Symbolic Search-Based Testing

Arthur Baars
Universidad Politécnica de Valencia
Valencia, Spain
abaars@pros.upv.es

Mark Harman
University College London
CREST Centre, London, U.K.
mark.harman@ucl.ac.uk

Youssef Hassoun
King's College London
London, U.K.
youssef.hassoun@kcl.ac.uk

Kiran Lakhotia
University College London
CREST Centre, London, U.K.
k.lakhotia@ucl.ac.uk

Phil McMinn
University of Sheffield
Sheffield, U.K.
p.mcminn@sheffield.ac.uk

Paolo Tonella
Fondazione Bruno Kessler
Trento, Italy
tonella@fbk.eu

Tanja Vos
Universidad Politécnica de Valencia
Valencia, Spain
tvos@dsic.upv.es

***Abstract*—We present an algorithm for constructing fitness functions that improve the efficiency of search-based testing when trying to generate branch adequate test data. The algorithm combines symbolic information with dynamic analysis and has two key advantages: it does not require any change in the underlying test data generation technique and it avoids many problems traditionally associated with symbolic execution, in particular the presence of loops. We have evaluated the algorithm on industrial closed source and open source systems using both local and global search-based testing techniques, demonstrating that both are statistically significantly more efficient using our approach. The test for significance was done using a one-sided, paired Wilcoxon signed rank test. On average, the local search requires 23.41% and the global search 7.78% fewer fitness evaluations when using a symbolic execution based fitness function generated by the algorithm.**

***Keywords*-Search–Based Testing, Symbolic Execution, Fitness Functions**

I. Introduction

Automation is essential in software testing because the process is very slow and consequently expensive if undertaken manually. This need to automate software testing has provided a rich set of challenging problems for the research community for over thirty years. One approach to software test automation that has achieved a great deal of recent attention is Search-Based Software Testing (SBST). SBST uses meta-heuristic algorithms to automate the generation of test inputs that meet a test adequacy criterion. One of the most widely-studied test adequacy criteria in SBST is branch coverage([1], [2], [3], [4], [5], [6], [7]), the adequacy criterion considered in this paper.

Despite the large body of work in SBST focusing on branch coverage, the state of the art fitness function definitions used for branch adequate testing have changed little since the early seminal work on the Daimler Automated Software Testing System, which has been in use for more than a decade [7]. Though there have been many developments in SBST, these focus on changing the search algorithms and the way in which they are used, rather than the underlying fitness functions on which all metaheuristic search relies.

This paper takes a different approach and proposes to use static analysis. In particular, we use a form of partial symbolic execution to statically collect information available at compile time that can be used to define richer and more expressive fitness functions. We do not perform a complete symbolic execution, as this would be computationally expensive. Rather, we compute smaller amounts of symbolic information that can be used to imbue a fitness function with a much finer characterisation of the true search landscape, which defines the location of the global optima that represent the coverage of individual branches.

Our aim in attacking the underlying fitness function is to provide an approach that makes SBST more efficient (and possibly more effective), regardless of the particular search algorithm used to generate the test data. We present results for two widely used approaches to demonstrate, empirically, that our approach does indeed make SBST more efficient. There are many search algorithms that we could have chosen to study in our experimental work. A recent survey of Search-Based Approaches in Software Engineering [8] listed 15 search-based algorithms that have been used in SBST work. Clearly, it is not possible to report results for all of them in this paper.

Rather than make an arbitrary choice of algorithms to study, we choose to empirically study a local search (the widely used Alternating Variable Method (AVM) of Korel [3]) and a global search (the Genetic Algorithm approach used by Wegener *et al.* [7] and widely followed by other subsequent SBST research). Our reason for this choice was that these two approaches characterise the two possible outcomes for the primary choice of which algorithm to use; whether it will be local or global. Most of the other algorithms used in SBST are formed by a combination of local and global search. Our results indicate that the partial symbolic information we compute can indeed improve the efficiency of SBST, for both real world production code and for open source. In the case of a local search, the information also leads to improved effectiveness of SBST.

The primary contributions of this paper are:

- We introduce an algorithm for improving SBST by enriching fitness functions with statically collected symbolic information. Because our approach targets the fitness function itself, it applies to any and every SBST technique and can be incorporated without change to the search-based algorithm that uses the enriched fitness function.

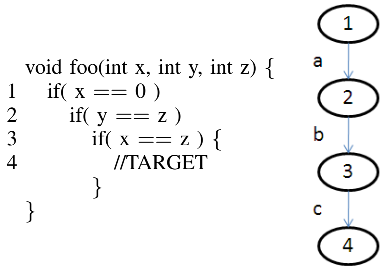
- We introduce an approach to overcome the problem of loops in traditional symbolic execution that allows us to approximate the impact of symbolic information on fitness. We introduce a new metric called the *approximation level*[1] to account for uncertainty whenever we cannot compute precise symbolic information, such as in the presence of loops.
- We present the results of an empirical study on both open and closed source code, the results of which indicate that our enriched fitness functions are significantly more efficient than their traditional counterparts.

The rest of the paper is organised as follows: the next section provides an overview of the standard fitness function used in SBST for branch coverage. Section III introduces our fitness function enhancement approach, while Section IV introduces the code analysis algorithm based on symbolic execution. Section V presents the empirical study, with corresponding threats to validity discussed in Section VI. Section VII describes related work and Section VIII draws conclusions.

II. Background

Meta-heuristic algorithms rely on a fitness function to guide the search towards a global optimum, *i.e.*, the desired test data. For branch coverage, the state of the art fitness function comprises two measures: a *branch distance* and an *approach level*. When both these measures are 0 the desired test data has been found. The approach level records how many of a target branch's control dependent nodes were not executed by a particular input. The fewer control dependent nodes executed, the 'further away' the input is from executing the target in control flow terms. Consider the example from Figure 1 and assume the target is the true branch of node (3). If an input takes the false branch at node (1) then the approach level is 2, and if an input takes the false branch at node (2), the approach level is 1 and so forth.

