Searching for Invariants using Genetic Programming and Mutation Testing

Sam Ratcliff
Dept. of Computer Science
University of York, UK
sr536@student.cs.york.ac.uk

David R. White
Dept. of Computer Science
University of York, UK
drw@cs.york.ac.uk

John A. Clark
Dept. of Computer Science
University of York, UK
jac@cs.york.ac.uk

ABSTRACT

Invariants are concise and useful descriptions of a program's behaviour. As most programs are not annotated with invariants, previous research has attempted to automatically generate them from source code. In this paper, we propose a new approach to invariant generation using search. We reuse the trace generation front-end of existing tool Daikon and integrate it with genetic programming and a mutation testing tool. We demonstrate that our system can find the same invariants through search that Daikon produces via template instantiation, and we also find useful invariants that Daikon does not. We then present a method of ranking invariants such that we can identify those that are most interesting, through a novel application of program mutation.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic Methods

General Terms

Algorithms, Verification, Experimentation

Keywords

Invariants, Daikon, Genetic Programming,Mutation Testing

1. INTRODUCTION

Invariants are mathematical descriptions of a program's behaviour. They specify the state of a calculation at a specific point or set of points in a program. An example is given in Algorithm 1, from [10, 14], where the process of summing an array is described by a precondition, a postcondition and a loop invariant. The program sums the contents of array b into variable s. We consider pre- and post- conditions as specific types of invariants.

Algorithm 1 Array sum program

```
$i,s := 0, 0;$
do $i \neq n \rightarrow$
    $i, s := i + 1, s + b[i]$
od

Precondition: $n \geq 0$
Postcondition: $s = \sum_{j=0}^{n-1} b[j]$
Loop invariant: $0 \leq i \leq n$ and $s = \sum_{j=0}^{i-1} b[j]$
```

Invariants provide information that may be used in many types of program construction and analysis. For example, in compiler optimisation, invariants support the application of partial redundancy elimination [19]. Invariants are useful in constructing or extending software in the paradigm of *design by contract* [18]. They can be used in software testing, where a tool may attempt to violate a given set of invariants.

Invariants are not always readily available: they must usually be specified manually at some point in the design process, which is time-consuming, and most commercial software is therefore not annotated by invariants. Thus, automated recovery of invariants from existing source code that was developed without their explicit use deserves attention.

In this paper, we demonstrate a new method of generating invariants from source code. We compare our method to the existing invariant generation tool Daikon, and then use mutation testing to identify "interesting" invariants. Such invariants are those that succinctly capture the intuition and semantics behind a program. We find that these new applications of heuristic search and mutation testing are effective, and suggest promising new lines of research.

1.1 Contributions

The main contributions of this paper are to present our method of invariant recovery and to demonstrate that:

- Evolutionary search can be used to find program invariants such as those found by Daikon.
- Evolutionary search is able to consider a wider range of invariants than template-based tools.
- Interesting invariants may be identified by combining mutation testing with evolutionary search.

2. BACKGROUND

In this section, we describe the origins of program invariants, the invariant tool Daikon, and give an overview of genetic programming and mutation testing.

2.1 Invariants

Invariants were originally devised as an aid to program construction. By first specifying the pre- and post- condition of a procedure, several authors have argued the case for systematic derivation of the program [8, 14]. In the 1990s they were incorporated in object-oriented development [18].

Ernst et al. [11] assert that the presence of "explicit invariants can help programmers by characterising certain aspects of program execution and identifying program properties that must be preserved when modifying code". They give a small case study where programmers were presented with invariants to aid them in modifying existing software.

2.2 Daikon

Daikon is an invariant detector [9, 10]. It generates invariants based on execution traces, rather than through symbolic execution or static analysis. It is divided between a language-independent invariant detector and a language-specific front-end that generates program traces [2].

Daikon uses a database of predefined templates, which are instantiated to suggest invariants for the given trace data. Templates are represented as Java classes; users may add templates by creating a new class. For example, an invariant checking a lower bound of a variable such as "$x \geq 0$" is represented by a single class, which we may extend to introduce a new invariant "$x \geq 1$". Daikon will then consider all instantiations of this invariant for a given method entry or exit, by replacing x with each numeric variable in scope. To make such an brute-force approach tractable, a maximum of three variables can be involved in an invariant. Variables used to instantiate these templates may actually be derived from other variables, such as indexing an array, and there is a limit to how "deeply" these derivations may proceed.

Each candidate invariant is tested against the data in the program traces, consisting of state information at method entry and exit points. By inserting extra method calls, loop invariants can also be generated. The ability of Daikon to find useful invariants depends upon the quality of the trace data: if boundary cases are not covered then Daikon will be unable to correctly specify partitions of a state space. This limitation applies to any approach reliant upon test data.

