

# Strengthening Inferred Specifications using Search Based Testing

Kamran Ghani, John A. Clark  
Department of Computer Science  
University of York  
YO10 5DD, York, UK  
{kamran,jac}@cs.york.ac.uk

## Abstract

*Software specification is an important element of the software development process. However, in most cases the specifications are out-of-date or even missing. One solution for this kind of problem is to use some process that infers the specification automatically. Work by Ernst et al [9, 22] has shown how specifications can be generated using program execution traces. These approaches are dependent on the test suites used to produce the traces, which may lead to unreliable specifications being inferred. Such specification inference is highly useful, however. In this paper we show how search based testing techniques can challenge and identify erroneous elements of such inferred specifications. This leads to a much tighter (accurate) inferred specifications. Thus, specification inference and search based test data generation are shown to be complementary.*

## 1 Introduction

The software specification captures the required behaviour of a program. It is an important ‘document’ used throughout the software development lifecycle. Specifications are informal plain text statements of needs, semi-formal structured or graphical descriptions, or statements with mathematical precision (usually referred to as formal specifications). Specifications may be identifiable documents in their own right, or else comprise assertion fragments embedded within code (e.g. as in the Design By Contract paradigm).

The extent to which a specification can be useful depends upon its specific form — each format has its own strengths and weaknesses. Formal specifications typically facilitate the automation of a variety of tasks (e.g. test data generation or proofs of correctness) but generally require a high level of skill to produce and read. Informal specifications are usable by a wider audience but may suffer from ambiguity.

In many cases we do not have any explicit specification. This is highly undesirable; specifications are highly useful documents to many stakeholders. Since generating and maintaining specifications is a tedious job, it can be greatly beneficial if the process is automated. Work has been done in this respect, exploring the use of static code analysis techniques (e.g. [4, 28]) and dynamic techniques (most prominently the work of Michael Ernst [9])

An invariant is a property of a program which remains true for all its executions and hence represents a partial specification. There are unlimited number of program invariants. Some will be fundamental (the set of invariants defining the program behaviour) and others may be derived as consequences.

Static analysis techniques for deriving invariants from program code are sound theoretically, but in practice they are difficult to implement. One recent approach used to overcome such problems is dynamic (runtime) analysis [9]. In this approach likely invariants are inferred from the actual execution traces of the program when exercised by test cases. Since the inferred invariants are largely dependent on test cases, they may not be correct. To get around this problem, many approaches have been proposed [22, 32, 6, 23]. An overview of these approaches is given in the Related Work section.

In this paper we present the use of Search Based Test Data Generation (SBTDG) techniques to verify and ‘strengthen’ the inferred putative invariants. An attempt is made to falsify inferred invariants using the available search techniques. The result is that an inferred specification can be iteratively challenged by SBTDG techniques resulting in a more credible set of invariants.

## 2 Background

### 2.1 Dynamic Invariant Generation

Dynamic Invariant Generation techniques have in recent years attracted significant research interest. As stated

above, data is collected from the state and IO execution traces of programs when test cases are run. Assertions can be generated over variables of interest. Assertions that are consistent with all traces are possible invariants. There are, of course, an infinity of such true statements but Michael Ernst et al. [9, 25] have demonstrated how *useful* invariants can be generated. In particular they have shown how invariants that make up the specification of a program can be generated. This “specification” is a best effort attempt to capture abstractly the behaviour of the implemented program. Daikon [10], their tool framework, uses a multi-step approach to inferring likely invariants. An instrumented version of a program (with code to record state data at program points during execution) is exercised with test cases. The likely invariants are detected and can be reported in many useful formats. The output consists of procedure pre- and post-conditions and generalized object invariants. Daikon checks for 75 different types of invariant and the extension mechanism is simple enough to include more. It also checks for conditional invariants and implications. A conditional invariant is only true part of the time. The statement

```
if (a < b)
  return a ;
else
  return b;
```

is an example of a conditional invariant. Support for many popular programming languages has already been provided and can be further extended easily to other languages.