Whenever an input misses the target branch, the branch distance measure is used to compute how close the input was to staying on a path leading to the target. It is computed using the condition of the last (Control Flow Graph) node in an input's execution trace which holds a transitive control dependence on the target, and where execution diverged from the target. Resuming the example from Figure 1, if an input takes the false branch at node (1), the branch distance is computed through $|x - 0| + K$, where K is a failure constant ($K = 1$ throughout this paper). Different branch distance formulae exist depending on the relational predicate types used within the condition of branching nodes on which the target is control dependent. The interested reader is referred to the work of Tracey *et al.* [9] for a complete list of branch distance functions.



```
  void foo(int x, int y, int z) {
1    if( x == 0 )
2       if( y == z )
3          if( x == z ) {
4             //TARGET
          }
  }
```

Fig. 1. Example C code used to demonstrate the standard fitness function used in Search-Based Testing.

For a target branch t, an input vector v, and a node n where execution diverged from t, the complete fitness value is then computed by combining the branch distance and approach level:

$$\mathit{ff}_n(t, v) =$$
$$approach\_level(t, v) + norm(branch\_distance(t, v))$$

Note that the branch distance measure is normalized to a value between $[0, 1]$, using either of the following normalization functions [10]:

$$norm(d) = \begin{cases} 1 - 1.001^{-d} & \text{or,} \\ \frac{d}{(d+1)} & \text{(used in this paper)} \end{cases}$$

This fitness function can be inefficient when multiple, interdependent conditions need to be satisfied as in the example from Figure 1. For instance, when trying to cover the true branch at node (3), the values chosen for the inputs `x,y` and `z` that satisfy the first two conditions are unlikely to traverse the true branch at node (3). This is because the probability of the search optimizing both `y` and `z` to 0 is low.

In general, optimizing each condition in 'isolation', as is the case with the standard fitness function, can be considered sub-optimal. Symbolic execution on the other hand is able to capture such constraints and interdependencies between variables in the form of a *path condition*. For example, the path condition describing the execution where all conditions evaluate to true in Figure 1 would be $\langle x = 0 \wedge y = z \wedge x = z \rangle$, where x, y and z denote symbolic variables corresponding to the three integer inputs `x,y,z`. For the purpose of testing, a path condition can be fed to a constraint solver to obtain concrete input values which can be used to execute the program.

However, it is well known that static symbolic execution of a program faces several challenges, arising from loops and code constructs that cannot easily be symbolically executed such as unknown (library) functions, complex pointer arithmetic and functions pointers to name but a few. Loops in particular are a common problem, because they can result in infinitely many program paths and further, when trying to cover a target branch, it may not be possible to determine the number of loop iterations necessary to reach the target a priori.

[1] Note that the definition of approximation level in this paper is not to be confused with the approximation level defined in [7], which is equivalent to the *approach level* metric described in Section II.

The field of Dynamic Symbolic Execution (DSE), also known as concolic testing, first introduced by Godefroid *et al.* [11] tries to overcome some of the challenges faced by static symbolic execution. In DSE, information obtained through dynamic analysis is used to aid symbolic execution. The work presented in this paper proposes to do the opposite, *i.e.*, use information gathered through symbolic execution to aid SBST.

III. Symbolically Enhanced Fitness Function

The hypothesis underlying the research work presented in this paper assumes that incorporating information obtained from symbolic execution into the fitness function for branch coverage reduces the number of fitness evaluations required to cover a branch. We call our approach *fitness function enhancement* because the information collected along a program path using symbolic execution is used in place of the traditional approach level and branch distance measures.

Before the formal presentation of the fitness function enhancement algorithm we provide some initial intuition. We propose to replace the branch distance measure introduced in Section II with a *path distance* measure. Assume an input follows the false branch at node (1) in Figure 1 and that our target is the true branch of node (3). We start by computing a path expression [12] representing all paths from node (1) to the target. Let this path expression be *abc* (the edge labels a, b, c refer to the sub-graph shown on the right in Figure 1). We then symbolically execute this path expression to obtain a set of (partial) path conditions. In our example this set denotes a singleton of the form $\{\langle x = 0 \wedge y = z \wedge x = z \rangle\}$ because there is only one path from node (1) to the true branch of node (3). Next we apply the branch distance measure from Section II to each of the atomic conditions in the *path condition* (*i.e.*, $x = 0, y = z, x = z$), and sum the results to form a path distance. In case we have more than one path distance, we choose the minimum for our fitness computation. The intuition behind this choice is that the path condition with the smallest path distance is the closest to being satisfied by an input.

It may not always be possible to symbolically execute a path expression due to sources of uncertainty. To account for this we introduce a second measure called the *approximation level*. The approximation level will be defined as the number of conditions that cannot be added to a path condition, and are thus not considered in the path distance. For example, a condition that uses variables whose definition originates from a statement inside a loop will be dropped. This is because in general we do not know how often a loop is executed, thus we also do not know the final value of the variables that are defined in a loop. Other sources of uncertainty can include variables defined through system calls to which we do not have access.

The next section will provide formal definitions of the *approximation level* and *path distance*, along with the algorithm for computing the enhanced fitness function (*eff*).

```
  void foo(int n, int a, int b) {
1   int s = 0;
2   if (n > 0) {
3     for (int i = 0 ; i < n ; i++)
4       s += i;
5     if (a == n) {
6       if (s == 10) {
7         if (b == s) {
8           // TARGET (t)
          }
        }
      }
    }
  }
```

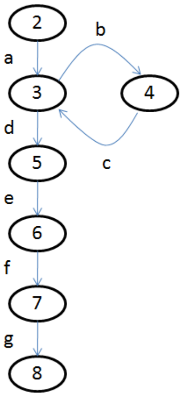


Fig. 2. Illustrative C code involving a loop with nested if-statements, used to demonstrate the symbolically enhanced fitness function.