The Daikon approach can be construed as a process of brute force construction followed by data driven restriction. Various heuristics have been applied to reduce the number of invariants produced and to identify the most useful invariants to present to a programmer. A limitation of this approach is that Daikon will only produce invariants that correspond to instantiations of its templates. Thus there may exist useful invariants that will not be found by Daikon.

2.3 Genetic Programming

Genetic programming (GP) [6, 16, 20] is a heuristic search method that can be used to locate programs, formulae and other expressions. It is an evolutionary algorithm, in that it takes some inspiration from biology and maintains a population of individual programs. Programs (or expressions) are evaluated according to a fitness function, selected according to their fitness values, and then recombined using operators such as subtree exchange to create the next generation.

The most traditional form is tree-based genetic programming, which is the type of GP used in this paper. Figure 1 gives an example of how we might represent the loop invariant from Algorithm 1 in tree-based GP.

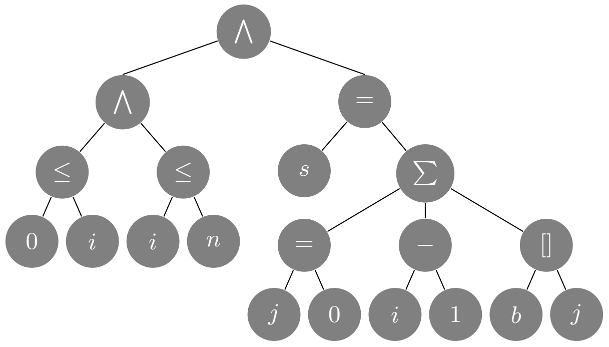

Figure 1: An expression tree representing the loop invariant from Algorithm 1

Many aspects of software engineering can be formulated as a search problem [7]. Recently heuristic search methods such as GP have been applied to program testing and analysis [22] and manipulation of existing source code [13, 23]. Solutions to such problems can be stated in an expression language, and are often amenable to search.

Genetic programming can be used as an "invention machine", in that it is relatively free of human assumptions when searching for solutions. It has been used previously for tasks that may be described as requiring insight, intuition, and precisely the type of understanding involved in manually creating invariants. The search space for the algorithm is defined by the function set we have chosen, but the search is free to find arbitrary invariants within that space.

2.4 Mutation Testing

Mutation testing [15] was devised as a method of measuring the effectiveness of test data. Here we propose a very different application of mutation testing, but it is instructive to give a little background regarding its original purpose.

The principle behind mutation testing is that if a program p is tested using test set T, then assuming that p is correct, it should pass all the tests in T. However, if the program p is mutated (a small syntactical change is applied) to produce p', then p' should fail to pass the test set unless p and p' are semantically equivalent. Through careful choice of mutation operators, the purpose of mutation testing is to create test sets that reflect program requirements and are specific enough to fail when common programming errors are made.

Interestingly, one existing relationship between invariants and mutation testing is the use of invariant generation to eliminate semantic equivalents in order to improve the efficiency of mutation testing [21]. In this paper, we exploit this relationship in the opposite direction: we use mutation testing to improve the efficacy of invariant generation.

Many mutation toolkits exist for a variety of different programming languages. The toolkits vary in terms of the mutation operators employed, the level of abstraction at which the mutation occurs and the testing framework used. μJava [17] is a mutation tool for the Java language, and is the toolkit used to generate mutants in this paper.

3. EVOLUTION OF INVARIANTS

We now provide an overview of the invariant generation system that we have developed.

