

Handling Constraints for Search Based Software Test Data Generation

Ramón Sagarna and Xin Yao

School of Computer Science, The University of Birmingham
Edgbaston, Birmingham B15 2TT, United Kingdom
R.Sagarna@cs.bham.ac.uk, X.Yao@cs.bham.ac.uk

Abstract

A major issue in software testing is the automatic generation of the inputs to be applied to the programme under test. To solve this problem, a number of approaches based on search methods have been developed in the last few years, offering promising results for adequacy criteria like, for instance, branch coverage. We devise branch coverage as the satisfaction of a number of constraints. This allows to formulate the test data generation as a constrained optimisation problem or as a constraint satisfaction problem. Then, we can see that many of the generators so far have followed the same particular approach. Furthermore, this constraint-handling point of view overcomes this limitation and opens the door to new designs and search strategies that, to the best of our knowledge, have not been considered yet. As a case study, we develop test data generators employing different penalty objective functions or multiobjective optimisation. The results of the conducted preliminary experiments suggest these generators can improve the performance of classical approaches.

1. Introduction

Among the problems related to software testing, the automatic generation of the inputs to be applied to the programme under test is especially relevant. Exhaustive testing is generally prohibitive due to the huge size of the input domain, so tests are designed with the purpose of addressing particular aspects of the software system [3]. This makes the generation of test inputs a non-trivial task, as they must conform to the test type and its requirements.

Many of the approaches for tackling this task aim at creating inputs that fulfil a structural adequacy criterion. Though several criteria comprising different levels of complexity can be found in the literature, *branch coverage* is accepted as a minimum mandatory criterion nowadays [3]. So, in this case, the aim is to generate a set of inputs exercising every programme branch.

The automation of structural test data generation is typically reached by means of *random*, *static* or *dynamic* strategies. A random strategy relies upon a probability distribution for sampling all the inputs. So, its performance heavily depends on this distribution, which is often uniform [5]. The main feature of static strategies is that programme execution is not required to create inputs, since they are obtained through a static analysis of the source code. These approaches, however, suffer from well-known problems which limit their performance [10]. In contrast to static, dynamic strategies do execute the programme, and the information available at run-time is exploited to guide the generation of the test inputs [8]. While these methods must incur the overhead of programme execution, many of the drawbacks of previous strategies are overcome [11].

In recent years, several approaches under the name of Search Based Software Test Data Generation (SBSTDG) have been developed, offering promising results [11]. SBSTDG tackles the test data generation as a search for the appropriate inputs by formulating an optimisation problem. This problem is then addressed using search methods. Although this field comprises any testing type and search algorithm, most of the research to date has focused on dynamic strategies using Evolutionary Computation techniques [10, 13, 16, 18, 21, 22, 23].

A number of works in the software testing literature have devised the test data generation as the achievement of a set of constraints. Thus, constraints based approaches have already been proposed for mutation testing [14] and non-SBSTDG static-dynamic strategies [15], to name a few. Surprisingly, concepts from constraint-handling have scarcely been adopted for dynamic SBSTDG [12, 20]. This might be an interesting topic, since it would allow for the application of a wider range of techniques than those considered so far.

In the present work, we explicitly formulate the dynamic test data generation for branch coverage as a constraint-handling problem [1, 4]. We can see then that many of the generators based on Evolutionary Computation so far have to some extent followed the same particular approach.

Moreover, this formulation opens the door to new designs and search strategies that, to the best of our knowledge, have not been used to solve this problem yet. As a case study, we develop test data generators employing a Genetic Algorithm with different penalty objective functions [4] or multiobjective optimisation [7]. A preliminary empirical analysis is conducted on their behaviour for different scenarios and encouraging results are obtained.

The remaining sections are arranged as follows. First, dynamic SBSTDG approaches so far are outlined. In the next section, the constraints handling point of view to SBSTDG is explained and some new approaches are proposed. We continue with the empirical evaluation of these approaches. Finally, we discuss conclusions and some ideas for future work.

2. Dynamic Search Based Software Test Data Generation

SBSTDG methods obtain test inputs employing search techniques during the process. In the case of dynamic approaches, a usual practice is to create an instrumented version of the programme, which will give feedback concerning the execution with an input. This run-time information is then used to guide the search technique.