A. Definitions

Let p be the path expression representing all paths between a start node n and a target node t. This path expression may contain loops, represented as terms of the form A^*. For such terms, we may opt for an arbitrary level of unrolling (*e.g.* k times), but we cannot handle unbounded (potentially infinite) unrollings. As a consequence, when the upper limit for the number of unrollings is reached, we make the assumption that variables defined in any successive loop iterations are destroyed, since in general, we cannot determine the required number of loop iterations. Such variables will be represented by the term $D[A]$. The path expression involving loops (*i.e.*, A^*) can thus be expanded as:

$$A^* = 1 + A + A^2 + A^3 + \ldots + A^k D[A]$$

By replacing A^* with $A^* = 1 + A + A^2 + A^3 + \ldots + A^k D[A]$ in the path expression p we obtain an *approximated path expression* p' which contains some *destroy* terms of the form $D[A]$. In the path condition produced by symbolic execution of p', we drop any clauses involving variables whose definition is inside a destroyed sub-path. Such destroyed clauses are counted and their number defines the approximation level used in the fitness function. Path conditions built by dropping one or more conditions are said to be *partial*; the others are said to be *complete*.

Definition 1. Branch distance: *A quantification in the range* $[0, 1]$ *of a boolean branch condition, such that the value zero is obtained iff the condition evaluates to true. Values close to* 1 *indicate that the condition is far from being satisfied. Intermediate values should be such as to smoothly guide the search toward satisfying the condition.*

Definition 2. Path distance: *A quantification of the (partial or complete) path condition, given by the sum of the branch distances computed for the conditions appearing as non-destroyed in the path condition for the approximated path expression. It is zero when all conjuncts in the path condition evaluate to true.*

Whenever we build a partial path condition, we are dropping a number of conditions which involve data dependencies originating in a loop. The number of dropped conditions is the approximation level.

Definition 3. Approximation level: *The approximation level along a path is the number of conditions that are dropped from the path condition since they involve variables defined inside loops that are used in the condition.*

An approximated path expression p' can always be normalized into a sum of alternative paths. In fact, in p', loops A^* are replaced by alternative k-bounded loop iterations, hence p' contains only sequence (multiplication) and alternative (sum) operators, which can be normalized into a sum of products by resorting to distribution of multiplication over sum.

Definition 4. Fitness function: *Let the normalized approximated path expression p' have the form $p_1 + p_2 + \ldots + p_h$, the fitness function eff for a node n is defined as:*

$$\mathit{eff}_n = \min\{\mathit{ff}_1, \mathit{ff}_2, \ldots, \mathit{ff}_h\}$$

where $\mathit{ff}_1, \mathit{ff}_2, \ldots, \mathit{ff}_h$ are the fitness functions for the atomic paths in p', each being computed as the sum of approximation level and path distance:

$$\mathit{ff}_i = \mathit{approximation\_level} + \mathit{path\_distance}(p_i)$$

Consider the example code in Figure 2 and assume our target is the true branch at node (7). If an input traverses the false branch at node (2), the path expression representing all paths from node (2) to the target is $a(bc)^*defg$. We may distinguish paths not entering the loop, $(bc)^*$, from paths which enter it one or more times (*i.e.*, $k = 1$). The first case (not entering the loop) is described by the path expression $adefg$. Symbolically executing this path expression yields the following path condition: $\langle n > 0 \wedge 0 \geq n \wedge a = n \wedge s = 10 \wedge b = s\rangle$. The path described by this path condition is clearly infeasible, because the conditions $n > 0$ and $0 \geq n$ are mutually exclusive. Hence, we will not consider it any further.

The second case (entering the loop one or more times) can be described by the path expression $abcD[bc]defg$. The term $D[bc]$ indicates that all variable definitions occurring along the path bc are to be treated as *unknown*, because the number of iterations for the loop bc is unknown (it will be one or more). For the example in Figure 2, two variables are defined inside the loop bc; `s` and `i`. Since we do not know how often the loop will be executed, we also do not know the final values of `s` and `i` when we exit the loop. Therefore we *drop* any conditions obtained by symbolically executing the sub-path following the term $D[bc]$ that involve such variables, *i.e.*, we do not add those conditions to the path condition.

The approximation level accounts for this by being incremented for each condition that is dropped. Completing our example, symbolically executing the path expression $abcD[bc]defg$ yields the path condition: $\langle n > 0 \wedge 0 < n \wedge a = n\rangle$. Since we dropped three conditions ($1 \geq n, s = 10, b = s$) the approximation level is 3. Thus, the approximation level allows us to distinguish an input reaching node (2) and taking the false branch, from an input taking the true branch at node (2) and thus getting closer to the target. Note that the approximation level reaches 0 once we reach node (5).

IV. Algorithm to Compute Symbolically Enhanced Fitness Functions

Algorithm 1 Compute symbolically enhanced fitness functions

```
Input $CFG$: control flow graph of the program under test; $t$: target edge to be covered
Output $\mathit{eff}_n$: fitness function to be used by each test case reaching node $n$, for each CFG node $n$ holding a transitive control dependency on $t$
for each $CFG$ node $n \in N | n$ holds a transitive control dependency on $t$ do
    Compute the sub-graph $subCFG_n$ of $CFG$ from $n$ to $t$, i.e., the intersection between nodes/edges forward reachable from $n$ and nodes/edges backward reachable from $t$
    Apply the node reduction algorithm [12] to determine the path expression $p$ for $subCFG_n$
    Compute the approximated path expression $p'$ from $p$ by approximating loops $A^*$ in the path expression $p$ as $A^* \approx 1 + A + A^2 + A^3 + \ldots + A^k D[A]$, for some fixed value of $k$
    Normalize the approximated path expression $p'$ into a sum of products: $p' = p_1 + p_2 + \ldots + p_h$
    for each path $p_i$ in the normalized path expression $p'$ do
        Perform a symbolic execution along $p_i$, keeping track of destroyed variables and annotating destroyed conditions as $D[c]$; the result is path condition $pc_i$
        Turn the path condition $pc_i$ into a fitness function $\mathit{ff}_i$ by replacing conditions with branch distances and destroyed conditions with 1
    end for
    Define the fitness function $\mathit{eff}_n$ for node $n$ as: $\mathit{eff}_n = min\{\mathit{ff}_1, \mathit{ff}_2, \ldots, \mathit{ff}_h\}$
end for
```

Algorithm 1 shows the pseudo-code for the computation of the enhanced fitness functions introduced in the previous section. Input to the algorithm is a program, represented as its Control Flow Graph (CFG), and a CFG edge t that represents the current test target, *i.e.*, the branch to be covered. The output produced by the algorithm is a set of symbolically enhanced fitness functions, one for each CFG node n that holds a transitive control dependency on t. For each such node that is part of the execution trace of an input, the corresponding fitness function is evaluated, with the minimum value forming the overall fitness for that input.