Rather than following Daikon's method of brute force con-

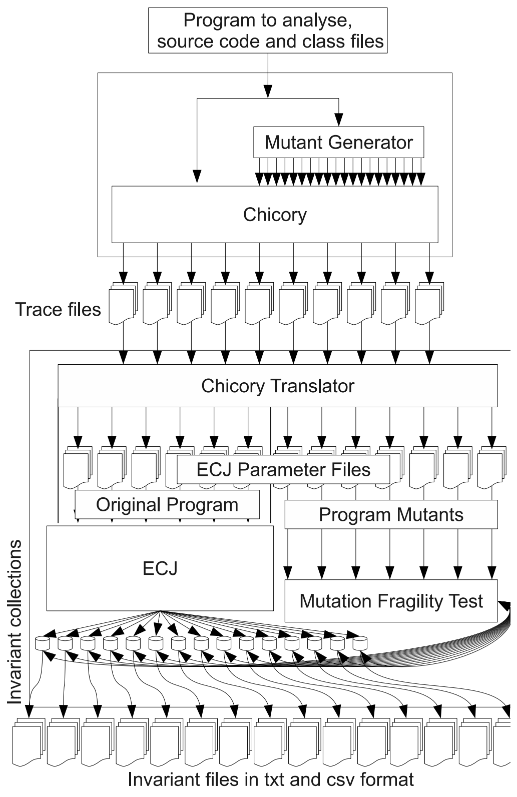

Figure 2: An overview of our method of invariant construction

struction and data-driven restriction, we take a different approach and use GP to construct candidate invariants in the form of expression trees. The tree representation uses program variables and constants as the leaves of the tree, whereas non-leaf nodes are chosen from a set of building blocks provided to combine variables and constants into invariants. A building block may be a simple mathematical function, but we also provide more sophisticated operators.

There is no restriction on the number of distinct program variables that can be involved in an invariant. The maximum size of an invariant is a parameter to the system limited by memory requirements. This allows for the construction of invariants that cannot be considered using Daikon. However, our approach does not necessarily ensure predictability: running Daikon with the same input will always return the same candidate invariants, whereas using GP does not *guarantee* this repeatability over different seeds. Also, as we will see, a much larger number of invariants may be produced.

3.1 Process Overview

Figure 2 gives an overview of our method of invariant construction. We take a program as input and sample test cases from a simple uniform distribution, although specific test data could be provided. Mutants of the original program were generated using a customised version of μJava. The original program and mutants were both run through Chicory, Daikon's front-end for Java. Chicory outputs the trace files for each of the mutants and the original program.

The next stage is to create parameter files for the evolutionary computation toolkit ECJ [3]. These files contain standard parameters to the genetic programming algorithm along with the trace data for the input program.

A GP search is then run for each method, including those added to find loop invariants. Standard tree-based GP is used, but mutation is favoured over crossover, with probability of mutation 0.9 and probability of crossover 0.1. A high mutation rate was found to be favourable in initial experimentation. Focusing on mutation does not reduce GP to a random search, rather it places the emphasis on individual improvements more than on subtree exchange.

Archiving is also added such that if an invariant is found that is consistent with all data points, it is added to a collection: this collection constitutes the output of the search process. Once an invariant has been encountered, the fitness of syntactically equivalent invariants is reduced, to encourage further exploration of the space of possible invariants.

Finally, mutation testing is employed to sort these candidate invariants by an estimation of their "usefulness" to a programmer. The candidate invariants found are passed through what we refer to as a "Mutation Fragility test". The consistency of each invariant is checked against the mutant trace data, and a record of the number of mutants that violate the candidate invariant is made. The intuition here is that *useful invariants are those general enough to be consistent with the provided test data yet specific enough to be inconsistent with the trace data of the mutant programs.*

3.2 Evaluating Individuals

A fitness function must be provided to ECJ to estimate the utility of a given invariant. We used the number of datapoints within a trace that satisfied the candidate invariant. We considered adding components such as a parsimony measure, and also using the addition of simple components to the fitness function to locate "interesting" invariants, but did not find them to be beneficial in our initial experimentation.

Once a fully consistent invariant has been found and added to the collection, subsequent individuals representing the exact same invariant are punished by reducing their fitness by a small amount equivalent to being found to be false for one trace data point. Semantic equivalents were not eliminated, as such filtering was outside of the scope of this paper.

3.3 Which Invariants are Interesting?

All invariant generation techniques are faced with the same concern: which candidate invariants are of most interest to the user? To some extent this depends on the intended use for the invariants, but there remains a large number of candidate invariants that are not useful. Tautologies such as $\mathrm{T} \vee \mathrm{F}$ are one example, but of greater concern are invariants that are not tautologies and are also not particularly descriptive, in that they could apply to many similar programs. The latter are much more difficult to eliminate.

Various techniques have been applied in previous work using Daikon [12] to eliminate candidate invariants. These methods are also applicable to our own work, but we adopt a new method that will enable us to order invariants according to some estimation of their interest to an end user. This approach is arguably more general than those used previously with Daikon, and is likely to be more suited to the very large numbers of invariants generated with search methods.

The feasibility of evolving invariants using GP, and the

Example	Inputs	Description
Abs	int x	x set to abs(x)
ArraySum	int[ ] b, int s	s set to sum(b)
FourTupleSort	int q0, q1, q2, q3	inputs ordered
GCD	int x, y	x set to gcd(x,y)
Max	int x, y, z	z set to max(x,y)
MinArray	int[ ] b, int x	x set to min. value in b
Perm	int x, y	x and y ordered by <

Table 1: Case Studies taken from Gries [14]

effectiveness of our prioritisation method, are examined in the following section.