Although different works have been developed in the field to date (see [11] and references therein), the idea underlying many of them is to solve a number of function optimisation problems, one for each structural entity to be covered. Thus, it is common to follow the general scheme in Figure 1. This scheme is an iterative two-step process where, first, an entity is selected (e.g., a branch) and marked as an objective. In the second step, the objective entity is assigned a function dependent on the programme input and its optimisation is sought.

Repeat until stopping criterion is met
 $E \leftarrow$ Select objective entity to exercise
 Obtain input optimising function for E

Figure 1. General scheme for dynamic SBSTDG approaches.

2.1. Selection Step

Despite the different options that have been implemented for the selection step in Figure 1 [10, 16, 23], the objective entity is often determined with the help of a graph that reflects the structural characteristics of the programme. In the case of branch coverage, a control flow graph [6] is typically employed. A control flow graph $G = (V, U)$ is defined by a

set V of vertices and a set $U \subseteq V \times V$ of arcs. Each vertex in V represents a *code basic block*, excepting two vertices labeled s and e , which refer to the programme entry and exit. A code basic block is a maximal sequence of code statements such that if one is executed, then all of them are. An arc $(v_1, v_2) \in U$, with v_1 and v_2 distinct from s and e , is such that the control of the programme can be transferred from block v_1 to v_2 without crossing any other block. Analogously, for every arc $(s, v_1) \in U$ or $(v_2, e) \in U$, it will be possible to transfer the flow of control from the entry to block v_1 and from block v_2 to the exit, respectively. Hence, in this kind of graph, a programme branch comes defined by every vertex v with $outdegree(v) > 1$. Given a programme input x , we will call *execution path of x* to the path starting from s that represents the flow of the programme's control when executed with x .

2.2. Optimisation Step

The next step in Figure 1 tackles an optimisation problem. That is, given the search space Ω formed by the programme inputs and a function $f : \Omega \rightarrow \mathbb{R}$, find $x^* \in \Omega$ such that $f(x^*) \leq f(x) \forall x \in \Omega$. Next, we discuss two popular objective functions that have been defined for branch coverage.

Classical Objective Function

A measure that is widely used to create the objective function is the so-called *branch distance* [22, 23]. Let b be the objective branch and $\mathcal{A} \text{ OP } \mathcal{B}$ an expression of the conditional statement **COND** associated with b in the code, with **OP** denoting a comparison operator. Only for notation purposes, we also consider the vertex v_c representing **COND** in the control flow graph of the programme. The branch distance value for an input x that reaches **COND** is determined by

$$f^c(x) = d(\mathcal{A}_x, \mathcal{B}_x) + K \quad (1)$$

where \mathcal{A}_x and \mathcal{B}_x are appropriate representations of the values taken by \mathcal{A} and \mathcal{B} in the execution, d is a distance measurement, and $K > 0$ is a previously defined constant. Typically, if \mathcal{A} and \mathcal{B} are numerical, then \mathcal{A}_x and \mathcal{B}_x are their values and $d(\mathcal{A}_x, \mathcal{B}_x) = |\mathcal{A}_x - \mathcal{B}_x|$. In the case of more complex data types, a binary representation of the values for \mathcal{A} and \mathcal{B} can be obtained and, for instance, let d be the Hamming distance [21]. Besides, if **COND** involves a compound expression, the overall branch distance is constructed from the distances for each subexpression. Given two subexpressions C_1 and C_2 with their respective branch distances f_1^c and f_2^c , and an input x , the value for the logical expression $C_1 \vee C_2$ is $\min\{f_1^c(x), f_2^c(x)\}$, the logical expression $C_1 \wedge C_2$ is calculated as $f_1^c(x) + f_2^c(x)$, and for $\neg C_1$ the value is known by propagating the negation inside

C_1 . By applying the associative and commutative properties to different logical expressions, the overall value for f^c can be obtained.

Hence, a classical objective function based on the branch distance is defined, keeping the notation above, as follows [10, 18, 22]:

$$f(\mathbf{x}) = \begin{cases} M & \text{if COND not reached} \\ f^c(\mathbf{x}) & \text{if COND reached and } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where M is the largest computable value. It is easy to note that this function yields a plateau for inputs not reaching the condition of the branch.