For all nodes n that hold a transitive control dependency on the target branch, Algorithm 1 determines the path expression

p representing all paths from n to the target t (steps 2-3). Then, loops are approximated (typically as $A^* \approx 1 + AD[A]$) and an approximated path expression p' is computed and normalized into a sum of products (steps 4-5). For each normalized approximated path expression p_i composing the path p', symbolic execution is used to compute the corresponding path condition pc_i (step 7). Whenever a destroyed sub-path is encountered during the symbolic execution, all variables defined inside the sub-path are collected among the destroyed variables. Successively added conditions which make use of destroyed variables are marked as destroyed conditions. In step 8, the path condition pc_i is converted into a fitness function for p_i by replacing conditions with branch distances, except for destroyed conditions, which increase the approximation level by one. The final fitness function for node n is the minimum among the fitness function values computed along the alternative paths appearing in the normalized approximated path expression.

V. Empirical Study

The aim of the empirical study in this paper is to analyse the impact of using the enhanced fitness function in SBST. The two research questions to be addressed by the study are as follows:

Research Question 1 - Effect of eff on branch coverage. The level of branch coverage achieved, *i.e.*, effectiveness of the testing technique, is often the main focus when investigating an automated test data generation approach. Our proposed change in fitness function should not negatively affect the level of coverage achieved by a test data generation technique. Does this hypothesis hold?

Research Question 2 - Effect of eff on efficiency. Alongside coverage, efficiency is also an important factor of any testing technique. Does the enhanced fitness function make SBST more efficient, and if so, what is the performance increase?

We selected two commonly used search algorithms for evaluation; a form of hill climbing known as the Alternating Variable Method (AVM), first introduced by Korel [3], and a Genetic Algorithm (GA) based on the approach described by Wegener *et al.* [7]. Details of the two algorithms can be found in Section V-A and Section V-B respectively. The search-based testing framework, IGUANA [13], was extended to include the enhanced fitness function proposed in this paper and subsequently used to perform the test data searches.

The study was performed on 338 branches, drawn from five different C programs, two of which were provided by Daimler, two are open source and one is the well-studied triangle program. The input domain for each function is composed of global variables and formal parameters. We chose not to use any input domain reduction and defined the domain of each variable according to its declared type. Details of the subjects used in the empirical study can be found in Table I.

The programs `f2` and `defroster` are industrial case studies provided by Daimler and represent production code for engine and rear window defroster control systems. The code is machine generated from a design model of the desired behaviour. To complement the industrial examples, two open-source case studies were selected. `tiff-3.8.2` is a library for manipulating images in the Tag Image File Format (TIFF). The functions tested comprise routines for placing images on pages and for building 'overview' compressed sample images. Finally, triangle is the well-known triangle classification program, often used as a benchmark program in automated test data generation studies.

Each search for test data was performed 30 times for every combination of fitness function and search algorithm. If test data was not found to cover a branch after $100,000$ fitness evaluations, the search was terminated. Serendipitous coverage, *i.e.*, branches covered by accident during the test data generation process, was ignored, so that a distinct search was carried out for every branch. The success or failure of each search was recorded, along with the number of fitness evaluations required to find the test data. From this, the 'success rate' of each branch can be calculated – the percentage of the 30 runs in which test data to execute the branch was found. The 30 runs were performed using an identical list of fixed seeds for random number generation, so as to provide a basis for assessment with tests for statistical significance using a one-sided, paired Wilcoxon signed rank test. Such tests are necessary to provide robust results in the presence of the inherently stochastic behaviour of the search algorithms.

To facilitate replication, we will now discuss the configuration of the two search algorithms used in the study.

A. Alternating Variable Method Setup

The AVM is a simple but effective optimization technique [2]. It is a form of hill climbing and works by continuously changing an input parameter to a function in isolation.

Initially all (arithmetic type) inputs are initialized with random values. Then, so called *exploratory moves* are made for each input in turn. These consist of adding or subtracting a delta from the value of an input. For integral types the delta starts off at 1, *i.e.*, the smallest increment (decrement).

When a change leads to an improved fitness value, the search tries to accelerate towards an optimum by increasing the size of the neighbourhood move with every step. These are known as *pattern moves*. The formula used to calculate the delta added or subtracted from an input is: $\delta = 2^{it} \cdot dir \cdot 10^{-prec_i}$, where it is the repeat iteration of the current move (for pattern moves), dir either -1 or 1, and $prec_i$ the precision of the i^{th} input variable. The precision applies to floating point variables only (*i.e.*, it is 0 for integral types). It denotes a scale factor for the size of a neighbourhood move. For example, setting the precision ($prec_i$) of an input to 1 limits the smallest possible move to ± 0.1. Increasing the precision to 2 limits the smallest possible move to ± 0.01, and so forth. For all experiments carried out in this paper, the precision for floating point variables was fixed at 3.

Once no further improvements can be found for an input, the search continues optimizing the next input parameter, and

TABLE I

Details of the test subjects.The Lines of Code column contains the *ansic* output of the sloccount tool [14] used in its default setting and applied to the root source directory of each program.