4. EXPERIMENTATION

4.1 Overview

In this section, we describe three experiments:

- Experiment A - comparing the output of our framework to that of Daikon for a set of given example problems, mostly taken from Gries [14].
- Experiment B - investigating the ability of our framework to produce the invariants actually specified by Gries (which were not the same as those produced by Daikon) for problems presented in that text.
- Experiment C - evaluating the extent to which mutation testing provides a way of prioritising invariants by their likely interest to a user.

4.2 Case Studies

We chose to test the system on two sets of examples. Firstly, we used most of the examples from Gries [14] originally used to test Daikon [9], given in Table 1. This allowed us to evaluate whether evolution can find the same kinds of invariant as Daikon. Also, Gries provides loop invariants and pre- and post-conditions, such that we can evaluate the ability of search to find these key invariants. We can also use them to asses whether we can distinguish them from other candidates as potentially more "interesting" to a user.

Our second set of examples was taken from introductory programming texts and includes the sorting algorithms bubblesort, insertion sort, quicksort and selection sort. These examples are more complex than those given by Gries. Each involves at least one loop invariant.

The source code for each example is too large to be provided here, but we have made it available online along with all files needed to repeat our work [5]. The formal definitions of the programs in Table 1 may also be found in Gries.

Most input data was generated randomly. Pre-conditions specified by Gries were met in generating input data, and arrays were randomly initialised to a length in [1,100]. Integer values were within the range [-100,100] (or [1,100] where Gries had placed a constraint on those values). Limiting the size of integers involved reduced the size of the test data, simplified the programming involved, and made output more readable. The only exception to the random generation of input data was the GCD example, where 50% of the input data (x, y) was generated such that $gcd(x, y) \neq 1$.

The source code of the examples does not (and cannot, in Java) exactly correspond with the descriptions given by Gries. Certain implementation details such as wrapping classes used to hold output variables are removed from our analysis for clarity.

Function	**Description**
Functions over Variables	
$=$	Equals
$= 0$	Equals zero
$>$	Greater than
$\geq$	Greater than or equal to
≥ 0	Greater than or equal to 0
≥ 1	Greater than or equal to 1
%	Modulo
$\neq$	Is not equal to
$\neq 0$	Is not equal to zero
Functions over Arrays	
ArrayElement	Value at array position
ArrayLessThan	Lexical comparison of two arrays
ArrayLEQ	Lexical comparison of two arrays
ArraysEqual	Numeric comparison of two arrays
IsMemberOf	Membership of an array
LEQAllElements	Compare value to all values in array
MaxIndex	The last index of an array
NotNull	Check if array is not null
PreviousElement	Element at position prior to argument
Size	Array size
SortedArray	Is array sorted?
Functions used by Gries	
AND	Logical AND
ArraySum	Array sum
GCD	Greatest common divisor of variables
IsMemberSubArray	Does the subarray contain this value?
LEQSubArray	Compare value to subarray
≤ 0	Less than or equal to zero
Negative	Multiply by -1
OR	Logical OR
PermOfFour	Permutation of four values
PermOfTwo	Permutation of two values

Table 2: The Function Set

4.3 Evolutionary Algorithm Parameters

The function set supplied to GP was designed to allow the expression of the kinds of invariants produced by Daikon for the example programs we studied. The size of the invariants was not controlled, beyond the maximum depth limit set by ECJ to prevent exhaustion of RAM, and the implicit limitations of the language defined by the return and argument types of the function set.

Table 2 lists the functions provided to the search. Input programs are limited to those manipulating integers and integer arrays. For a given program, only the rele-

Example	Inputs	Output
BubbleSort	int[ ] x	x ordered by <
InsertionSort	int[ ] x	x ordered by <
QuickSort	int[ ] x	x ordered by <
SelectionSort	int[ ] x	x ordered by <

Table 3: Sort Case Studies, taken from Swartz [4] and Algolist.net [1]

Repetition	Number Found	Percentage Found
1	280	83
2	287	85
3	283	84
4	285	85
5	285	85
6	283	84
7	285	85
8	284	84
9	280	83
10	279	83

Table 4: Number of Daikon Invariants found by evolution

vant functions (for the available datatypes) were provided to the search, and the functions specific to those examples taken from Gries [14] were added for the relevant evolutionary runs.

The selection of these functions was made easier by knowing the types of invariants we were looking for: it is likely that we may be able to suggest relevant functions to some extent with any program, but certainly some manual guidance has been given through this selection. Providing more functions (for example, in the absence of any suggestions as to which should be discarded) would result in a larger search space, longer run times and a greater number of uninteresting invariants produced.