## 2.2 Search Based Test Data Generation

In search based test data generation achieving a test requirement is modelled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of “guidance” to the search process via feedback from program executions. The guidance usually comes in the form of fitness or cost function, used interchangeably in this text.

For example, suppose we seek test data to satisfy the condition  $X \leq 20$ . We can associate with this predicate a cost that measures how close we are to satisfying it, e.g.  $cost(X \leq 20) = max(X - 20, 0)$ . The value  $X = 25$  clearly comes closer to satisfying the condition than does  $X = 50$ , and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost function  $cost(X \leq 20)$  over the range of possible values of  $X$ .

SBTDG for functional testing generally employs a search/optimisation technique with the aim of causing assertions at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to

be satisfied (or not) to achieve an identified execution path (or other goal); we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e. falsification testing — finding test data that breaks the specification) [29] ; or else we may simply require a proposed invariant to be falsified (e.g. breaking some safety condition [31] , or causing a function to be exercised outside its precondition).

There has been a growing interest in such techniques and we see more and more applications of these techniques to software testing. Some of the techniques that have been successfully applied to test data generation are Local Search (LS) [21, 16], Simulated Annealing (SA) [29], Genetic Algorithms (GA) [15, 2, 24], Tabu Search (TS) [8], Ant Colony Optimisation (ACO) [18], Artificial Immune Systems (AIS)[19], Estimation of Distribution Algorithms (EDA) [26], Scatter Search (SS) [27] and Evolutionary Strategies (ES) [3].

## 2.3 Fitness Function

We use the basic fitness function proposed by [30] for our work as shown in Table 1. This is a modified form of the work proposed by [16]. The fitness function is based on evaluating branch predicates. It gives a value of 0 if the branch predicate evaluates to the desired value and a positive value otherwise. The lower the value, the better is the solution. The table indicates the cost for specific assertions. Where more than one assertion is of interest (e.g. when a sequence of branch predicates must be satisfied to follow an identified path) then the basic costs per predicate are combined in some way. In the table,  $K$  represents a failure constant which is added to further punish incorrect test-data.

**Table 1. Fitness-Functions**

| Element      | Value#1   |
|--------------|---|
| $a = b$      | if $abs(a - b) = 0$ then 0<br>else $abs(a - b) + K$ |
| $a \neq b$   | if $abs(a - b) \neq 0$ then 0<br>else $K$           |
| $a < b$      | if $(a - b) < 0$ then 0<br>else $(a - b) + K$       |
| $a \leq b$   | if $(a - b) \leq 0$ then 0<br>else $(a - b) + K$    |
| $a > b$      | if $(b - a) < 0$ then 0<br>else $(b - a) + K$       |
| $a \geq b$   | if $(b - a) \geq 0$ then 0<br>else $(b - a) + K$    |
| $a \vee b$   | $min(cost(a), cost(b))$                             |
| $a \wedge b$ | $cost(a) + cost(b)$                                 |

## 2.4 SBTDG for Invariant Falsification

SBTDG techniques find input data that cause identified assertions to be true or false (as required). For specification strengthening purposes we simply target those assertions generated by dynamic techniques such as Daikon as fragments of inferred specifications. We need only represent such assertions in a form that is amenable to search. This can be readily obtained by representing an invariant in branch predicate form. Suppose we have a proposed invariant  $inv$  for some identified point in the program. If we insert a program statement “if(! $inv$ );” at that point we can view the falsification of the invariant as a branch reachability problem. This enables us to use the standard SBTDG techniques, where we try to find test data to satisfy this branch. In this work we have used our own SBTDG tool, but the overall approach we adopt can readily be used with any reasonably effective SBTDG tool.

## 3 Related Work

Nimmer et al. [22] proposed a combination of dynamic and static analysis techniques to generate specifications and prove their correctness. They used Daikon with ESC/Java [7, 17]. Their work shows that specifications generated from program execution are reasonably accurate. However, due to limitations of the tools spurious inferred invariants may remain.