Test Subject/ Function	Lines of Code	Number of Branches Attempted	Number of Loops	Approximate Domain Size
bibclean	10,252			
check_ISBN		54	1	2^{112}
check_ISSN		54	1	2^{112}
defroster	179			
Defroster_main		72	0	2^{137}
f2	305			
F2		46	0	2^{272}
tiff-3.8.2	47,794			
TIFF_GetSourceSamples		32	2	2^{135}
TIFF_SetSample		28	0	2^{1102}
PlaceImage		24	0	2^{8402}
triangle	53			
triangle		28	0	2^{96}
Total	**58,583**	**338**	**4**	

may recommence with the first input if necessary. In case the search stagnates, *i.e.*, no move leads to an improvement, the search restarts at another randomly chosen location in the search-space. This is known as a *random restart* strategy and is designed to overcome local optima and enable the AVM to explore a wider region of the input domain for the function under test.

B. Genetic Algorithm Setup

A GA is a global search algorithm first proposed by Holland in the 1970s [15]. The configuration of the GA used in this paper is based on the approach described by Wegener *et al.* [7] who used GEATbx by Hartmut Pohlheim [16].

An overall population of 300 individuals is divided into six competing sub-populations, which begin with 50 individuals each. After evaluation, individuals in each sub-population are sorted using a linear ranking method [17] with a selection pressure of 1.7. Then, individuals are selected for reproduction through Stochastic Universal Sampling (SUS) [18]. In SUS, the probability of an individual being selected is proportionate to its (rank-based) fitness value. Selected individuals are recombined using a discrete recombination strategy [19], whereby an offspring receives each gene from either parent with an equal probability.

After recombination, offspring individuals are mutated according to the breeder genetic algorithm mutation strategy [19]. The mutation operator is applied with probability $\frac{1}{len}$, where len is the number of genes in an individual (*i.e.*, the length of the input vector). For each gene to be mutated, a mutation range $r_i = size \cdot dom_i$ is defined, where dom_i is the domain size of the i^{th} input parameter and $size$ is a mutation step size. The mutation step size varies for each of the six sub-populations and is defined as $size = 10^{-pop}$ with $1 \leq pop \leq 6$. The mutated value of an input parameter can thus be computed as $v_i = x_i \pm r_i \cdot \eta$. Addition or subtraction is chosen with an equal probability, and $\eta = \sum_{x=0}^{15} \alpha_x \cdot 2^{-x}$, where α_x is 1 with a probability of $\frac{1}{16}$ and 0 otherwise. After mutation, offspring are reinserted into a population using an elitist reinsertion strategy. That is, the top 10% of the current generation is retained and the remaining individuals are replaced by fitter offspring.

A feature of the Wegener model is that the six sub-populations of the GA compete with one another for the number of individuals each sub-population evolves. An average fitness value is computed for each sub-population and this value is used to linearly rank the sub-populations (again using a selection pressure of 1.7). The rank-based fitness value $rank$ of a sub-population is then used to compute a progress value $prog$ for the population in generation g using the formula $prog_{g+1} = 0.9 \cdot prog_g + 0.1 \cdot rank$. Then, after every four generations, the populations are ranked according to their progress value $prog$, and the size of each sub-population is updated, with weaker sub-populations transferring individuals to stronger ones. However, no sub-population can lose its last five individuals, preventing it from dying out. Finally, a general migration of individuals takes place after every 20^{th} generation, where sub-populations randomly exchange 10% of their individuals with one another.

C. Results

Research Question 1 - Effect of *eff* on branch coverage. Figure 3 shows the coverage achieved by the AVM and the GA for each test subject. A branch is counted as covered if the search for test data succeeded in at least one out of the thirty runs. As can be seen, using a symbolically enhanced fitness function does not negatively affect the level of branch coverage achieved by either local or global search. Instead, the GA is able to cover a branch that it previously failed to cover. Similarly, the local search is able to cover more branches when using the enhanced fitness functions.

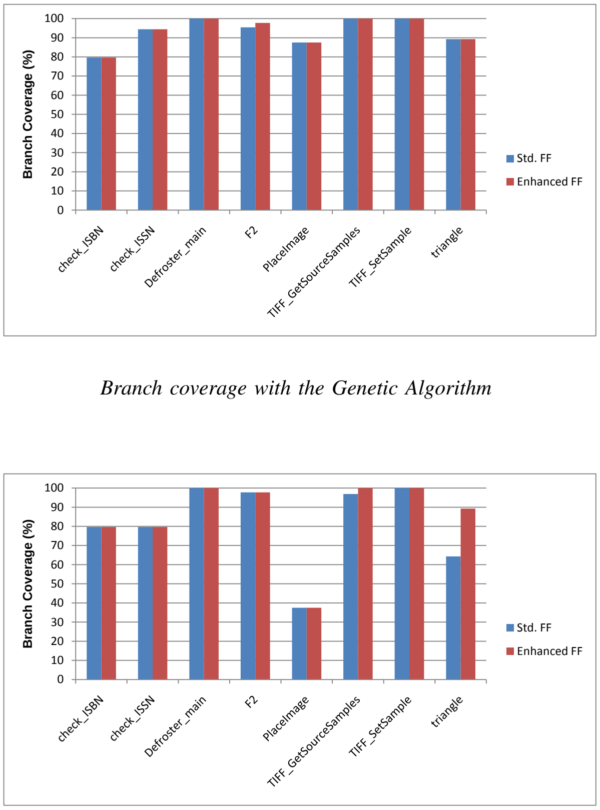
To gain a better understanding of how the proposed approach affects each search algorithm, we also computed the success rate for each search target. Table II lists the branches for which we observed a difference in success rate when using the enhanced fitness function. The GA exhibits little variation. For three branches, the success rate is slightly reduced when using the enhanced fitness function. However, for five branches the success rate increases.