We used a type hierarchy implemented using Strongly-Typed GP, which relies on atomic and set types to restrict the search space. There are six atomic types in the system: Booleans, integers, arrays, derived integers, "conjunct Booleans" and "disjunct Booleans". The first three are self-explanatory. Derived integers are returned by ArrayElement, MinIndex, PreviousElement, Size, ArraySum and Negative. A separate "derived variable" type is used to avoid uninteresting comparisons between derived variables. The special Boolean types restrict invariants to a limited sized conjunctive normal form, to reduce the ability of search to generate arbitrary-size tautologies such as $s_1 \vee s_2 \vee \ldots \vee \mathrm{T}$. For full typing information, the parameter files are available online [5].

4.4 Experiment A

4.4.1 Research Question

In this experiment, we investigated whether evolutionary search could find the same kinds of invariants as Daikon for the example programs, and whether it was able to find most or all of the invariants suggested by Daikon.

4.4.2 Method

Each program from Tables 1 and 3 was run 10 times through Chicory to create 110 trace files. We ran Daikon 4.6.4 on the trace data, using its default settings, which generated a total of 337 candidate invariants consistent with the trace data over the 23 program points in the example programs. We ran ECJ using the array and variable functions given in Table 2 (i.e. without the Gries-specific functions), with population size and generations set to 100. Standard tree-based GP was used, with probability of crossover 0.1 and probability of mutation 0.9. All other parameters were left at their defaults in ECJ: our purpose here was to discover whether evolution could produce the same kind of invariants as Daikon, rather than the optimality or robustness of specific parameter settings. After the run was complete, we compared the output from the evolutionary search to that of Daikon.

Example	Program Point	Percentage Found (Median)
Abs	abs	100.00
ArraySum	arraysum	100.00
ArraySum	loop	92.86
BubbleSort	bubblesort	100.00
BubbleSort	inner loop	100.00
BubbleSort	outer loop	100.00
FourTupleSort	fourtuplesort	100.00
GCD	gcd	100.00
GCD	loop	66.67
InsertionSort	insertionsort	100.00
InsertionSort	inner loop	53.45
InsertionSort	outer loop	86.84
Max	max	100.00
MinArray	minarray	100.00
MinArray	loop	100.00
Perm	perm	100.00
Quicksort	dummy	100.00
Quicksort	partition	63.56
Quicksort	quicksort	100.00
Quicksort	quicksortrecursive	62.50
SelectionSort	selectionsort	100.00
SelectionSort	inner loop	72.50
SelectionSort	outer loop	100.00

Table 5: Percentage of Daikon Invariants found by Example and Program Point

4.4.3 Results

The percentage of Daikon invariants reported by evolution for each run is given in Table 4. The median percentage of the invariants discovered is 84%, and this figure is robust to the seed supplied. The results are grouped by example in Table 5. For many examples, all invariants suggested by Daikon were usually found by evolutionary search. It may be that more compute power is required to reliably find the "missing invariants", however it is clear that the same kind of invariants can be found using evolutionary search as those produced by Daikon. Many more invariants were produced by the evolutionary search than just those that matched Daikon: for example, across the first repetition a total of 17045 invariants were suggested, as opposed to 337 for Daikon. Many of these invariants could be eliminated via the same techniques as Daikon uses. However, the number produced by evolutionary search is markedly greater, and increases with the amount of computational effort used.

As an example of the large number of invariants that may be produced, examine Figure 3. The figure shows the median number of syntactically unique invariants over 10 runs for population and generation sizes 100, 250, 500 and 1000. The approach of searching for invariants necessitates a way of filtering this large set. Many of these invariants could be deemed uninteresting: some of them tautologies, some of them inequalities between evidently unrelated variables.