Harder et al. [14] proposed the Operational Difference (OD) technique for generating, augmenting, and minimizing the test suites. The main idea is to generate an *operational abstraction* (OA), an abstraction of the program’s runtime behaviour, from program executions and then try to improve it. The technique starts with an empty test suite and empty OA. Test cases are generated and evaluated by means of the change that it brings in the OA. A test case that improves the OA can be added to a test suite and a test case that doesn’t can be removed. They also proposed *operational coverage* as a measure of difference between the OA and the correct specification. This is a relative term and requires the presence of an oracle to be computed. The technique developed fault revealing test suites, however, it is not guaranteed that the change brought by a test case in the OA is correct. For example, consider a variable  $A \geq 0$ . A test case, say  $A = 5$ , may cause the OA to include  $A \neq 0$  unless and until another test case  $A = 0$  is executed. Thus elements of the OA may be untrue. Our technique on the other hand searches for such “missing” or revealing test cases and hence increases the quality of the OA.

Gupta [11] proposed modelling the invariant detection problem as a test data generation problem in a manner similar to our approach. However, they did not apply search based techniques. They [12] further proposed an invariant

coverage criterion based on a set of definition-use chains of variables of an invariant property.

Hangal [13] and Xie [32] used specification violation approaches to improve their inferred specifications. Hangal’s work [13] was mainly aimed at detecting bugs. It is implemented in the DIDUCE tool, which continually checks the program’s behaviour against the invariants during program execution and reports all detected violations at the end. Xie’s [32] approach uses Daikon with ParaSoft JTest [1] with the intention of improving the test suites for unit testing of Java programs.

Pacheco [23] proposed a technique that selects from a large test set, a small subset of test inputs that are likely to reveal faults in the software under test. Their technique infers an operational model of the software’s operation from the correct execution of a program using the Daikon invariants detector. The program is then executed using randomly generated ‘candidate’ inputs with provided inferred invariants monitored to see if they are satisfied. A classifier system labels candidate input as *illegal*, *normal operation* or *fault-revealing*. The interesting ones are the fault-revealing inputs which may indicate a fault in the program. If more than one input violates the same property then only one input is selected from that set. The technique has been implemented in a tool called as *Eclat*. The technique was tested with some sample programs and compared with another tool, JCrasher [5], *Eclat* yields good results. The technique is effective and revealed previously unknown faults. However, it requires a priori correct test suites which may not be available. It also sometimes classifies an input as fault-revealing, though it may not be so.

## 4 Our Approach

One limitation of many dynamic invariant inference approaches is that they are at the mercy of the test data. Test suites may not be wholly appropriate for the purposes of inference. We may also get spurious or ‘not interesting’ [11] invariants. Though some approaches [11, 14, 32] have been proposed to modify test suites, they do not completely preclude the generation of spurious invariants. This is especially the case when the domain of input data is large or the program is complex. We may have a test suite satisfying a certain structural criterion, but due to lack of enough test cases, we may get erroneous specifications. Figure 1 is an example of such a program. In order to refine the specification for such programs we adopt a systematic approach.

Our approach is similar to that of [13, 32]. However, we propose a Search Based Test Data Generation approach and unlike previous approaches, our domain of application is more ‘procedural’ in nature.

```

public class Prog{
    private int b = 10;
    private int c = 30;
    public void test(int a){
        b = 5 * a;
        c = 100000 - 5 * a;
    }
}

```

**Figure 1. A simple program with a large input space**

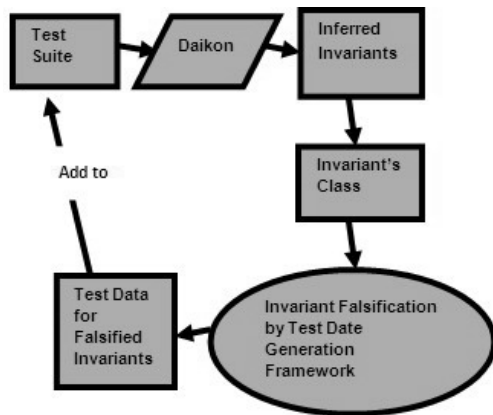
#### 4.1 High Level Model

Figure 2 shows a high level model of our approach. We can divide our model into three main steps.