Compared to the GA, the enhanced fitness function has a bigger impact on the success rate of the AVM. For branches where we observed a difference, the trend is in an increase in success rate. Five branches stand out particularly because the AVM failed to find test data for these using the standard fitness function. With the enhanced fitness function, the search was able to find the required test data in all of the 30 runs. The effect of the enhanced fitness function is not always beneficial though; for example branches in the function `F2` from Daimler are covered with a reduced success rate. This function is interesting because some `if` statements check if a subtraction operation (on operands of type `short int`) resulted in an over- or underflow. For example, for one branch where the enhanced fitness function performed worse than the standard fitness function, the path distance measure is computed using the path condition $\langle V11 \geq 0 \wedge V9 < 0 \wedge (V11 - V9) < 0 \rangle$. The conjuncts of the path condition correspond to three nested `if` statements in the original code. When the path distance is computed, the first two conjuncts *pull* into the opposite direction of the last conjunct. That is, as the branch distance for the first two conjuncts converges towards 0, the branch distance for the third conjunct increases until an overflow occurs. The standard fitness function, which optimizes each of the conjuncts in turn, does not appear to suffer from this problem and is able to reliably find the required test data. All cases where the enhanced fitness function did worse than the standard fitness function for `F2` were in code that checks for over- or underflow errors.

Research Question 2 - Effect of eff **on efficiency**. The results support the hypothesis that enhancing the fitness function with information gathered from symbolic execution can reduce the number of fitness evaluations required to cover a branch. Details of the average number of fitness evaluations required by each search technique are given in Table III.

The trend for the GA is to require fewer fitness evaluations with the enhanced fitness function. This difference is particularly visible for three functions, where we observed more than a 25% reduction in fitness computations. However, there is again one case (`PlaceImage` from `tiff-3.8.2`) where we see a small increase in the number of fitness evaluations.

As with the success rates, the AVM benefits more from the enhanced fitness function than the GA. Four functions require fewer than 50% of the fitness evaluations compared to the standard fitness function. This is not surprising since all these functions contain branches for which the AVM failed to find test data using the standard fitness function, but for which it achieved a 100% success rate using the enhanced fitness function. Conversely, the AVM uses more fitness evaluations with the enhanced fitness function for `F2`, because branches are covered with a lower success rate, and each failed search results in 100,000 fitness evaluations.



Branch coverage with the Genetic Algorithm

Branch coverage with the Alternating Variable Method

Fig. 3. This Figure shows the branch coverage achieved by the Genetic Algorithm (top) and the Alternating Variable Method (bottom) when using the standard and enhanced fitness functions. The graphs confirm that symbolically enhanced fitness functions are equally or more effective than the standard fitness functions.

To see if the differences in efficiency for the GA and the AVM are statistically significant, we used the statistical tool R [20] to perform a paired, one-sided Wilcoxon signed rank test with continuity correction and specified an alpha level of 0.01. For both the GA and AVM we obtained a p value of 2.2×10^{-16}. This p value indicates that the difference in the number of fitness evaluations required by each search algorithm is statistically significant ($p \leq \alpha$).

Finally, we also recorded the time taken to perform the upfront static analysis required by the enhanced fitness function. To obtain a reasonable sample pool we repeated this analysis 30 times for each function. The average analysis times, alongside standard deviation are recorded in Table IV.

Loops often result in path explosion, even when only a single loop unrolling is performed. Thus, the analysis takes longer for functions containing one or more loops. Note that the symbolic analysis is performed once per function and can be re-used by a search algorithm for all branches contained within that function. Therefore, compared to the overall execution time of the test data generation algorithms, we consider the analysis times reported in this paper as acceptable. Future work might investigate how we can make the static analysis

TABLE II

Difference in success rates with the standard fitness function and the enhanced fitness function. Branches are only listed if there is a difference for either the AVM or GA.

Test Subject/ Function (Branch ID)	AVM Standard/ Enhanced		GA Standard/ Enhanced	
bibclean				
check_ISSN (53T)	0% / 0%	(0%)	43.33% / 60.00%	(**+16.67%**)
check_ISSN (55T)	0% / 0%	(0%)	16.67% / 66.67%	(**+50.00%**)
check_ISSN (58T)	0% / 0%	(0%)	100% / 96.67%	(**-3.33%**)
check_ISSN (58F)	0% / 0%	(0%)	100% / 76.67%	(**-23.33%**)
f2				
F2 (4T)	100% / 53.33%	(**-46.67%**)	100% / 100%	(0%)
F2 (15T)	100% / 46.67%	(**-53.33%**)	100% / 100%	(0%)
F2 (35T)	100% / 100%	(0%)	100% / 96.67%	(**-3.33%**)
F2 (43T)	3.33% / 3.33%	(0%)	0% / 6.67%	(**+6.67%**)
tiff-3.8.2				
TIFF_GetSourceSamples (14T)	6.67% / 100%	(**+93.33%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (17T)	0% / 100%	(**+100%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (20T)	10.00% / 100%	(**+90.00%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (23T)	10.00% / 100%	(**+90.00%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (26T)	13.33% / 100%	(**+86.67%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (29T)	16.67% / 100%	(**+83.33%**)	100% / 100%	(0%)
TIFF_GetSourceSamples (32T)	23.33% / 100%	(**+76.67%**)	100% / 100%	(0%)
TIFF_SetSample (2T)	13.33% / 100%	(**+86.67%**)	100% / 100%	(0%)
TIFF_SetSample (5T)	13.33% / 100%	(**+86.67%**)	100% / 100%	(0%)
TIFF_SetSample (8T)	13.33% / 100%	(**+86.67%**)	100% / 100%	(0%)
TIFF_SetSample (11T)	26.67% / 100%	(**+73.33%**)	100% / 100%	(0%)
TIFF_SetSample (14T)	13.33% / 100%	(**+86.67%**)	100% / 100%	(0%)
TIFF_SetSample (17T)	20.00% / 100%	(**+80.00%**)	100% / 100%	(0%)
TIFF_SetSample (20T)	23.33% / 100%	(**+76.67%**)	100% / 100%	(0%)
triangle				
triangle (14T)	0% / 100%	(**+100%**)	100% / 100%	(0%)
triangle (15T)	0% / 93.33%	(**+93.33%**)	100% / 100%	(0%)
triangle (15F)	0% / 100%	(**+100%**)	96.67% / 100%	(**+3.33%**)
triangle (17T)	0% / 100%	(**+100%**)	100% / 100%	(0%)
triangle (17F)	0% / 100%	(**+100%**)	96.67% / 100%	(**+3.33%**)
triangle (19T)	0% / 96.67%	(**+96.67%**)	100% / 100%	(0%)
triangle (21T)	0% / 96.67%	(**+96.67%**)	100% / 100%	(0%)

TABLE III

Normalized average fitness evaluations required by the GA and AVM using the standard and enhanced fitness functions.