Advanced Objective Function

An objective function with more gradient than the one in Equation 2 was presented in [23] and further studied in [2]. Besides the branch distance, this function employs an *approach level* to the condition of the branch, which is in turn based on the notion of *critical condition*.

Given a control flow graph G , we call a vertex v_1 a critical condition of vertex v_2 iff $outdegree(v_1) > 1$, a path p from v_1 to v_2 exists, and a path from v_1 to e not containing any vertex in p exists. Intuitively, a critical condition v_1 has an arc from which it is impossible to attain v_2 , so we must follow one of the other arcs. Now, given a branch b and an input \mathbf{x} , let v_c be the vertex representing the condition of b in the control flow graph, and let p be the execution path of \mathbf{x} . We can define a distance between a vertex $v \in p$ and v_c , $D(v, v_c)$, based on the number of critical conditions of v_c in a path from v to v_c . If several paths from v to v_c with different numbers of critical conditions exist, then we may take $D(v, v_c)$ as the minimum or maximum number of critical conditions, depending on whether we choose an optimistic or pessimistic approach [2]. For the sake of convenience, if no path exists from v to v_c then $D(v, v_c) = \infty$. So, based on these concepts, we describe the approach level as $l^c(\mathbf{x}) = \min_{v \in p} D(v, v_c)$. In other words, the approach level alludes to the number of critical conditions not achieved between v_c and its closest vertex in the execution path.

Thus, maintaining previous notation, the function in Equation 2 is extended as follows:

$$f(\mathbf{x}) = \begin{cases} l^c(\mathbf{x}) + \frac{f^a(\mathbf{x})}{M} & \text{if COND not reached} \\ \frac{f^c(\mathbf{x})}{M} & \text{if COND reached and } b \\ & \text{not attained} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where $l^c(\mathbf{x}) = D(v_a, v_c)$ and M is a normalisation term. The main idea of this function is to evaluate an input not reaching **COND** by combining two concepts: the approach level to v_c and how far the input was from taking

the *correct* arc at the vertex straying away from the path to v_c .

Even though the granularity of this function is greater or equal than for the classical function, it still presents problems with some code structures that have no clear solution yet [2], e.g. the influence of an optimistic or pessimistic choice for $D(v, v_c)$. Nonetheless, Equation 3 has been widely employed in recent works [2, 13, 19, 23]; see also references in [11].

2.3. Other Elements of a Test Data Generator

Although the search technique deals with one optimisation problem at a time, the real goal of the approaches following the scheme in Figure 1 is to solve a set of problems. Several works in the literature have taken this into account to improve the process. The alternative suggested by some works is to profit from the good solutions found by not only evaluating an input for the current objective branch, but also with regard to all the others. Thus, each branch is assigned a set containing the best inputs so far. The strategy to select the objective branch consists then of choosing the branch with a highest quality set of inputs. Moreover, for the optimisation step, this set is used to seed the initial phase of the search method [10, 18, 23].

3. Dynamic SBSTDG as Constraint-Handling

As we will next see, many of the SBSTDG generators discussed in the previous section implicitly follow a particular constraint-handling approach. However, we believe an explicit formulation of the problem in term of constraints can provide insights on the behaviour of the generator and it opens the door to techniques that, eventually, may lead to new designs. From now on we restrict ourselves to branch coverage.

3.1. Problem Formulation

The attainment of an objective branch consists of finding an input that makes the control of the programme flow until the branch is exercised. That is, if the branch is represented by an arc (v_c, v_o) in the control flow graph, we look for an input whose execution path contains (v_c, v_o) . Although several such paths may be possible, the critical conditions of v_c indicate the arcs we must follow to achieve v_c (see Section 2.2), i.e. they identify a set of arcs that are common to different paths. To clarify the discussion, we will call this set of arcs a *critical set for v_c* . The coverage of the branches a critical set represents is then a necessary (though not sufficient) restriction to attain the objective branch. In

other words, we can see the coverage of each branch associated to the critical set as a constraint to be fulfilled. Given an arc (v_1, v_2) in a critical set for v_c , and b , the branch in the source code represented by (v_1, v_2) , the constraint value for an input \mathbf{x} is given by

$$g^1(\mathbf{x}) = \begin{cases} \frac{f^1(\mathbf{x})}{M} & \text{if } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

where f^1 is the branch distance in Equation 1 and M is a normalisation term.