*Invariant Inference:* Invariants are generated from a randomly generated test suite containing ‘enough’ test cases to allow a reasonably succinct set of invariants to be inferred.

*Invariants’ Class:* The invariants generated are then ‘imported’ into an intermediate class. It contains all the invariants in a format suitable to apply our search based test data generation techniques.

*Invariant Falsification:* A test data generation tool attempts to falsify the invariants.



**Figure 2. High Level Model of our Approach**

These are further elaborated below.

#### 4.2 Invariant Inference:

In this step we use Daikon to infer likely invariants. To begin with, a test suite of ‘reasonably’ large set of test cases is given as input to Daikon. Daikon dynamically infers a list of invariants based on the execution of test cases. The test suite is either generated randomly or by our test data

generation tool, described later. We can ensure that identified structural criteria are satisfied, whichever approach is adopted. Though the inferred invariants are affected by the number of test cases in a test suite, there is no direct correlation between the size of test suite and the invariant inference process [14]. Therefore, we chose a test suite of arbitrary size, ensuring at the same to include enough test cases that may satisfy the structural criteria many times. The reason for this is to get a list of generalized likely invariants and to eliminate the less interesting ones e.g. invariants peculiar to a specific test set used, such as “x is one of {4, 8, 9}”.

Consider the running example in Figure 1. We used Daikon to infer invariants using 15000 randomly chosen values of “a” between -100 and 100000. The output from Daikon is shown in Figure 3. The invariant  $a! = 0$  is not necessarily true, but since the probability of randomly choosing  $a = 0$  as an input is  $1/101000$ , we can see why a random process is unlikely to generate such a case (even with 15000 trials). 15000 trials would generate at most 15000 distinct values of “a” and the range of “a” comprises 101000 of which 0 is but one element. Thus in random test generation techniques, which seem to be the choice for invariants falsification in previous approaches [13, 32], we are very likely to get the spurious invariants. Note that stronger coverage criteria, such as *Invariant Coverage* are also not effective here as a single test case, e.g.  $a = 2$  may give us the coverage for all these criteria.

#### 4.3 Intermediate Invariants’ Class

For counter-example generation purposes we insert additional fragments into the program. At point *P* in the program if an invariant such as  $(a! = x)$  is proposed then we insert a code fragment *if*  $(a == x)$ ; In test data generation terms breaking the invariant corresponds to reaching the true branch of the inserted fragment. Thus we model breaking invariants as path satisfaction problems. Fragments are inserted for each invariant we want to falsify.

#### 4.4 Test Data Generation Framework

The generated class is then given as an input to the test data generation framework, which searches data to falsify the invariants. The test data generation framework is a research tool developed to investigate the application of search based techniques to software testing. We can use any search based technique such as hill climbing, simulated annealing (SA) or evolutionary search techniques. We used SA here to falsify the invariants.

If an invariant can not be falsified even after exhausting the search criteria, the search is terminated for that invariant and the invariant is assumed correct. Here we use the tool’s abilities to target branch conditions. We terminate the

search process when either a solution is found or a maximum numbers of trials have been made. If a test case is found that falsifies the invariant, it is added to the existing test suite. The process continues until test data to falsify all the invariants have been searched for. A report is then generated, which shows the invariants that have been falsified and the respective test data. Figure 4 shows such a report for the example program. The modified test suite is again executed by Daikon and a modified list of invariants is generated. The process is repeated and if no further invariant is falsified in the following iteration, the process is terminated. For the example program, all the falsified invariants were found in the first iteration thus giving us more refined specifications. as shown in figure 5.

```

Prog::OBJECT
this has only one value
this.b != 0
this.c != 0
this.b != this.c
=====
Prog.Prog()::EXIT
this.b == 10
this.c == 30
=====
Prog.test(int)::ENTER
this.b == 0 (mod 5)
this.c == 0 (mod 5)
a != 0
this.b != a
this.c != a
=====
Prog.test(int)::EXIT
this.b == 0 (mod 5)
this.c == 0 (mod 5)
this.b + this.c - 100000 == 0
this.b != orig(this.b)
this.b != orig(this.c)
this.b != orig(a)
this.b - 5 * orig(a) == 0
this.c != orig(this.b)
this.c != orig(this.c)
this.c != orig(a)
this.c + 5 * orig(a) - 100000 == 0

```

Figure 3. Initial Output from Daikon.

