Improving Evolutionary Testing by Flag Removal

Mark Harman, Lin Hu and Robert Hierons
Brunel University,
Uxbridge, Middlesex, UB8 3PH, UK
Mark.Harman@brunel.ac.uk

André Baresel and **Harmen Sthamer**
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, D-10559 Berlin
Andre.Baresel@daimlerchrysler.com

Abstract

This paper argues that Evolutionary testing can be improved by transforming programs with flags into flag free programs. The approach is evaluated by comparing results from the application of the Daimler-Chrysler Evolutionary Testing System to programs with flags and their transformed flag-free counterparts. The results of this empirical study are very encouraging. Programs which could not be fully covered become fully coverable and the number of generations required to achieve full coverage is greatly reduced.

1 INTRODUCTION

Evolutionary testing generates test data to cover certain structural program features, using evolutionary algorithms to search the space of possible program inputs. Evolutionary testing has been shown to be an effective way of automatically generating test data for white box (or structural) test adequacy criteria [13, 21, 19, 15, 11]. The approach works well for well-behaved programs, but for certain programming language features the approach performs poorly.

One such problem arises with programs which use flag variables. A flag variable is one whose value is either `true` or `false`. Flags typically 'flag' the presence of some special condition if interest. The use of flag variables with current approaches to fitness function definition, yields a coarse fitness landscape with a single super-fit plateau and a single super-unfit plateau (corresponding to the two possible values of the flag variable). This causes the search to degenerate to a random search. Where the super-fit plateau is small, such a random search fails to find suitable test data, reducing the coverage achieved by the approach.

Embedded systems, such as engine controllers, typically make extensive use of flag variables to record state information concerning the devices controlled. Such systems can therefore be hard to test using evolutionary testing approaches to automated test data generation. This is a serious problem, since generating such test data by hand is prohibitively expensive, yet the correct operation of such embedded systems is clearly of paramount importance.

This paper presents a transformation-based approach, which addresses the problem. The approach allows certain forms of commonly arising flags to be transformed out of the program, thereby dramatically improving the results of evolutionary testing. The rest of the paper is organised as follows: Section 2 gives a brief overview of evolutionary testing, while Section 3 explains the flag problem. Section 4 describes our solution to the flag problem and Section 5 presents and discusses the results of applying this solution to typical flag-based programs.

2 APPLYING EVOLUTIONARY ALGORITHMS TO SOFTWARE TEST DATA GENERATION

Evolutionary testing designates the use of metaheuristic search methods for test case generation. The input domain of the test object forms the search space in which one searches for test data that fulfill the respective test goal. Due to the non-linearity of software (if-statements, loops etc.) the conversion of test problems to optimisation tasks mostly results in complex, discontinuous, and non-linear search spaces. Neighbourhood search methods like hill climbing are not suitable in such cases. Therefore, metaheuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing or tabu search. In this work, evo-

lutionary algorithms will be used to generate test data, since their robustness and suitability for the solution of different test tasks has already been proven in preceding work [13, 21]. The only prerequisites for the application of evolutionary tests are an executable test object and its interface specification. In addition, for the automation of structural testing, the source code of the test object must be available to enable its instrumentation.

In order to automate software tests with the aid of evolutionary algorithms, the test goal must itself be transformed into an optimisation task. For this, a numeric representation of the test goal is necessary, from which a suitable fitness function for evaluation of the generated test data can be derived. Depending on which test goal is pursued, different fitness functions emerge for test data evaluation. For structural testing the fitness functions can be based on computation of a distance for each individual that indicates how far it is away from executing the program predicate in the desired way [13, 21, 19].

For example, if a branching condition `x==y` needs to be evaluated as `true`, then the fitness function may be defined as $\mid \mathtt{x} - \mathtt{y} \mid$ (provided that the fitness values are minimised during the optimisation). Each individual of the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the fitness value is determined for the corresponding individual.