Now, given the objective branch b , represented by arc (v_c, v_o) in the control flow graph $G = (V, U)$, and a critical set for v_c , $\{(v_i, v'), v_i \in V, \forall i \in \{1, 2, \dots, n\}, v' \in V\}$, which has n arcs, we can formulate the coverage of b as a constrained optimisation problem [4] (we keep the notation)

$$\begin{aligned} \text{minimise } f(\mathbf{x}) &= \begin{cases} \frac{f^c(\mathbf{x})}{M} & \text{if } b \text{ not attained} \\ 0 & \text{otherwise} \end{cases} \\ \text{s.t. } g^i(\mathbf{x}) &= 0, \quad i = 1, 2, \dots, n, \end{aligned} \quad (5)$$

or as a multiobjective optimisation problem [7]

$$\begin{aligned} \text{minimise } f(\mathbf{x}) &= \begin{cases} \frac{f^c(\mathbf{x})}{M} & \text{if } b \text{ not attained} \\ 0 & \text{otherwise,} \end{cases} \\ \text{minimise } g^i(\mathbf{x}), & \quad i = 1, 2, \dots, n. \end{aligned} \quad (6)$$

In both cases, solutions meeting either f or g^i define an optimal region of points in the search space. A solution to the problem is then a point in the intersection of the optimal regions defined by f and g^i , $\forall i \in \{1, 2, \dots, n\}$. Loosely speaking, according to Equation 5, we seek the best solution (input) among the feasible ones (inputs fulfilling all the constraints), while as for Equation 6, the coverage of the objective branch and the constraints are equally important aims.

It is important to notice that different critical sets might exist for v_c . If so, in order to conform to any of the two previous formulations, first, we would need to choose one of such sets. The strategy employed for this selection might be an issue relevant to the behaviour of the approach, e.g. similarly to the optimistic/pessimistic options for the function in Equation 3. Nonetheless, we leave a study on this matter for future work and, from now on, we assume there is only one critical set for v_c .

3.2. A Constraint-Handling View of Previous Works

The early work by Schoenauer and Xanthakis [20] is the only we have found in the literature explicitly facing this problem by handling constraints. There, the authors develop an approach based on Genetic Algorithms (GAs) for solving a (general) constrained optimisation problem. The

main idea is to apply a GA for solving one of the constraints (g^i in Equation 5) at a time. When dealing with a constraint, the initial population of the GA is seeded with the population resulting from the previous constraint. During the search, if an individual does not fulfil a previously handled constraint then it is penalised with the worst possible value (death penalty [4]). The GA runs until a partial population of individuals fulfilling the current constraint has been obtained. In the last step of the approach, the GA is used to find the optimum of the objective function (f in Equation 5). Again, the initial population is seeded with the partial population obtained from the last constraint. As it can be noted, in this approach, constraints are handled in a particular order, which makes it suitable for problems where such an ordering is naturally imposed. Clearly, this is the case for branch coverage, as it is unknown whether a branch was exercised until a previous in the critical set was covered, i.e. in Equation 5, the value of $g^i(\mathbf{x})$ is unknown unless $g^{i-1}(\mathbf{x}) = 0$. With this motivation the authors build a dynamic SBSTDG generator, obtaining promising results.

Comparing this work with the SBSTDG approaches that conform to Equation 3 a similarity can be observed. The function of the latter generators penalises solutions (inputs) proportionally to how far they are from meeting the last constraint, following the same order as in [20]. Therefore, the search points are encouraged to pursue the optimal regions defined by constraints in this particular order. Depending on the topology of these regions and the functions encoded by the constraints, this demarcation of the path to the optimum may hinder the search. Additionally, such a restrictive way of achieving each constraint might lead to a lack of diversity. Actually, in [20], the authors admit this problem in their approach.

The generators using Equation 2 correspond, from a constrained optimisation point of view, to a death penalty approach. That is, solutions (inputs) not satisfying a constraint are assigned the worst possible value. In this case, low performance issues might arise owing to the lack of guidance towards the intersection of the optimal regions in the search space. Anyway, in spite of its simplicity, this type of penalty functions perform satisfactorily for some problems [4].