## 5 Case studies

For further experiments we used four programs, i.e, Middle, WrapRoundCounter and BubbleSort from [29]. These programs were used for specification conformance and were originally written in Ada. The fourth program, CalDate, is taken from [19]. These programs have been given in the appendix. Details of the input domains, test cases and numbers of falsified invariants are shown in the Table 2.

```

Invariant (this.c!=0) falsified by a=30
Invariant (this.b!=0) falsified by a=10
Invariant (a!=0) falsified by a=0
Invariant (this.b!=orig(this.c)) falsified by a=6
Invariant (this.b!=orig(this.b)) falsified by a=2
Invariant (this.b!=orig(this.a)) falsified by a=0
Invariant (this.c != orig(this.c)) falsified by a=19994
Invariant (this.c != orig(this.b)) falsified by a=19998

```

Figure 4. Output from Test Data Generation Framework

### 5.1 Middle Program

Middle program takes 3 input variables and returns a variable having a value between the other two. If two variables have same value then the third one is reported as middle. We tried programs with different input domains. Usually, if we keep the input domain small, we don't expect spurious invariants to be generated. However, in this case even though we kept the input domain very small, still a spurious invariant was nevertheless inferred. By further increasing the input space, Daikon inferred a set containing no spurious invariant. However, when the domain of input values was further increased more spurious invariants were inferred by Daikon. In each case the search based test data generation framework successfully generated test data to falsify these spurious invariants and the final lists in all cases contained the same invariants.

### 5.2 WrapRoundCounter

WrapRoundCounter is a simple program which counts from 0 to 10 and then wrap-around back to 0 again. This program was included to investigate conditional invariants as well. No inferred invariant was falsified by the search tool in this case as the program was simple enough for the Daikon to infer correct invariants. Also there was no effect changing the input domain because of the nature of program.

### 5.3 BubbleSort

This program sorts the elements of an array in ascending order. This is an example of program where the input domain does not have an effect on the inferred invari-

```

Prog::OBJECT
this.c != 0
this.b != this.c
=====
Prog.Prog()::EXIT
this.b == 10
this.c == 30
=====
Prog.test(int)::ENTER
this.b == 0 (mod 5)
this.c == 0 (mod 5)
this.c != a
=====
Prog.test(int)::EXIT
this.b == 0 (mod 5)
this.c == 0 (mod 5)
this.b + this.c - 100000 == 0
(orig(a) == 0) ==> (this.b == 0)
(this.b == 0) ==> (orig(a) == 0)
this.b - 5 * orig(a) == 0
this.c != orig(a)
this.c + 5 * orig(a) - 100000 == 0

```

**Figure 5. Final Output from Daikon.**

ants. Daikon inferred a correct set of invariants in this case. Additionally, the manual instrumentation of the code with Daikon invariants was very tedious.

#### 5.4 CalDate

This program converts the date given as day, month and year into a Julian date (not including the fractional time part). The Julian date is then calculated as the number of days from the 1st January, 4713 B.C. If the date to be converted is after 15th October 1582, which is the date of introduction of the Gregorian calendar, then the Julian date is adjusted. The Julian date is then returned. In this case we changed the number of test cases rather than the domain as the domain needs to remain fixed. The search tool was able to eliminate the spurious inferred invariants in each case. However, as expected, the number of spurious invariants decreases with the test suite size. Again the final set of invariants were identical in each case.