The approach adopted in the work reported in the present paper, used the DaimlerChrysler Evolutionary Testing System. Multiple strategies and competitions between these were used. All experiments used a population of 300 individuals split into 6 subpopulations of 50 individuals. In order to combine the multiple strategies, migration was introduced to permit an exchange of the best individuals between subpopulations at regular intervals. The details of the implementation of the evolutionary approach to software testing are described elsewhere [13, 21, 19, 20]. In the present paper, the focus is upon the way in which transformation can be used to improve the behaviour of these established techniques.

3 THE FLAG PROBLEM

A flag variable will be taken to mean any variable, the type of which is boolean, but the transformations presented here may well extend to other variables which are assigned one of a small number of possible scalar values.

Generating test data using evolutionary testing [10, 20, 19, 15, 11] has been shown to be successful. However, evolutionary testing relies upon a fitness function which uses the predicate which controls a branch. Where such a predicate is simply a reference to a flag variable, the search has little information to guide it, making the evolutionary technique perform poorly. More precisely, using an evolutionary algorithm, the presence of flag variables (and unordered enumeration types in general) can create a coarse fitness landscape.

This reduces the effectiveness of the search. That is, the fitness landscape consists of two plateaus, corresponding to the two possible flag values. One of these plateaus will be super-fit and the other super-unfit. A search-based approach, such as evolutionary testing, will not be able to locate the super-fit plateau any better than a random search, because the fitness landscape provides no guide to direct the search from unfit to fit regions of the landscape. Where the fit plateau may be very small relative to the unfit plateau, this makes the program hard to test. A similar problem is observed with n–valued enumeration types, whose fitness landscapes contains n discrete values, as n becomes larger the program becomes progressively more testable, as the landscape becomes progressively more smooth and therefore, more guidance is available.

Figure 1 illustrates the transformation approach to the flag variable problem. The original program (in column (a)) is hard to test using currently defined fitness functions for evolutionary testing. The first dotted section indicates code which does not assign to `n`, the second dotted section of code does not assign to `flag`. Suppose `n` is an unsigned integer value. The value of `n` required to cause the second conditional to follow the `true` branch must be odd and less than four, namely it must be either 1 or 3. Random testing is very unlikely to ‘stumble’ across these two values, so a more intelligent search is required. This is where evolutionary testing could help. Unfortunately, the presence of the flag variable inhibits the search, because the fitness landscape is insufficiently smooth to guide the search. Therefore, it is difficult to cover both branches of the final `if` statement.

4 A TRANSFORMATION-BASED SOLUTION

A program transformation [5, 16, 17, 2] is a rule which defines the way in which a program can be modified. It can be thought of as a function from program syntax to program syntax. Some transformation rules have side conditions. These are conditions which must be true for the transformation to be correct. As a simple ex-

flag = n<4;
...
if (n%2==0) flag = 0;
 	
...
if (a[i]!='0' && flag)
 	
...
	
	...
	flag=(n%2==0)?0:(n<4);
	
	...
	if (a[i]!='0' && flag)
	
	...
		
		...
		n' = n;
		flag=(n'%2==0)?0:(n'<4);
		...
		if (a[i]!='0' && flag)
		
		...
			
			...
			n' = n;
			flag=(n'%2==0)?0:(n'<4);
			...
			if (a[i]!='0' &&
			(n'%2==0)?0:(n'<4))
			...

(a) Original	(b) Single flag assignment	(c) Independent Assignment	(d) Flag removed

Figure 1: Flag Removal

ample, consider the simple transformation rule 'reverse if' which reverses the branches of an `if-then-else` statement, and negates the predicate. This transformation produces an equivalent program while altering the structure of the original. Such a transformation will be denoted like this[1]:

$$\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2$$
$$\Rightarrow$$
$$\texttt{if not(}E\texttt{) then } S_2 \texttt{ else } S_1$$

Many transformation rules are extremely simple. On their own they achieve little of value. However, when combined into sequences of transformations, or into mini-programs, called transformation tactics, the combined effect on the program under consideration can be startling. A set of transformation tactics is typically collected together into a transformation strategy; an algorithm for manipulating the subject program into a semantically equivalent, but more syntactically and structurally amenable form.