Test Subject/ Function	AVM Standard/ Enhanced		GA Standard/ Enhanced	
bibclean				
check_ISBN	100% / 99.99%	(**-0.01%**)	100% / 99.99%	(**-0.01%**)
check_ISSN	100% / 99.98%	(**-0.02%**)	100% / 93.71%	(**-6.29%**)
defroster				
Defroster_main	100% / 99.57%	(**-0.43%**)	100% / 68.15%	(**-31.85%**)
f2				
F2	100% / 157.44%	(**+57.44%**)	100% / 91.61%	(**-8.39%**)
tiff-3.8.2				
TIFF_GetSourceSamples	100% / 12.41%	(**-87.59%**)	100% / 73.05%	(**-26.95%**)
TIFF_SetSample	100% / 8.78%	(**-91.22%**)	100% / 73.87%	(**-26.13%**)
PlaceImage	100% / 99.17%	(**-0.83%**)	100% / 100.53%	(**+0.53%**)
triangle				
triangle	100% / 35.35%	(**-64.65%**)	100% / 98.79%	(**-1.21%**)

more efficient, for example by re-using symbolic information for nested branches.

VI. Threats to validity

Naturally there are threats to validity in any empirical study such as this. The first issue to address is the threat to the *internal validity* of the experiments, *i.e.*, whether there has been a bias in the experimental design that could affect the obtained results.

One potential source of bias comes from the configuration of the algorithms used in the test data generation tool IGUANA. The settings for the GA and AVM were taken from previous studies [1], [21] that looked at generating branch adequate test data. Thus, they have been shown in the past to provide a good trade-off between effectiveness and efficiency.

Another potential source of bias comes from the inherent stochastic behaviour of the meta-heuristic search algorithms. The most reliable (and widely used) technique for overcom-

TABLE IV
AVERAGE TIME (IN MILLISECONDS) TAKEN OVER 30 TRIALS TO PERFORM THE UP-FRONT STATIC ANALYSIS REQUIRED BY THE ENHANCED FITNESS FUNCTION. THE STANDARD DEVIATION IS SHOWN IN THE RIGHT MOST COLUMN.

Test Subject/ Function	Analysis Time(ms)	(StdDev)
bibclean		
check_ISBN	1,909,741.13	(2,172.47)
check_ISSN	1,792,913.50	(2,827.38)
defroster		
Defroster_main	34,509.87	(345.55)
f2		
F2	318,279.23	(431.68)
tiff-3.8.2		
TIFF_GetSourceSamples	516,514.23	(878.52)
TIFF_SetSample	1,092.77	(12.22)
PlaceImage	956.43	(105.36)
triangle		
triangle	716.03	(6.24)

ing this source of variability is to perform statistical tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments were repeated 30 times, providing a reasonable pool of data from which to draw observations, and ensuring sample means were normally distributed. To show that the enhanced fitness function is more efficient than the standard fitness function used in SBST, a test for a statistical significant difference in the sample means was performed. We used a one-sided, paired Wilcoxon signed rank test with the confidence level set at 99%.

A further source of bias includes the selection of the functions used in the empirical study, which could potentially affect its *external validity*, *i.e.*, the extent to which it is possible to generalize from the results obtained. The study draws upon code from real world programs, both from industrial production code and from open source. While we sampled a variety of programming styles and sources, we only considered functions from five programs. Therefore caution is required before making any claims as to whether these results would be observed on other functions. Instead, the results reported herein should only be seen to provide some initial intuition and a larger study is required to validate or refute our findings.

VII. Related Work

The present paper is the first to develop an amended form of symbolic execution for SBST. Previous work on developing symbolic execution as a practical means of improving automated testing focussed on constraint based testing techniques, leading to the development of the very active field now known as 'Dynamic Symbolic Execution' (DSE). This field began with the seminal work by Godefroid *et al.* [11] on Directed Automated Random Testing (DART), which combined symbolic execution with random testing. Since then a number of authors have followed this approach, which is sometimes referred to as 'concolic testing' [22] as well as DSE [23], [11], [24].

DSE and SBST have developed as separate schools of thought in automated software testing, each with their own advantages and disadvantages. The introduction of Dynamic Symbolic Execution creates a significant step forward in the development of previous constraint based approaches to automated test data generation, on which DSE builds.

Our introduction of partial symbolic execution as a means of augmenting SBST seeks to provide a similar impetus to SBST research. Like DSE, we augment an existing test automation technique with a form of symbolic execution and like DSE, we need to amend traditional symbolic execution to ameliorate its problems. However, DSE performs a complete symbolic execution, sometimes using concrete values in place of symbolic values, whereas our approach does not use concrete values, but retains the symbolic nature of symbolic execution. Rather than performing a complete symbolic execution, we perform a localised or 'partial' symbolic computation and use approximation to overcome the problems of static symbolic execution.

Majumdar and Sen [25] were the first to use random testing as a means to increase code coverage when combined with DSE. A random search is used to explore the state space of a program, while DSE is used to provide a locally exhaustive exploration from the points reached by the random search.

The first authors to propose a combination of SBST and DSE were Inkumsah and Xie [26] with the EVACON framework. Their framework targets test data generation for object oriented code written in JAVA and uses two existing tools: eToc [27], an evolutionary test data generation tool, and jCUTE [28], a DSE tool. Method sequences putting the class containing the method under test into specific states, are constructed by eToc. Then, jCUTE is used to maximize code coverage of a given method sequence by generating values for the sequences' input parameters. The method sequences with optimized parameter values are then passed back to eToc for further optimization.