Program Point	Invariant	Success(%)
abs	$(x = orig(x) \wedge orig(x) \geq x) \vee (x = -orig(x) \wedge orig(x) \leq 0)$	0%
arraysum	$orig(n) \geq 0$	100%
arraysum	$s = \sum_{k=0}^{n-1} b[k]$	100%
arraysum.loop	$i \geq 0$	100%
arraysum.loop	$n \geq i$	100%
arraysum.loop	$s = \sum_{k=0}^{i-1} b[k]$	100%
fourtuplesort	$q0 \leq q1$	100%
fourtuplesort	$q1 \leq q2$	100%
fourtuplesort	$q2 \leq q3$	100%
fourtuplesort	$\{q0, q1, q2, q3\}$ is perm. of $\{orig(q1, q2, q3, q4)\}$	0%
gcd	$orig(x) \geq 1$	100%
gcd	$orig(y) \geq 1$	100%
gcd	$x = gcd(orig(x), orig(y)$	100%
gcd	$x \geq 1$	100%
gcd	$x = y$	100%
gcd	$gcd(x, y) = gcd(orig(x), orig(y))$	30%
gcd.loop	$x \geq 1$	100%
gcd.loop	$y \geq 0$	100%
gcd.loop	$gcd(x, y) = gcd(orig(x), orig(y))$	100%
max	$z \geq x$	100%
max	$z \geq y$	100%
max	$(z = x) \vee z = y)$	60%
minarray	$orig(n) \geq 1$	100%
minarray	$\forall k, 0 \geq k \leq n-1, x \leq b[k]$	100%
minarray	$x \in b$	100%
minarray.loop	$i \geq 1$	100%
minarray.loop	$n \geq i$	100%
minarray.loop	$\forall k, 0 \geq k \leq i-1, x \leq b[k]$	100%
minarray.loop	$x \in b$	100%
perm	$x \leq y$	100%
perm	$\{x, y\}$ is perm. of $\{y, x\}$	100%

Table 6: Success rates in finding Gries Invariants

There are two interesting observations to be made of the comparison with Daikon. Firstly, Daikon did not report most of the invariants that Gries used to derive the example programs. It may be the case that previous experiments that found the contrary relied on templates not supplied to Daikon by default. Secondly, Daikon produces a large number of invariants in some cases: 64 invariants are produced for the "partition" method in the quicksort example. Even this number may be too many for a programmer, and so our attempts at prioritising the most interesting invariants may be worthwhile not only for filtering the results of evolutionary search, but also the output of Daikon itself.

4.5 Experiment B

4.5.1 Research Question

The invariants found by both Daikon and evolutionary search in Experiment A are sometimes useful, but they are usually not the invariants that Gries used to derive the programs, that is the invariants that succinctly capture the semantics of the program. For example, in the case of the "Max" program, Daikon tells us that the output is greater or equal to both the original inputs, but that is not enough to capture the full behaviour of the program. The invariant (post-condition) we would prefer to discover is

$$(z \geq x) \wedge (z \geq y) \wedge (z = x \vee z = y) \qquad (1)$$

In Experiment B, we attempted to discover this and similar invariants. Perhaps Daikon could discover these invariants with suitable templates, but evolutionary search does not require such specific templates: we only need to add a small number of functions to our function set, such as logical OR and AND. This flexibility is the great benefit of using a heuristic search method, at the cost of predictability and the large number of uninteresting invariants generated (although the latter will be addressed shortly).

4.5.2 Method

To allow for the larger size of such invariants, we increased the computational effort to a population size and generation number of 250. The run-time of the evolutionary search for most problems was still a matter of seconds. The rest of the evolutionary algorithm was unchanged. We reused the trace data generated in Experiment A. We restrict ourselves to the Gries examples given in Table 1. Along with the function set used in Experiment A, we also included those functions that were used by Gries to specify invariants for these problems.

4.5.3 Results

The results are given in Table 6. Search manages to find more of the Gries invariants than Daikon, although it fails to find two of them. For the Abs and FourTupleSort examples, there is unlikely to be much of a gradient for the search. Although there may be some intermediate solutions that are partially correct, these are likely to be dominated by the number of other candidate invariants in the same population that are successful on more test cases. It is clear that more guidance must be given to the search, to punish the less interesting invariants.

The final experiment in this paper suggests one method of guidance, and the results in Table 6 are sufficient to enable us to proceed with a test of this method.

4.6 Experiment C

4.6.1 Research Question

In Experiment C, we investigated the efficacy of mutation testing to prioritise the invariants found in Experiment B by their potential interest to a user.

4.6.2 Method

We re-ran the seventh repetition from Experiment B and applied a mutant fragility test in order to highlight the Gries invariants found. In this evolutionary run, a total of 45 997 invariants consistent with the trace data were discovered by search. Gries specifies just 31 individual invariants. Is it possible that a mutant fragility test could help us quickly identify the 29 invariants from Gries found in that repetition of Experiment B, constituting only a fraction of a percentage of those invariants discovered by search?