Transformation has been applied to many problems including automatic parallelization [12, 23] program comprehension [3, 18, 9], reverse and re-engineering [16] and efficiency improvement [1]. In this paper transformation will be used to remove the flag problem, by transforming predicates which contain flags into flag-free predicates.

Our approach to the flag problem will be to transform a flag-based program into an equivalent flag free version. The transformations we use will preserve the branches of the original program, so that test data will achieve branch coverage for the original program if and only if it does so for the transformed program. This allows us to replace the (harder) problem of generating test data for the flag-based program with the (easier) problem of generating test data for the transformed, flag-free version. Once we have generated the test data (using the transformed program) we have no further need of the transformed flag-free program, and it is discarded. This application of program transformation differs from conventional transformation in this regard: For us, transformation is a means to an end, rather than an end in itself. The transformed program is of use only to the evolutionary testing system and is never presented to the human.

Consider, the program from Figure 1. The version in column (d) is equivalent, but easier to test because the use of the flag variable `flag` has been replaced by an expression which denotes its value at the point of use. Thus, this version of the program produces a smoother fitness landscape at the second predicate, thereby guiding the search toward the two fittest values sought. Columns (b) and (c) show the intermediate transformation steps required to reach the result in column (d). In column (b) assignments to the flag variable have been collected together. In column (c) a temporary variable, n', is used to capture the current value of the variable `n`. Finally, in column (d) the expression denoting the value of `flag` is substituted for the single use which it reaches.

Unfortunately, space restrictions prevent a full treatment of the flag removal algorithm. Figure 2 presents a sketch of the algorithm. One transformation step that we found particularly useful is that known as 'program slicing' [22, 14, 6, 8], and in particular its amorphous formulation [7]. Slicing removes parts of the program which cannot affect a particular variable of interest. Slicing is useful in flag removal, because it allows us to isolate the code that captures the computation on the flag variable.

5 RESULTS

The DaimlerChrysler Evolutionary Testing system was used to generate test data for flag-based programs and these results were compared with those obtained from running the testing system with identical parameters on the transformed, flag-free versions of the programs.

[1] In general, this paper will adopt the convention that $A \Rightarrow B$ denotes the fact that code fragment A can be transformed to code fragment B.

In this section we present three indicative experiments, which illustrate various incarnations of the flag problem and the effect upon evolutionary test data generation of their removal. The figures show the results obtained on the left hand side against the relevant fragments of the corresponding programs on the right-hand side. The program fragments are shown to illustrate the particular flavour of flag problem considered. However, when using the system, the human need not be aware either of the flag-free version of the program, nor indeed of the evolutionary process itself. The user simply submits a program (possibly with flags) and obtains a set of optimised test data.

The results plot the coverage achieved (for six separate executions of the evolutionary testing systems) against the number of fitness evaluations. They are therefore a measure of effectiveness against effort.

A test goal consists of attempting to optimise test data to cover a particular branch. The coverage for each trial therefore increases in steps, as each test goal is satisfied. In all examples we present, a test set which achieves full branch coverage *exists* (there are no infeasible branches).

5.1 Triangle Classification Program

The classify triangle program is widely used as a benchmark in software testing. The program has three variables (`a`, `b` and `c`), which represent the side lengths of a figure. The goal of the program is to determine whether the three side lengths represent a triangle, and if they do, to categorise the triangle type.

Input values are `double` values within range -1000 to 20000 with a precision of 0.00001. This gives a search space of size of approximately 10^{27}. We experimented with two versions of the a 'Validity check' program and a 'Special Value' program. These two variants of the triangle program illustrate the range of difficulty introduced by flags from none (Validly Check) through to severe (Special Value). The results for each variant are shown in Figures 3 and 4.