All in all, generators conforming to equations 2 or 3 so far follow a constrained optimisation formulation and use two particular penalisation strategies. However, as we will see next, a wider range of approaches can be applied to solve the problem.

3.3. New Approaches

The formulations presented in Section 3.1 suggest the use of strategies for the components of a SBSTDG generator that, to the best of our knowledge, have not been applied before in this context. Namely, we concentrate

on a criterion for the selection step of the generator and on the use of different approaches for the search method. Although these changes lead to new designs, we build upon a test data generator with a set of inputs associated to each branch, as described in Section 2.3.

Branch Selection

Most of the research on dynamic SBSTDG so far has focused on the optimisation step of the scheme in Figure 1 [10, 13, 16, 18, 21, 22, 23]. However, the selection step is an important component of a generator as well, since the strategy implemented defines the order in which the coverage of the branches will be sought.

From the constraints based formulation for the coverage of the objective branch, we see that solving the problem implies exercising all the branches associated to the constraints. Thus, in order to maximise the number of branches covered when the objective branch is attained, we may define a selection criterion that consists of choosing as objective the branch with the largest critical set. In case of tie, we select the branch with a highest quality set of inputs among the tied branches.

Search Methods

If we were able to overcome the restriction of following the order naturally given by the problem, i.e. by the critical set, we would dramatically open the range of search techniques that can be applied to the test data generator. This might enable to alleviate some of the limitations of the SBSTDG generators we have discussed.

Actually, this can be achieved through the *testability transformation* presented in [12]. Such a transformation is a controlled modification of the source code of the programme which aims at improving some aspect of a test data generator. The testability transformation proposed in [12] consists of a particular instrumentation of the source code that allows to obtain the branch distance value for every critical condition associated to the objective branch. The main idea to achieve this is to remove the conditional statements corresponding to the critical conditions of the objective branch, and to compute the branch distance instead. This way, we can calculate the value for any constraint g^i and for the objective function f in equations 5 or 6. Figure 2 gives an example of the type of instrumentation presented in [12]. Only one critical condition is considered, which corresponds to the branch distance f^2 .

It is worth to remark, however, that this instrumentation might pose some issues for certain conditional statements. For instance, it might be the case the branch distance is not defined (has no value) for the input at hand. In this situation, we could choose to return the worst possible value for that condition. For details regarding further issues, the reader is referred to [12].

Having a means to calculate any of the values in equations 5 and 6 for any programme input, we can use the techniques developed in the fields of constrained and multi-objective optimisation. However, the general problem of meeting a set of constraints is known to be NP-complete [9], which has motivated the widespread use of heuristic methods and, in particular, Evolutionary Algorithms. Since virtually any function may be encoded in a condition and, hence, in the branch distance, we may assume the same complexity for the general case of branch coverage when following a constraints-handling formulation.

In our case, we fix the search method to be a simple GA with truncation selection, one point crossover and a mutation operator that consists of substituting the corresponding allele with a random number. As a case study, we propose several penalty functions [4] for tackling the constrained optimisation problem in Equation 5, and different Pareto rankings [7] for facing the multiobjective optimisation problem in Equation 6. Next, we keep using the notation in Section 3.1.

Penalty Functions

One common approach to deal with constrained optimisation problems is to penalise solutions not fulfilling some of the constraints. That is, the objective function value is added a certain value based on the amount of constraint violation. This way, the constrained optimisation problem is transformed into an unconstrained one, which is a suitable scenario for an Evolutionary Algorithm [1].

A large number of penalty functions have been proposed in the literature (see [4] and references therein). Amongst the most widely used are *static* functions, which penalise by just taking into account the constraint value of the solution. We adopt two static functions, f^{s1} and f^{s2} :

$$\begin{aligned} f^{s1}(\mathbf{x}) &= f(\mathbf{x}) + \sum_{i=1}^n \delta_i(\mathbf{x}), & \delta_i(\mathbf{x}) &= \begin{cases} 1 & g^i(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases} \\ f^{s2}(\mathbf{x}) &= f(\mathbf{x}) + \sum_{i=1}^n (1 + g^i(\mathbf{x})) \end{aligned}$$

It can be noted that the penalty of f^{s1} is simply the number of violated constraints, while for f^{s2} the penalisation is given by the amount of violation of each constraint.

