runtime costs of their method to the induction-iteration method, their invariant discovery search is more intelligent.

5 Predicate Abstraction

Another example of recent work in inferring loop invariants that uses advanced AI techniques is the predicate abstraction work from Flanagan and Qadeer, [5]. One of the driving initiatives of their work was to improve upon ESC/Java. ESC/Java checks statically for programming errors that normally occur at runtime. Several of these errors, such as array bound errors, involve testing for loop invariants. ESC/Java requires a programmer to essentially annotate the loop invariants prior to checking, especially in the case of universally-quantified invariants. As discussed before, this is a long, tedious process that most programmers do not want to do. One goal of their research then was to create a system that automatically infers any type of loop invariant. Previous work by others had produced predicate abstraction-based invariant finders, but none of these approaches could handle universally-quantified loop invariants. In other words, they could not generate correct loop invariants for unbounded data like arrays or vectors. These programs also had only mediocre performance due to the fact that the number of calls to the theorem prover was exponential in practice. Despite the seeming mutual exclusiveness of inferring all types of loop invariants and getting good performance in practice, Flanagan and Qadeer managed to meet both of these goals.

As mentioned already, their approach involved *predicate abstraction.* Predicate abstraction is basically what it sounds like. Predicates are abstracted from a problem to another universe where they are treated as boolean variables. Formally, consider a set of predicates $p_1, \ldots, p_n$ and a matching set of boolean variables $b_1, \ldots, b_n$. An abstract domain element *f* is a boolean function over these boolean variables. To bring the predicates back into the "concrete" universe, a concretization function $\gamma(f)$ is used where: $\gamma(f) = f(b_1 \leftarrow p_1, \ldots, b_n \leftarrow p_n)$. The main idea is that by abstracting, one has to manipulate only a finite number of boolean variables as opposed to predicates with potentially infinite values. Furthermore, it is easy to incorporate universally-quantified loop invariants using predicate abstraction by allowing the predicates to refer to skolem constants (variables not used in the program but added by us to aid our verification proofs). Since the constants are not used in the program but are fixed to some unknown value, there is no loss in rigor or correctness when they are universally quantified in the final loop invariant. The earlier predicate abstraction approaches could have done this approach as well but would have suffered with the burden of significantly more overhead.

The pseudocode for inferring loop invariants is found in Figure 4. We refer the reader to [5] for complete understanding of this function as we want to focus on the abstraction function $\alpha(Q)$, where Q is a predicate. this function returns the strongest boolean function such that $Q \rightarrow \gamma(\alpha(Q))$. The basic idea of the infer function can be seen on line 10 in figure 4. A new invariant is calculated by $\alpha(Q)$ and ored with the old version. If the two remain the same, which happens since $\alpha(Q)$ is the strongest function, then they have found their invariant.

The majority of [5] is dedicated to how Flanagan and Qadeer optimized the calculation of $\alpha(Q)$. One, they analyze loops in-order. This comes from the insightful observation that the invariant of one loop will constrain the possible initial states of a subsequent loop. This will lead to faster

```
<Formula; Stmt> infer (Stmt C, Stmt S) {
1: let "{P,I} while e do B end" = S;
2: Stmt H = havoc(targets(B));
3: AbsDomain r = _(Norm(true; C));
4: while (true) {
5:   Formula J = γ(r);
6:   Stmt A = "assume e ∧ I ∧ J";
7:   Stmt B' = traverse("C ; H ; A", B);
8:   Formula Q = Norm(true, "C ; H ; A ; B' ");
9:   AbsDomain next = r ∨ α(Q);
10:  if (next = r) return <J,B'>;
11:  r = next;
12:  }
13:}
```

Figure 4. The *infer* function used by Flanagan and Qadeer. *Traverse* is a function that traverses over a statement and returns an inferred invariant for each loop in that statement. From [5].

convergence to an invariant due to the smaller solution space that has to be searched. In calculating $\alpha(Q)$ itself, the basic idea is to identify the relationship between the predicate Q and the predicates p_i. In terms of the abstract domain, this entails finding all maximal clauses (clauses that contain n literals of the boolean variables b_1 through b_n) that imply Q and then anding them together. The simplistic approach is to just try every possible maximal clause, but this would require 2^n queries on "long" clauses to the theorem prover. Instead, they simplify this procedure in three ways:

1. One optimization involves the calculation of $r \vee \alpha(Q)$. Due to logical consequence, one only needs to consider maximal clauses that are implied by r.
2. This optimization is a heuristic based on the observation that in practice, $\alpha(Q)$ can often be expressed as a conjunct of small, non-maximal clauses. By observation, it was noted that three generally worked. With this in mind, they greedily strip off literals from a valid maximal clause m to get a smaller clause c. The previous idea can be used to improve the quality of what is stripped as well.
3. Another optimization is to use a divide-and-conquer approach that works to shrink a maximal clause m to a stronger clause c such that $r \rightarrow c$ and $Q \rightarrow \gamma(c)$. This clause c can then be further stripped greedily as mentioned before. Calling this an optimization is not fully accurate. It is possible to have $O(n \cdot 2^n)$ queries to the theorem prover in the worst-case. However, according to Flanagan and Qadeer, this rarely occurs in practice. In fact, one can look at their divide-and-conquer procedure and see that such a loop would be a very pathologically unfriendly loop.

These three optimizations significantly reduce the number of clauses that need to be enumerated. Furthermore, the sizes of the clauses are significantly reduced, thus improving the performance for the theorem prover when it is called.

There is one problem with this predicate abstraction approach that Ireland's and Stark's work from the previous section did not suffer from: where do the predicates come from? Recalling part of Flanagan's and Qadeer's goals, they wanted to avoid having programmers annotate the code. Listing predicates is a time-consuming task on the same order of determining the loop invariants themselves. Thus, they present also in [5] a series of heuristics that generate predicates from the code itself. These heuristics are designed to work in ESC/Java, and are thus limited by the annotation language therein.

When tested, the predicate-finding heuristics and the predicate abstraction loop invariant inferer performed quite well compared to another tool. In particular, the reduction in theorem prover queries was quite significant. In terms of correctness, the modified version of ESC/Java was pitted against an unmodified version on a program with 520 loops spread out over 2418 routines.

The unmodified version could not verify 326 of these routines, most of which contained non-trivial loops. The modified version improved the verification to all but 31 routines. Upon human inspection, the loops involved in these routines involved subtle invariants and/or predicates the heuristics could not generate.

Flanagan and Qadeer's approach to loop invariant inference is another good example of using an advanced artificial intelligence technique on this problem. Aside from the improved performance and their capability of handling universally-quantified invariants, the important contribution of this work is that they did not strive for perfection, instead focusing on giving a good enough answer most of the time. As the author's point out, their method significantly reduces the annotation burden on the programmer.

6 Dynamic Invariant Detection

In step with the idea of not generating every loop invariant, we now turn to the idea of dynamic invariant detection. This involves running a program over a test suite of inputs and examining what the program computes. Then, this data is analyzed for patterns and relationships among the variables. Potential *program invariants* are then proposed and validated. Note that this procedure is not just for loop invariants; other program invariants, like preconditions and postconditions, can also be generated. Furthermore, any invariant found is only an invariant over the test suite. An input not in the test suite might invalidate a dynamically determined invariant. However, these guessed invariants can be useful to a programmer. Confidence about the program's correctness can be raised, and the validity of the test suite itself can be judged.

Daikon, a prototype tool using this approach created by Mike Ernst and others, [2,3,4], has shown initial success at using this dynamic detection method. This program works in three stages. The first stage manipulates the program to record data traces. The second stage then runs the altered program over the provided test suite and records the data trace. Finally, in the third stage, invariants are proposed based on this data. To bound the search, Daikon limits itself to checking for only specific types of invariants:

- Invariants for up to three numeric variables:

  Examples: $x > 2,\ \ y = 3x - 1,\ \ x + y \equiv 3 \bmod 5$
- Invariants for up to two sequence variables:

  Examples: element ordering, minimum sequence values, subsequences

The search itself is fairly efficient. Each potential invariant is checked over each data sample collected. One an invariant is found to be invalid on one data sample, no further tests on that invariant will occur. For the more complicated invariants, like $x = ay + bz + c$, where x, y, z are variables and a, b, c are unknown constants, full theorem prover queries are not really needed. Three data samples will usually be enough to determine a, b, and c using standard linear algebra, and then other samples can be checked on this equation with simple arithmetic. Daikon does not return all invariants found, though. Through a probability analysis, it gives a confidence estimation on each invariant and only returns an invariant if the confidence is high enough.