In the 'Validity Check' variant, the flag is assigned a value which represents a set of validity checks on inputs. There are many sub-criteria (boolean terms), many inputs which satisfy each sub-criteria and many which fail to satisfy each. Therefore, the fitness landscape does not contain a small high fitness plateau. Furthermore, each of the sub-criteria is also checked later on in the program by a separate conditional and so each sub-criteria also forms a separate test goal. In this situation the presence of flags presents no difficulty.

By contrast the 'Special Value' variant of the triangle program represents the *worst* form of flag-based program. The flag variable is set to true by only very few inputs, creating a tiny plateau of high fitness. Furthermore, the sub-conditions mentioned in the boolean expression assigned to the flag variable are not tested anywhere else in the program. In such a situation evolutionary testing degenerates to random testing.

The results show that for the 'Validity Check' version of the program, the removal of flags makes practically no difference, with all trials reaching maximum fitness, and with all doing so with a similar spread of effort. On the other hand, the 'Special Value' variant shows how bad the flag problem can be. After 40,000 fitness evaluations, none of the trail runs has risen above a coverage of .86 and after 120,000 evaluations none has risen above 0.92. No trial reached the maximum possible fitness. However, for the flag free version, after only 25,000 fitness evaluations, *all* of the trail runs has reached a coverage of more than 0.86 and after only 80,000 evaluations all have reached maximum possible coverage (1.0).

5.2 Calendar Program

The calendar program computes dates, but takes account of special days and date corrections which have taken place throughout the centuries. These special dates are denoted by flags in the program.

In 1751, the British Parliament passed

> "An Act for Regulating the Commencement of the Year, and for Correcting the Calendar Now In Use."[4]

The act became known as the 'Calendar Act'. One of the aspects of this act was that the date of September the 2nd, the following year was to be immediately followed by September the 14th. An decision which caused much consternation and a demand for the return of the 'stolen 11 days'. These stolen days form a special case in the calendar program which is denoted by a flag.

The calendar program is a typical flag-based program which tests for an 'unusual' condition and sets the value of a flag according to this test. This is typical because flags often test for exceptional cases. That is, the value assigned is far more likely to take one of the two possible values than the other (because the condition tested is 'unusual'). In this case, the program contains 10 character variables, which take values within range 0 to 10. This gives a search space of approximately 10^{10}, with a flag representing the 11 stolen days.

The results of evolutionary test data generation for the calendar program, together with the relevant fragments of code are shown in Figure 5. The flag-free code has been simplified for readability. The actual transformed program produced by the flag-removal algorithm contains many temporary variables. Of course, the fact that the transformed version has poor readability is not an issue for this work (unlike most work on transformation) because the transformed program is not read by a human.

The results show that flag removal helps in this instance, because all of the runs achieve the maximum possible fitness using the flag-free version, while none does so using the original program. It can also be seen from the growth of coverage for each run, that using the flag-free version we obtain higher coverage, faster than using the flag-based version.

6 CONCLUSION

This paper has introduced a transformation-based approach which improves evolutionary testing in the presence of flag variables. Flag variables inhibit the successful application of evolutionary techniques to automated structural software test data generation. The transformation based approach, removes the reliance upon flag variables, thereby improving both the time to produce test data and the coverage achieved.

Flag variables are very common in embedded systems. The correct behaviour of these systems is also a paramount concern because they control real-world devices, the failure of which can lead to severe consequences. The results presented show that flag removal works well, when applied to typical flag-based programs.

7 ACKNOWLEDGEMENTS

The authors benefitted greatly from discussion of this work with members of the EPSRC-funded SEMINAL network.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[2] Baxter, I. D. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse* (Austin, TX, USA, Jan. 1999).