A more sophisticated type of penalty is given by *dynamic* functions, which incorporate the current generation number. We consider the following function:

$$f^d(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^n (1 + (t + 1) \cdot g^i(\mathbf{x}))$$

where t is the current generation number in the GA. The idea underlying this function is to be less strict in the early stages of the GA in order to allow for exploration, and to be more restrictive as the search progresses.

The last type of penalty function we have used is an

<pre> void example (int a, int b) (1){ (2) if (a<b) (3) if (a*a-b+5==0) (4) // objective branch (5)} </pre>	<pre> void example_transformed (int a, int b) (1){ (2) compute_bd(a<b); ← returns $f^2(a, b)$ (3) compute_bd(a*a-b+5==0); ← returns $f^3(a, b)$ (4) // objective branch (5)} </pre>
---	---

Figure 2. Example of the type of instrumentation proposed in [12]. Original programme to the left and transformed programme to the right.

adaptive function:

$$f^a(\mathbf{x}) = f(\mathbf{x}) + \lambda(t) \sum_{i=1}^n g^i(\mathbf{x}),$$

$$\lambda(t+1) = \begin{cases} (1/\beta_1) \cdot \lambda(t) & \text{if case 1} \\ \beta_2 \cdot \lambda(t) & \text{if case 2} \\ \lambda(t) & \text{otherwise} \end{cases}$$

where cases 1 and 2 denote situations where the best individual in the last 5 generations was always (case 1) or was never (case 2) feasible, $\beta_1 > \beta_2 > 1$. That is, this function allows either an increase or a decrease of the penalty during evolution based on the feasibility of the best solution in the last generations (in our case, 5).

Pareto Rankings

A key concept to evaluate the optimality of a solution in a multiobjective optimisation problem is *dominance* [7]. Informally, a solution \mathbf{x} dominates another solution \mathbf{x}' if it is better than \mathbf{x}' for at least one of the objectives and it is equally good to \mathbf{x}' for the rest. By contrast, if no solution dominates \mathbf{x}' then we say it is non-dominated. Thus, in multiobjective optimisation, we seek the set of non-dominated solutions, which is known as the *Pareto-optimal* set.

In the context of multiobjective Evolutionary Algorithms, it is common to define the fitness of a solution according to a Pareto ranking [7], *PT1*, where the rank value for a solution is given by the number of solutions in the population that dominate it.

However, as it has been suggested by some authors [4], this ranking suffers from a lack of guidance to the optimal regions of the search space. In order to bias the search towards these regions, we propose the following variant of *PT1*. First, rank solutions according to *PT1*. Once finished, make a second ranking among the solutions with the same value in the first ranking; in this second ranking, the value for a solution \mathbf{x} is given, summing over the solutions \mathbf{x}' that dominate it, by the number of objectives by which \mathbf{x}' dominates \mathbf{x} . It is important to remark that we implement this approach hierarchically, that is, if a solution \mathbf{x} has a higher rank than \mathbf{x}' in *PT1*, it will keep having a higher

value after the second ranking. We denote this strategy by *PT2*.

As it may be noticed, using either *PT1* or *PT2*, non-dominated solutions in the population receive a rank value of 0. As a result, there is no guidance regarding these solutions, which might lead the population to converge towards undesirable regions of the search space. For instance, a solution might be non-dominated because evaluates to an outstanding value for one of the objectives, whilst having a bad evaluation for the rest. So, in order to bias the search towards the intersection of all the optimal regions (which is where optima are in our problem), we develop one more ranking, *PT3*. This ranking is as *PT2* but, if a solution is not dominated by any other, then we evaluate it with the sum of the objective function values, i.e. $f(\mathbf{x}) + \sum_{i=1}^n g^i(\mathbf{x})$. Similarly to *PT2*, we implement this approach in a hierarchical way.

4. Empirical Evaluation

We conducted a preliminary evaluation of the proposed test data generators to see whether they might be a competitive option with regard to generators so far. For this, we compare the performance of the generators in three programmes typically used in the literature; namely, *Triangle*, *Atof* and *Remainder* [23]. *Triangle* owns 26 branches and a maximum number of critical conditions (constraints) of 6, *Atof* has 30 branches and a maximum of 8 critical conditions, and *Remainder* has 18 branches and a maximum of 4 critical conditions. All the experiments were run for three population sizes: 50, 100 and 150. The rest of the parameters of the GA were kept constant for every programme and test data generator. So, selection pressure was set to 0.5, probability of crossover was 1, probability of mutation was 0.1 and maximum number of generations was set to 50.