To answer this question, we used μJava with its default settings to generate "Traditional Mutants" (method mutations) of the 7 Gries examples, giving a total of 310 compilable mutants. A single trace was generated for each mutant,

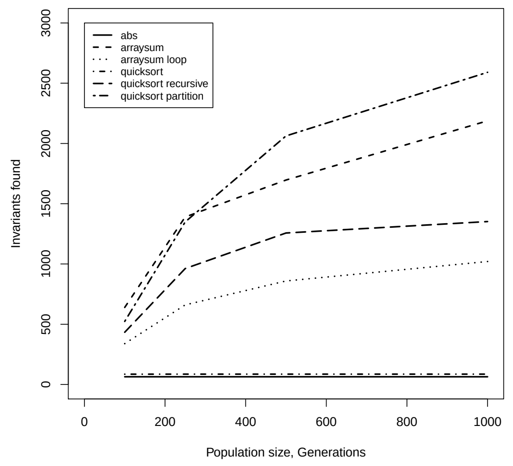

Figure 3: Number of Invariants found by Evolution

and this data was encoded in the ECJ parameter file for that problem. Using the same seed from the seventh repetition of Experiment B, we attempted to prioritise the invariants produced using mutation testing. At the end of the evolutionary run, each invariant i in the archive was assigned a priority score $p(i)$ as follows:

$$p(i) = \frac{1}{|M|} \sum_{m \in M} \frac{1}{|S|} \sum_{s \in S} c(i, s) \quad (2)$$

Where M is the set of mutants for the relevant program point, S is the set of non-empty samples for that program point and $c(i, s)$ is 1 if invariant i is consistent with the sample s and zero otherwise. Although we only created a single trace for each mutant, three of the program points were loop invariants and therefore potentially had zero, one or more than one sample point. A lower value of $p(i)$ indicates that the invariant was broken by many mutants, intuitively implying that it captures something particular to that program, and that it is therefore a more interesting invariant.

4.7 Results

Table 7 gives the results. Each Gries invariant found in Experiment B is listed. The “ranking” column indicates the position of the invariant ordered by mutation score. With a limited set of mutants, there were a limited number of unique scores, and many invariants were ranked as equally interesting. This can be seen by the limited number of rankings in the table. The “depth” column indicates how far down a list sorted by priority a user would have to read to find the invariants. Without the priority score, the user would have to examine the total number of invariants. Thus, by comparing the last two columns, we can see how the priority score has affected the amount of manual inspection needed.

The first observation we may make is that in general mutation testing has greatly reduced the number of invariants a user must check. In some cases, it has worked spectacularly well: these figures are highlighted in the table. Consider the last GCD loop invariant: from 3114 candidate invariants, mutation testing ranks this important one specified by Gries first. Similarly, the post-condition that most summarises the behaviour of ArraySum is lifted to the top of the list, along with four others. Many more are in the top 10, 20, 30 or 40 invariants. These numbers are sufficiently small enough to be practical.

Method	Invariant	Ranking	Depth	Total
arraysum	$orig(n) \geq 0$	9/9	1837	1837
arraysum	$s = \sum_{k=0}^{n-1} b[k]$	1/9	**4**	1837
arraysum.loop	$i \geq 0$	6/9	340	789
arraysum.loop	$n \geq i$	5/9	289	789
arraysum.loop	$s = \sum_{k=0}^{i-1} b[k]$	1/9	**13**	789
fourtuplesort	$q0 \leq q1$	16/23	123	557
fourtuplesort	$q1 \leq q2$	19/23	136	557
fourtuplesort	$q2 \leq q3$	17/23	142	557
gcd	$orig(x) \geq 1$	26/26	1685	1685
gcd	$orig(y) \geq 1$	26/26	1685	1685
gcd	$x = gcd(orig(x), orig(y)$	2/26	**15**	1685
gcd	$x \geq 1$	25/26	1228	1685
gcd	$x = y$	9/26	135	1685
gcd	$gcd(x, y) = gcd(orig(x), orig(y))$	11/26	243	1685
gcd.loop	$x \geq 1$	30/90	358	3114
gcd.loop	$gcd(x, y) = gcd(orig(x), orig(y))$	39/90	691	3114
gcd.loop	$x = gcd(orig(x), orig(y)$	1/90	**1**	3114
max	$z \geq x$	13/13	11055	11055
max	$z \geq y$	7/13	555	11055
max	$(z = x) \vee z = y)$	10/13	1172	11055
minarray	$n \geq i$	10/10	1506	1506
minarray	$\forall k, 0 \geq k \leq i-1, x \leq b[k]$	2/10	**22**	1506
minarray	$x \in b$	3/10	**24**	1506
minarray.loop	$i \geq 1$	12/25	571	2173
minarray.loop	$n \geq i$	12/25	571	2173
minarray.loop	$\forall k, 0 \geq k \leq i-1, x \leq b[k]$	3/25	**38**	2173
minarray.loop	$x \in b$	1/25	**18**	2173
perm	$x \leq y$	7/16	**40**	243
perm	$\{x, y\}$ is perm. of $\{y, x\}$	1/16	**8**	243

Table 7: Prioritising Gries Invariants

There are some invariants that are not prioritised as we may require. On closer inspection, some of these are invariants that capture behaviour external to the method that has been mutated, so it is impossible for these invariants to be broken. We may also consider that such pre-conditions, when they are not enforced in code internal to a method, are uninteresting to us when thinking about that method in isolation. Wider mutation testing across the test harnesses used to generate traces would solve this problem.