[3] Bennett, K., Bull, T., Younger, E., and Luo, Z. Bylands: reverse engineering safety-critical systems. In *IEEE International Conference on Software Maintenance* (1995), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 358–366.

[4] Calendar Act. Calendar Act, Anno vicesimo quarto George II, cap. xxiii., 1751.

[5] Darlington, J., and Burstall, R. M. A tranformation system for developing recursive programs. *J. ACM 24*, 1 (1977), 44–67.

[6] De Lucia, A. Program slicing: Methods and applications. In 1st *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[7] Harman, M., and Danicic, S. Amorphous program slicing. In 5th *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[8] Harman, M., and Hierons, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[9] Harman, M., Hu, L., Zhang, X., and Munro, M. Side-effect removal transformation. In 9th *IEEE International Workshop on Program Comprehension (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 310–319.

[10] Jones, B., Sthamer, H.-H., and Eyres, D. Automatic structural testing using genetic algorithms. *The Software Engineering Journal 11* (1996), 299–306.

[11] Pargas, R. P., Harrold, M. J., and Peck, R. R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9* (1999), 263–282.

[12] Ryan, C., and Walsh, P. The evolution of provable parallel programs. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 295–302.

[13] Sthamer, H. *The Automatic Generation of Software Test Data Using Genetic Algorithms.* PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[14] Tip, F. A survey of program slicing techniques. Tech. Rep. CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[15] Tracey, N., Clark, J., and Mander, K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)* (January 1998), IFIP, pp. 169–180.

[16] Ward, M. Reverse engineering through formal transformation. *The Computer Journal 37*, 5 (1994).

[17] Ward, M., and Bennett, K. A practical program transformation system. In *Working Conference on Reverse Engineering* (Baltimore, MD, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 212–221.

[18] Ward, M., Calliss, F. W., and Munro, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[19] Wegener, J., Baresel, A., and Sthamer, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43*, 14 (2001), 841–854.

[20] Wegener, J., and Grochtmann, M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems 15*, 3 (1998), 275 – 298.

[21] Wegener, J., Sthamer, H., Jones, B. F., and Eyres, D. E. Testing real-time systems using genetic algorithms. *Software Quality 6* (1997), 127–135.

[22] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[23] Williams, K. P. *Evolutionary Algorithms for Automatic Parallelization.* PhD thesis, University of Reading, UK, Department of Computer Science, Sept. 1998.

The essential aim of the algorithm, is to reduce a program with flags into one with a single assignment to the flag variable, which can be substituted for the use, within a predicate under test. For example:-

```
flag = a==0;                              flag = a==0;
 :                                          :
 :  /* no assignments to flag or a */   ⇒   :  /* no assignments to flag or a */
if(flag) ...                              if(a==0) ...
```

Where there is a single assignment, which cannot be substituted because of the presence of intervening assignments to other variables needed by the definition of the flag variable, temporary variables are used to reduce the problem to that previously considered. For example:-

```
                          Ta = a;                  Ta = a;
flag = a==0;              flag = a==0;             flag = a==0;
 :                         :                        :
a=a+1;            ⇒       a=a+1;           ⇒       a=a+1;
 :                         :                        :
if(flag) ...              if(flag) ...             if(Ta==0) ...
```

Where there are multiple assignments, these are gathered together into a single assignment, which can then be handled by the approach above. For example:-

```
x = y+1;               x = y+1;                            x = y+1;
y = x*2;               y = x*2;                            y = x*2;
flag = x>y;       ⇒    flag = x>y;                    ⇒    flag=x>y || (y+x>y) ==0;
y = y + flag;          flag=flag || (y + flag) ==0;        x = y*x;
flag=flag || y ==0;    x = y*x;
x = y*x;
```

Where the multiple assignments to the flag variable occur on different branches in an acyclic control flow graph which defines the flag variable, these are transformed into conditional assignments, which can then be treated using the approach above.