Tables 1 to 3 show, for each generator, the average results of 30 runs for two measurements: the coverage percentage (%) and the number of generated inputs throughout the whole process (#). The best results for each measurement are marked in bold. f^{cl} and f^{ad} refer to the generators with the classical (Equation 2) and advanced (Equation 3)

functions respectively, both following the approach with a set of inputs associated to each branch (Section 2.3). The rest of generators are denoted as in Section 3.3.

		50	100	150
f^{cl}	%	93.97	94.1	94.36
	#	9813	17177	22650
f^{ad}	%	94.23	93.85	94.36
	#	9548	18793	22280
f^{s1}	%	93.59	94.1	94.62
	#	10137	18187	20975
f^{s2}	%	93.08	93.72	95.39
	#	10497	16337	19835
f^d	%	92.95	94.49	94.36
	#	9662	12907	22430
f^a	%	93.21	94.36	94.49
	#	8803	16910	25115
$PT1$	%	92.56	92.69	93.08
	#	5956	11314	16227
$PT2$	%	92.44	92.69	93.46
	#	6060	11348	15917
$PT3$	%	93.08	92.82	92.69
	#	5768	11215	16684

Table 1. Results for Triangle.

Regarding the classical (f^{cl}) and advanced (f^{ad}) generators, it can be observed they both behave similarly, except for $At\ of$ with a population of 50 individuals, where the f^{ad} clearly improves f^{cl} .

As for the penalty functions, the results for the coverage are again similar, though a difference can be appreciated for the number of inputs. Unfortunately, we cannot draw any conclusion on the superiority of a particular approach, since the best values for # alternate among the functions and vary with the programme. Several works in the literature have alluded to the unpredictability of penalty functions [1, 4, 17] and their dependency on the parameters associated to the penalty terms. For instance, it is common to assign a weight to these penalty terms. Ideally, values for the weights should be chosen upon information of the problem at hand. However, for the sake of simplicity, we have devised the programme as a black box and, so, each term corresponded a weight of 1. In other words, in order to find the most suitable penalty function for a programme the use of a-priori knowledge is advisable.

Concerning the multiobjective approaches, a clear dissimilarity can be observed between the coverage of $PT3$ and the two other rankings in $At\ of$; in this programme $PT3$ clearly improves $PT1$ and $PT2$. No relevant difference is seen in the two other programmes however. So, it seems that $PT3$ can make a difference, though, as with the

		50	100	150
f^{cl}	%	93.44	99.89	100
	#	8238	5350	6755
f^{ad}	%	99	99.89	100
	#	4713	5347	5920
f^{s1}	%	99.22	99.78	100
	#	6720	7837	6530
f^{s2}	%	99.44	99.89	100
	#	5295	5417	5040
f^d	%	99.11	99.78	99.78
	#	6235	6433	7090
f^a	%	99.56	100	99.89
	#	6020	8587	9305
$PT1$	%	81.78	92.33	93.78
	#	20324	20538	26169
$PT2$	%	85.67	91.22	94.67
	#	16377	22466	22374
$PT3$	%	93.56	97.89	98.56
	#	11560	10699	13915

Table 2. Results for $At\ of$.

penalty functions, this depends to a large extent on the programme at hand. Again, owing to the arbitrary complexity of branch coverage, the use of a-priori information from the programme presents as an important requirement to obtain a well suited approach.

In general terms, it can be observed that no clear difference appears to be among all the generators in terms of coverage. An exception to this, however, is the case of multiobjective based generators for programme $At\ of$, where their coverage looks lower than for the others. This result conforms to those of authors [17] in other domains, which suggest multiobjective optimisation is not suitable for handling constraints at the present stage of research. Nonetheless, as it may be noted, the best results in terms of the number of inputs generated belong to multiobjective approaches. All in all, the best result values are given by either a penalty function or a Pareto ranking.