More recently, Lakhotia *et al.* [29] investigated a combination of SBST and DSE in order to improve DSE's ability to handle constraints over floating point variables. Their work integrated the AVM, also used in this paper, and Evolution Strategies into Pex [24], a DSE tool for .NET.

Lakhotia *et al.* [30] also proposed a combination of symbolic execution with search in order to improve SBST. Inspired by the work on CUTE [22], they use symbolic execution to extend and improve the AVM for pointer inputs.

The work presented in this paper differs from all previous work in that it is the first to consider symbolic execution in order to improve a fitness function used in SBST. A benefit of this approach lies in its generality; it may be used with any search algorithm. Furthermore, the enhanced fitness function does not require a constraint solver, despite making use of symbolic execution techniques. This is an advantage when testing code that contains floating point computations or calls to system libraries.

VIII. Conclusion

This paper has introduced and evaluated a symbolic search-based software testing approach for the branch coverage test adequacy criterion. We propose to replace the existing *branch distance* and *approach level* measures with two new measures: *path distance* and *approximation level*. The new metrics make use of information gathered from symbolic execution. An empirical study, performed on 338 branches, taken from a mix of open source and industrial programs, confirmed our hypothesis that a symbolically enhanced fitness function can make search algorithms more efficient. The proposed approach was evaluated with two commonly used algorithms in Search-Based Software Testing: the Alternating Variable Method and a Genetic Algorithm.

ACKNOWLEDGEMENT

Arthur Baars, Kiran Lakhotia, Paolo Tonella and Tanja Vos are funded through the European Union project FITTEST (ICT-2009.1.2 no 257574). Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London. Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference).

REFERENCES

[1] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, “The Impact of Input Domain Reduction on Search-Based Test Data Generation,” in *ESEC/SIGSOFT FSE*, 2007, pp. 93–101.

[2] M. Harman and P. McMinn, “A theoretical and empirical study of search-based testing: Local, global, and hybrid search,” *IEEE Trans. Software Eng*, vol. 36, no. 2, pp. 226–247, 2010.

[3] B. Korel, “Automated software test data generation,” *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[4] K. Lakhotia, P. McMinn, and M. Harman, “An empirical investigation into branch coverage for C programs using CUTE and AUSTIN,” *The Journal of Systems and Software*, vol. 83, no. 12, pp. 2379–2391, Dec. 2010.

[5] C. C. Michael, G. McGraw, and M. Schatz, “Generating software test data by evolution,” *IEEE Trans. Software Eng*, vol. 27, no. 12, pp. 1085–1110, 2001.

[6] R. P. Pargas, M. J. Harrold, and R. Peck, “Test-data generation using genetic algorithms,” *Softw. Test, Verif. Reliab*, vol. 9, no. 4, pp. 263–282, 1999.

[7] J. Wegener, A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” *Information & Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[8] M. Harman, A. Mansouri, and Y. Zhang, “Search based software engineering: A comprehensive analysis and review of trends techniques and applications,” Department of Computer Science, King’s College London, Tech. Rep. TR-09-03, April 2009.

[9] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid, “An automated framework for structural test-data generation,” in *ASE*, 1998, pp. 285–288.

[10] A. Arcuri, “It does matter how you normalise the branch distance in search based software testing,” in *ICST*, 2010, pp. 205–214.

[11] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[12] B. Beizer, *Software Testing Techniques, 2nd edition*. International Thomson Computer Press, 1990.

[13] P. McMinn, “IGUANA: Input generation using automated novel algorithms. A plug and play research tool.” University Of Sheffield, Tech. Rep., 2007.

[14] D. A. Wheeler, “More than a gigabuck: Estimating GNU/Linux’s size,” http://www.dwheeler.com/sloc/, Jun. 2001.

[15] J. H. Holland, “Genetic algorithms and the optimal allocation of trials,” *SIAM Journal of Computing*, vol. 2, no. 2, pp. 88–105, Jun. 1973.

[16] H. Pohlheim, “Evolutionary algorithms: Overview, methods and operators.” documentation for: Genetic evolutionary algorithm toolbox for use with matlab version: toolbox 1.92 documentation 1.92,” 1999.

[17] D. Whitley, “The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best,” Computer Science Dept., Colorado State University, Fort Collins, CO, Tech. Rep., 1989.

[18] J. E. Baker, “Reducing bias and inefficiency in the selection algorithm,” in *Genetic Algorithms and their Applications (ICGA’87)*, J. J. Grefenstette, Ed., 1987, pp. 14–21.

[19] H. Mühlenbein and D. Schlierkamp-Voosen, “Predictive models for the breeder genetic algorithm, I: Continuous parameter optimization,” *Evolutionary Computation*, vol. 1, no. 1, pp. 25–49, 1993.

[20] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2011, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[21] M. Harman, K. Lakhotia, and P. McMinn, “A multi-objective approach to search-based test data generation,” in *GECCO*, 2007, pp. 1098–1105.

[22] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[23] C. Cadar and D. R. Engler, “Execution generated test cases: How to make systems code crash itself,” in *Model Checking Software, 12th International SPIN Workshop*, vol. 3639, 2005, pp. 2–23.

[24] N. Tillmann and J. de Halleux, “Pex-white box test generation for.NET,” in *TAP*, 2008, pp. 134–153.

[25] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *ICSE*, 2007, pp. 416–426.

[26] K. Inkumsah and T. Xie, “Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs,” in *ASE*, November 2007, pp. 425–428.

[27] P. Tonella, “Evolutionary testing of classes,” in *ISSTA*, 2004, pp. 119–128.

[28] K. Sen and G. Agha, “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools,” in *CAV*, 2006, pp. 419–423.

[29] K. Lakhotia, N. Tillman, M. Harman, and J. de Halleux, “FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution,” in *ICTSS*, 2010, pp. 142–157.

[30] K. Lakhotia, M. Harman, and P. McMinn, “Handling dynamic data structures in search based testing.” in *GECCO*, 2008, pp. 1759–1766.