Figure 2: Sketch of the Flag Removal Algorithm

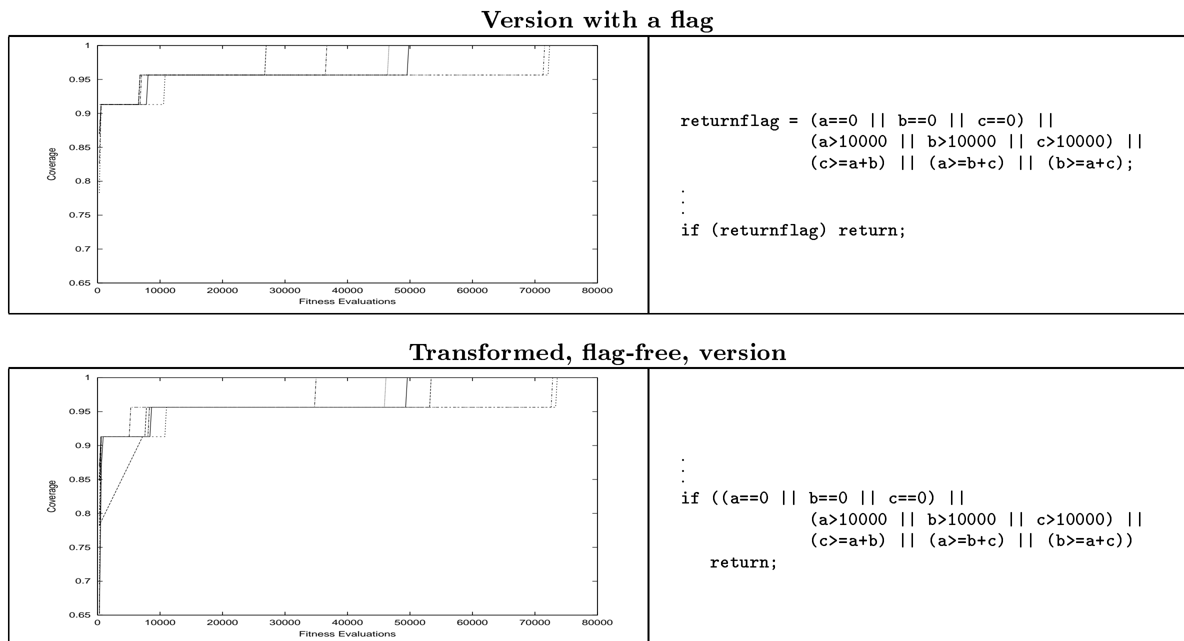
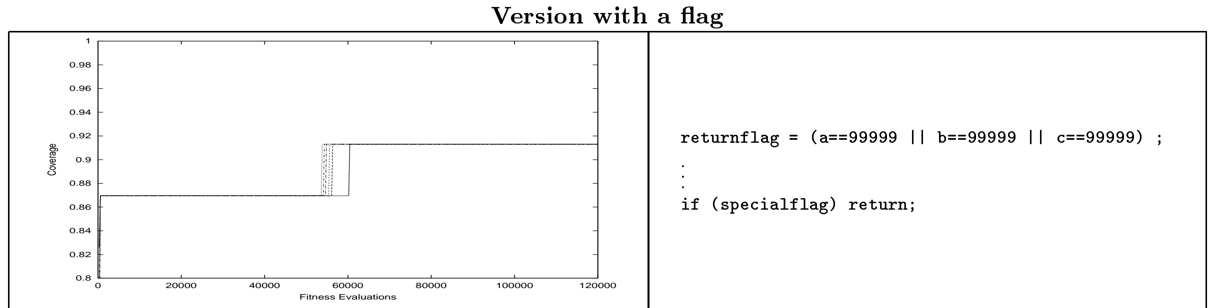