5. Conclusions

In the present work, we have formulated the SBSTDG based dynamic test data generation for branch coverage as a constraint-handling problem. This formulation let us apply a criterion for the selection step and search strategies that, to the best of our knowledge, had not been used to solve this problem yet. More precisely, we developed test data generators employing a Genetic Algorithm with several penalty objective functions and Pareto rankings. The results of the conducted experiments, though not conclusive at all, sug-

		50	100	150
f^{cl}	%	89.07	88.89	89.07
	#	6967	13433	19650
f^{ad}	%	88.89	88.89	89.44
	#	7217	13767	18870
f^{s1}	%	88.89	89.44	89.07
	#	7550	12720	20000
f^{s2}	%	89.07	89.63	88.89
	#	6612	13527	21900
f^d	%	88.89	89.44	89.44
	#	7633	12790	18905
f^a	%	89.07	88.89	89.07
	#	6467	15100	22140
PT1	%	89.07	88.89	89.07
	#	5089	10268	15251
PT2	%	88.89	88.89	89.26
	#	5170	10267	14866
PT3	%	89.07	89.44	89.26
	#	5103	10018	15168

Table 3. Results for Remainder.

gest constraints based generators can improve the performance of approaches so far. Much work needs to be done however. For instance, it would be interesting to shed some light on how the functions encoded in the conditional statements influence the constraints based generator. Also in this line, the interactions between the regions defined by those functions are an important topic to research. Since the constraint handling techniques employed here are rather simple, another line for future work could be the exploration of more sophisticated techniques. On the one hand, the functions encoded in the conditions may be arbitrarily complex, so we believe that, in order to achieve the best performance, information on the programme at hand should be obtained and used for the search, e.g. by making a previous static analysis of the source code. On the other hand, there is a vast literature on constraint-handling and multiobjective optimisation. By following the formulation presented here, we could benefit from all this existing knowledge to create improved designs.

Acknowledgment

This work is supported by an EPSRC grant (EP/D052785/1). The authors also wish to thank Per Kristian Lehre and Andrea Arcuri for the useful comments provided during the preparation of this work.

References

- [1] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. *Evolutionary Computation 2. Advanced Algorithms and Operators*. Institute of Physics Publishing, Bristol, UK, 2000.
- [2] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Mateo, CA, 2002. Morgan Kaufmann.
- [3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.
- [4] C. A. Coello. Theoretical and numerical constraint-handling techniques used in evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11–12):1245–1287, 2002.
- [5] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.
- [6] N. E. Fenton. The structural complexity of flowgraphs. In Y. Alavy, G. Chartrand, L. Lesniak, D. R. Lick, and C. E. Wall, editors, *Graph Theory with Applications to Algorithms and Computer Science*, pages 273–282. John Wiley & Sons, New York, 1985.
- [7] C. M. Fonseca and P. J. Flemming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms - part i: A unified approach. *IEEE Transactions on System, Man, and Cybernetics - Part A: Systems and Humans*, 28(1):26–37, 1998.
- [8] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
- [9] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [10] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.
- [11] P. McMinn. Search-based software test data generation: a survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.
- [12] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*. To appear.
- [13] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In K. Deb, R. Poli, W. Banzhaf, H. G. Beyer, E. K. Burke, P. J. Darwen, D. Dasgupta, D. Floreano, J. A. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. M. Tyrrell, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1363–1374, Seattle, WA, 2004. Springer.
- [14] W. Miller and D. Spooner. Automatic generating of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

- [15] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software - Practice and Experience*, 29(2):167–193, 1999.
- [16] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(4):263–282, 1999.
- [17] T. P. Runarsson and X. Yao. Search biases in constrained evolutionary optimization. *IEEE Transactions on System, Man, and Cybernetics: Part C*, 35(2):233–243, 2005.
- [18] R. Sagarna and J. A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence*, 19(5):457–489, 2005.
- [19] R. Sagarna and J. A. Lozano. Dynamic search space transformations for software test data generation. *Computational Intelligence*, 24(1):23–61, 2008.
- [20] M. Schoenauer and S. Xanthakis. Constrained ga optimization. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 573–580, San Mateo, CA, 1993. Morgan Kaufmann.
- [21] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.
- [22] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In D. Redmiles and B. Nuseibeh, editors, *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 285–288. IEEE CS Press, 1998.
- [23] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.