In the current version of Daikon, loop invariants are not dynamically tested for. Part of the motivation of creating Daikon was to aid programmers in software evolution. When making changes to code, loop invariants are likely to not have as far-reaching dependencies as compared to pre and postconditions. Furthermore, when the normal data trace methods are used in stage 2 on loops, an enormous amount of repeated data is returned. [2] discusses some means of reducing the amount of information returned. Not only can this easily become unmanageable, it could potentially skew the loop invariants returned. With lots of instances of a particular data sample, Daikon will view the probability of it occurring by chance as very small, therefore assigning it a high confidence. This effect is similar to the over training problems that can occur with neural networks, decision trees, and other machine learners. Despite this current design decision, it should be noted that in early tests of Daikon, [3], loop invariant detection was explored. On a series of small test programs, Daikon successfully detected loop invariants. The overconfidence and data glut did not occur with these small programs. Of particular note is that Daikon found some needed loop invariants that were left out of the test programs' original specifications.

Rating Daikon's abilities on inferring loop invariants is difficult without tests on more complex programs. The tool, however, introduces another approach to guessing probably loop invariants: data mining. Getting a data trace of a loop over many iterations over many inputs should provide a considerable amount of information from which a good initial guess can mined. It is often the case that searching for a solution proceeds faster if one can start the search relatively near the solution.

7 Further Artificial Intelligence Approaches

In the course of this literary review, we were a bit surprised to find the apparent lack of papers using artificial intelligence approaches to find loop invariants. As we have previously mentioned, the first work in loop invariant detection took place in the early 1970's in compiler research. Since then, there appears to have been little research in the area despite any advances in artificial intelligence. While we acknowledge the work in [5] and [6], we believe that AI techniques have great potential in inferring loop invariants. In the following paragraphs, we will briefly brainstorm the possible pros and cons of using two standard AI approaches: machine learning and randomized search.

In all actuality, we do not expect machine learning to be of much value in this problem domain. Some of the Daikon papers, [2] and [3], devote a section to this point of view. One view of machine learning is the creation of a special widget for making decisions. After training the widget on a variety of examples (some of which might be erroneous, i.e. noise in the training set), the widget can then make usually accurate predictions for future situations. How can this be applied to finding loop invariants? Due to the incredible variety of loops that exist in programs, we would need a training set that covers a significant number of these to properly train the widget! This does not seem practical or even remotely feasible. Another reason against using machine learners is that we really do not have a situation that requires learning. Consider our widget again. The widget looks at loop L1 from program A, makes a guess at an invariant, and learns that it was wrong (or right). Will this help at all when we want to find the invariant for loop L2 from program B? The answer is probably not if the programs are significantly different in structure. While we would suggest keeping an open mind, we do theorize that machine learning is likely not a viable source of ideas for finding loop invariants.

Despite the above comments, there is one potential use for machine learning that is related to loop invariant finding. In Section 4, we mentioned Ireland and Stark's observation that different critics performed more efficiently on different loop invariant proofs. It is reasonable to surmise that a learner could be developed that upon being given a loop to analyze, it would determine the appropriate critic to use. This is not a direct application of AI to the loop invariant problem but does propose one way of improving the efficiency of invariant finders.

Random search, as opposed to machine learning, appears to have promise for finding loop invariants. For notoriously difficult problems where deterministic approaches can only do so well, such as satisfiability, *intelligent* random search has been shown to be amazingly accurate under most situations. The emphasis on intelligent is important. Let us consider the problem space we have:

Given: a set of variables X // finite
a set of values Y that the variables can take on // potentially infinite
the Hoare statement H: {P} while C do S {Q}
Goal: A loop invariant I for H that is composed of predicates derived from X and Y

One can expect a randomly chosen set of predicates derived from X and Y will be a loop invariant with extremely low priority. Thus, pure random search is unlikely to produce invariants within reasonable time. Intuitively, though, one can start from a poor first choice and incrementally improve it till it becomes an invariant. This local repair strategy has been used successfully on difficult problems like satisfiability. With this in mind, we now look at more guided random searches like hillclimbing and genetic algorithms instead of pure random search.

In order to use these intelligent search methods, we will require a metric to assess the quality of different potential invariants. We need some way of saying potential invariant I1 is better than potential invariant I2. Furthermore, we need this metric to work incrementally. Continuing the example, we change I2 to get I2' and find that I2' is better than both I2 and I1. Determining a good metric would likely be the most difficult (and likely the most interesting) task in designing a random search based invariant finder. One potential source for a metric would be to look at the proof plans or trees used to prove an "invariant" as faulty. One can reason that the longer the disproof takes, the more accurate the invariant was.