Figure 3: Results for the ‘Validity Check’ Triangle program

Version with a flag



Transformed, flag-free, version

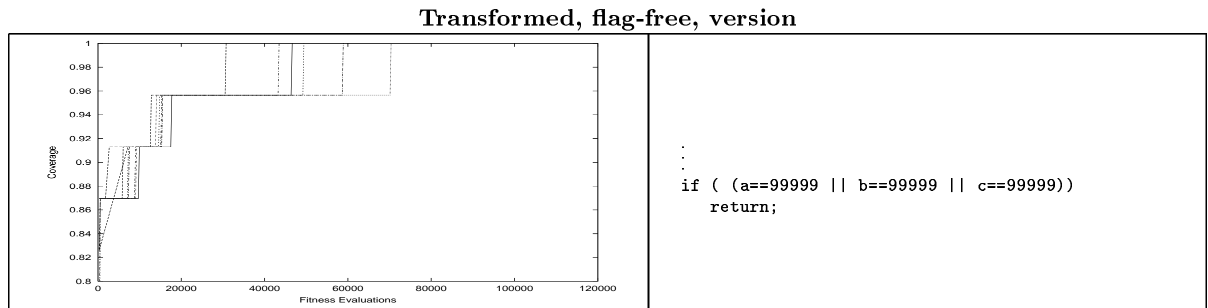
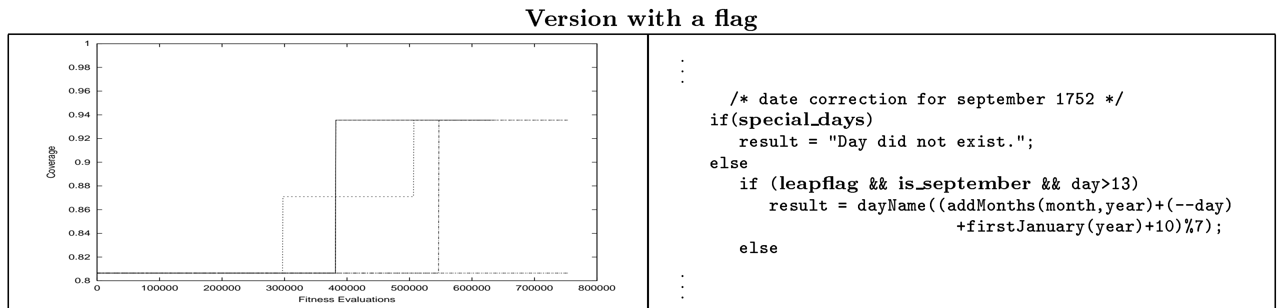


Figure 4: Results for the 'Special Value Check' Triangle program

Version with a flag



Transformed, flag-free, version

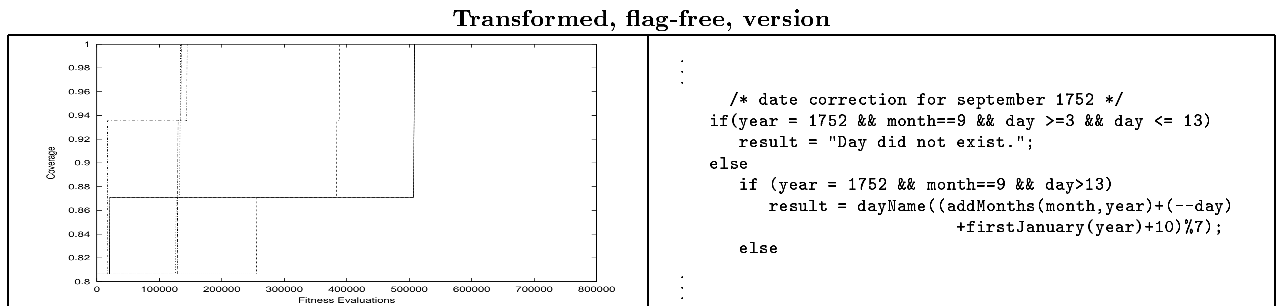


Figure 5: Results for the Leap Year Program