### 6 Conclusion and future work

Specifications are important. In many cases, they will not exist or (often even worse) be out of date. Specification synthesis tools such as Daikon offer a promising solution to this problem. However, Daikon, and indeed other dynamic inference tools, make inferences based on the traces of the program when executed with a given test set. Inferred specifications may differ between test sets used; this presents a problem. Coverage criteria may be postulated but there

**Table 2. Case Studies**

| Program                         | Input Domain   | # of Test cases | # of Falsified Invariants |
|---------------------------------|----------------|-----------------|---------------------------|
| Middle                          | -10 — 10       | 15000           | 1                         |
| Middle                          | -100 — 100     | 15000           | 0                         |
| Middle                          | -500 — 500     | 15000           | 1                         |
| Middle                          | -1000 — 1000   | 15000           | 2                         |
| Middle                          | -5000 — 5000   | 15000           | 2                         |
| BubbleSort                      | -100 — 100     | 15000           | 0                         |
| BubbleSort                      | -1000 — 1000   | 15000           | 0                         |
| WrapRound-Counter               | 0 — 11         | 100             | 0                         |
| WrapRound-Counter               | 0 — 100        | 100             | 0                         |
| WrapRound-Counter (conditional) | 0 — 11         | 100             | 0                         |
| Example-Program                 | -1000 — 100000 | 15000           | 8                         |
| CalDate                         | various        | 500             | 3                         |
| CalDate                         | various        | 1000            | 3                         |
| CalDate                         | various        | 15000           | 1                         |

would appear to be no clear candidate. In many cases users resort to random testing whilst in others structural criteria are used. These often allow erroneous invariants to be inferred. Our work shows that extant search based software test data generation approaches can be used to stress each inferred invariant with a view to falsifying it. Test data that falsifies inferred invariants can be added to the test suite and the inference tool can be rerun. This leads to more accurate inferred specifications. Currently the search for test data is carried out automatically by our tool. Instrumenting the program with the Daikon invariants to be falsified is at present a manual process though this should shortly be automated.

Our technique uses our own Java search based test data generation tool but we think the approach can clearly be adopted with any other search based test data generation tool. The approach shows that search based test data generation and specification inference are complementary. (Search based test data generation techniques can also be used to generate the initial test suite itself.) Our approach is a pragmatic way to enable the important task of specification inference to be improved.

### References

- [1] Jtest, ParaSoft Corporation, <http://www.parasoft.com>.