Let's assume now that we have such a metric. For the sake of this thought experiment, we will loosen our requirements and make our loop invariant finder part of an interactive discovery system that uses a genetic algorithm to find invariants. The programmer writes down a list of what he or she thinks are the invariants for the code. These predicates form the basis for all the individuals in the population. Some individuals are subsets of this basis, while others include statements about variables from the code not mentioned in the basis. Some individuals might also contain mutations of what the programmer entered, e.g. $x < 3$ instead of $x < 10$. The mutation and crossover operators for this population are fairly obvious. The genetic algorithm then runs, breeding potential invariants till it finds an actual invariant. Essentially, it is doing in parallel what a human would do in trying to find invariants: guess, correct, guess, correct, ... until an invariant is found.

There are some potential downsides to this approach. One, during each generation, we will have to do something to determine the quality of each individual in the population. This might be computationally expensive, particularly if numerous theorem prover calls are required. If this is the case, we might need to explore some the optimization techniques, like caching in the prover, proposed in [8]. Another problem is that we've cheated and had a user guess the initial invariant. While this might be perfectly reasonable for some software development situations, guessing invariants for a program containing thousands of loops would be tedious. However, we have several options for automating the guess. One, similar to the induction-iteration method and Ireland and Stark's work, we can start from the loop's postcondition. Another option is to use the heuristic predicate finder from [5] and then just randomly assign values as needed. A third approach would be to do data mining off of trace data from a test suite as Daikon does. It is difficult to say which approach would be best. Also, it opens the interesting research question of how much does the quality of the initial guess affect the convergence rate of the genetic algorithm?

However, we are still ignoring one problem with using these randomized searches: sometimes they fail to ever converge. If we run a search x times without generating a solution, we have no guarantee that there is no solution. Sometimes, we can use the search to say that the chance of there still being a solution is some probability p(x). At this point, we can adopt the philosophy used by Flanagan and Qadeer. If we have managed to improve the search time for a large number of problem instances, but some problems still remain difficult to solve, then that is good enough. And if we are lucky, the cases that appear most often in practice will be the ones our new method works well on.

Randomized searches, like we mentioned with machine learning, might also be useful for problems related to invariant finding. For example, we surmised that Flanagan and Qadeer's predicate abstraction approach failed to find all invariants partially because their heuristic predicate generator could not generate all the predicates needed. Consider a large set of potential heuristic rules for generating predicates. Some of these are probably unneeded and a few probably contradict each other. What one really wants is the subset of these that generates the "best" predicates. Defining what is "best" returns to the issue of finding a metric, but search methods could be applied as well to this problem.

These approaches seem promising on this first-glance, intellectual level. However, it bothers us that we could not find any literature that described any attempt even remotely similar to the ideas we just proposed. It might be the case that software engineers have not considered these approaches and that AI researchers have not yet focused on this problem domain. Unfortunately, this might be an example of where researchers attempted an idea, found it didn't work, and opted not to publish the negative results.

8 Conclusion

The automatic finding of loop invariants will continue to expand and evolve from the ideas presented in this paper. Due to the gap in research, old ideas need to be investigated and improved upon. While improvements in the field of theorem provers will enhance the performance of the invariant finders, it is the author's belief that future work will rely more on heuristic and potentially random approaches. Furthermore, the focus will move away from perfectly finding loop invariants to improving the performance for a large number of cases.

Bibliography

[1] A. Blass and Y. Gurevich. Inadequacy of Computable Loop Invariants. *ACM Transactions on Computational Logic.* 2(1):1-11, January 2001.

[2] M. Ernst. Dynamically Detecting Likely Program Invariants. PhD Dissertation, University of Washington, Department of Computer Science and Engineering, August 2000.

[3] M. Ernst, J Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likepy Program Invariants to Support Program Evolution. *IEEE Transactions in Software Engineering*, 27(2):1-25, February 2001.

[4] M. Ernst, A. Cseizler, W. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000),* Limerick, Ireland, June 7-9, 2000, pp. 449-458.

[5] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 2002.

[6] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. Fourth Nasa Langley Formal Methods Workshop, 1997.

[7] N. Susuki, and K. Ishihata. Implementation of an Array Bound Checker. *4th ACM Symposium on Principles of Programming Language*s. Los Angeles, CA. (January 1977).

[8] Z. Xu, T. Reps, and B. Miller. Safety Checking of Machine Code. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000),* Vancouver, B.C., June 2000.