This way of prioritising invariants seems promising. It provides a way of filtering the output of the search process, but it could also be used to *guide* the search process, by incorporating the fragility test into the fitness function. Furthermore, we suggest that this approach may be used separately, for example in conjunction with Daikon, to reduce a list of invariants provided by another source.

5. CONCLUSIONS & FUTURE WORK

5.1 Improving this Method

We have demonstrated that invariants can be found using search, and that the large number of invariants produced may be prioritised through the novel application of mutation testing. There are two outstanding problems to be solved: firstly, to reduce the number of uninteresting invariants produced and secondly, to guide the search to invariants that may be interesting but "deceptive" to the search.

Some techniques to reduce the number of candidate invariants have already been implemented in Daikon, which could be used here. We would like to focus the search on more interesting parts of the search space. A scheme of rewarding invariants containing "interesting" functions was found to be ineffective in some initial experimentation. We instead suggest that the mutation testing part of our work is used as a component of the fitness function, such that the search favours those invariants broken by mutation. We may only accept invariants above a certain level of interest.

This integration of mutation testing within the search process may also improve the ability of search to find more complex invariants, by focusing on those that are most interesting. We also suggest that an improved representation is adopted, to facilitate a gradient in the search space, and that a richer function set is used. Furthermore, rather than using the simple and discrete fitness function based on the number of sample points consistent with the invariant, perhaps fitness could be based upon a distance metric that represents how incorrect an invariant actually is.

5.2 Filtering Daikon Invariants

The particular application of mutation testing in this paper is as far as we know unique, and it represents a substantial advance in itself. We are currently applying this method to rate the interest of invariants produced by Daikon, as often Daikon produces a large number of invariants itself. This represents a very generally applicable way of isolating interesting invariants from a crowd of candidates.

5.3 Trace Data Coverage

The quality of the invariants produced is heavily dependent on the quality of test data. For example, if a boundary case is not covered then a branch may be missed. By integrating the search with software (even search-based) testing methods, we could provide more accurate data both as inputs for our own framework and Daikon. Scalability could be improved by co-evolving a subset of the samples.

6. ACKNOWLEDGEMENTS

We would very much like to thank Jeff Offut for supplying us with the source code to μJava. Also to Juan Tapiador and Bill Langdon for insightful discussions, and to Matthew Patrick and Yue Jia for advice on mutation testing. This work is funded by SEBASE (EP/D050618/1).

7. REFERENCES

[1] Algolist. http://www.algolist.net/.

[2] Daikon homepage. http://groups.csail.mit.edu/pag/daikon/download/.

[3] ECJ Website. http://cs.gmu.edu/˜eclab/projects/ecj/docs/.

[4] Fred Swartz's Java notes. http://www.fredosaurus.com/.

[5] Online resources for this paper. http://www.cs.york.ac.uk/˜drw/papers/gecco2011.

[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[7] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating Software Engineering as a Search Problem. *Software, IEE Proceedings*, 150(3):161 – 175, 2003.

[8] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., Uni. of Washington Department of Computer Science and Engineering, 2000.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE '99*, pages 213–224. ACM, 1999.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering likely Program Invariants to Support Program Evolution. *IEEE Trans. on Software Engineering*, 27:213–224, 2001.

[12] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *ICSE 2000*, pages 449–458, 2000.

[13] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A Genetic Programming Approach to Automated Software Repair. In *GECCO '09*, pages 947–954, 2009.

[14] D. Gries. *The Science of Programming*. Springer, 1981.

[15] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. on Software Engineering*, PP(99), 2010.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[17] Y. Ma, J. Offutt, and Y. R. Kwon. MuJava: an Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[18] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[19] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[20] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu, 2008.

[21] D. Schuler, V. Dallmeier, and A. Zeller. Efficient Mutation Testing by Checking Invariant Violations. In *ISSTA '09*, pages 69–80. ACM, 2009.

[22] S. Wappler and J. Wegener. Evolutionary Unit Testing of Object-Oriented Software using Strongly-Typed Genetic Programming. In *GECCO '06*, pages 1925–1932. ACM, 2006.

[23] D. R. White, A. Arcuri, and J. Clark. Evolutionary Improvement of Programs. *IEEE Trans. on Evolutionary Computation*, PP(99):1–24, 2011.