- [2] J. Alander, T. Mantere, and P. Turunen. Genetic algorithm based software testing. In *Artificial Neural Nets and Genetic Algorithms*, pages 325–328, Wien, Austria, 1998. Springer-Verlag.
- [3] E. Alba and J. Chicano. Software testing with evolutionary strategies. *LNCS 3943*, 169(2):50–65., September 2005.
- [4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages POPL'78*, Tuscan, Arizona United States, January, 1978.
- [5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Software Practice and Experience*, 34(11):1025–1050, Sept. 2004.
- [6] T. Denmat, A. Gotlieb, and M. Ducass. Proving or disproving likely invariants with constraint reasoning. In *Proceedings of the 15th Workshop on Logic-based Methods in Programming Environments (WLPE'05)*, Sitges Barcelona, Spain, October 5 2005.
- [7] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159 SRC-RR-159, Compaq Systems Research Center, December 1998.
- [8] E. Diaz, J. Tuya, and R. Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 310 – 313, Oct. 2003.
- [9] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, August 2000.
- [10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.
- [11] N. Gupta. Generating test data for dynamically discovering likely program invariants. In *Proceedings of Workshop on Dynamic Analysis (WODA 2003)*, pages 21–24, Portland, Oregon, May 9, 2003.
- [12] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8-10, 2003.
- [13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.
- [14] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, 2003.
- [15] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.
- [16] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16:8:870–879, 1990.
- [17] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/java users manual. Technical report 2000-002, Compaq Systems Research Center, Palo Alto, California, October 12 2000.
- [18] H. Li and C. P. Lam. Software test data generation using ant colony optimization. In *Proceedings of the International Conference on Computational Intelligence*, pages 1–4, 2004.
- [19] P. May. *An Artificial Immune System Approach to Mutation Testing Test Data Generation*. PhD thesis, The University of Kent at Canterbury, 2007.
- [20] H. M. Deitel and P. J. Deitel. *Java How to Program 6/e*. Prentice Hall, 2005.
- [21] W. Miller and D. Spooner. Automatic generation of floating point test data. *IEEE Transactions on Software Engineering*, 2(3):223226, September 1976.
- [22] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22-24, 2002.
- [23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the ECOOP 2005 19th European Conference Object-Oriented Programming.*, pages 504–527, Glasgow, Scotland, July 27-29, 2005.
- [24] R. Pargas, M.J. Harrold, and R. Peck. Test data generation using genetic algorithm. *Journal of Software Testing, Verification, and Reliability*, 9(3):263–282, September 1999.
- [25] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 23–32, New York, NY, USA, 2004. ACM.
- [26] J. L. Sagarna. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence.*, 19(5):457–489., 2005.
- [27] J. L. Sagarna. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412., 2006.
- [28] P. H. Schmitt and B. Wei. Inferring invariants by symbolic execution. In *Proceedings of the 4th International Verification Workshop (VERIFY'07)*, Bremen, Germany, 2007.
- [29] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing'. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 73 – 81, Clearwater Beach, Florida, United States, 1998.
- [30] N. Tracey, J. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *Proceedings of the Automated Software Engineering (ASE'98)*, pages 285–288, 1998.
- [31] N. J. Tracey. *A Search-Based Automated Test-data Generation Framework for safety-critical Softwares*. PhD thesis, The University of York, 2000.
- [32] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8-10 2003.

## A Programs used for experimentation

### A.1 Middle

```
public class Middle {
    public int findMiddle(int a, int b,
        int c){

        if((a<b && b<c)|| (c<b && b<a)){

            return b;
        }
        else if((a<c && c<b)|| (b<c
            && c<a )){

            return c;
        }
        else if((b < a && a < c) ||
            (c<a && a<b) ){

            return a;
        }
        else if (b==c){

            return a;
        }
        else if (b==a){

            return c;
        }
        else {

            return b;
        }
    }
}
```

### A.2 WrapRoundCounter

```
public class WrapRoundCounter {

    public int wrap_inc(int n){
        if (n>10)
        {
            n=0;
            return n;
        }
        else
        {
            n=n+1;
            return n;
        }
    }
}
```

### A.3 BubbleSort

```
//Modified form of program taken
//from chapter 7 of [20]

public class BubbleSort{

    public void sort( int ar[] )
    {
        for(int i=1;i<ar.length;i++) {
            for ( int element = 0;
                element < ar.length-1;
                element++ ) {
                if (ar[element]>ar[element+1])
                    swap(ar,element,element+1);
            }
        }
    }

    public void swap( int ar[], int first,
        int second )
    {
        int hold;
        hold = ar[ first ];
        ar[ first ] = ar[ second ];
        ar[ second ] = hold;
    }
}
```

### A.4 CalDate

```
import java.lang.Math;
import java.io.*;
public class CalDate {
    public double toJulian ( int day ,
        int month , int year ) {
        int JGREG = 15+31*(10+12*1582);
        double HALFSECOND = 0.5;
        int julianYear = year;
        if ( year <0){
            julianYear = julianYear+1;
        }
        int julianMonth = month;
        if (month>2){
            julianMonth = julianMonth+1;
        } else {
            julianYear = julianYear-1;
            julianMonth = julianMonth+13;
        }
        double t = Math.floor(365.25*
            julianYear );
        double s = Math.floor (30.6001*
            julianMonth );
        double julian = t+s+day+1720995.0;
        int temp = day+31*(month+12*year);
        if ( temp>=JGREG){
            int ja = (int) (0.01*julianYear);
            julian = julian+2-ja +(0.25*ja);
        }
        System.out.println(julian);
        return Math.floor(julian );
    }
}
```