Diversifying Focused Testing for Unit Testing

HÉCTOR D. MENÉNDEZ*, Middlesex University London
GUNEL JAHANGIROVA, Università della Svizzera italiana
FEDERICA SARRO, University College London
PAOLO TONELLA, Università della Svizzera italiana
DAVID CLARK, University College London

Software changes constantly because developers add new features or modifications. This directly affects the effectiveness of the test suite associated with that software, especially when these new modifications are in a specific area that no test case covers. This paper tackles the problem of generating a high quality test suite to cover repeatedly a given point in a program, with the ultimate goal of exposing faults possibly affecting the given program point. Both search based software testing and constraint solving offer ready, but low quality, solutions to this: ideally a *maximally diverse* covering test set is required whereas search and constraint solving tend to generate test sets with biased distributions. Our approach, Diversified Focused Testing (DFT), uses a search strategy inspired by GödelTest. We artificially inject parameters into the code branching conditions and use a bi-objective search algorithm to find diverse inputs by perturbing the injected parameters, while keeping the path conditions still satisfiable. Our results demonstrate that our technique, DFT, is able to cover a desired point in the code at least 90% of the time. Moreover, adding diversity improves the bug detection and the mutation killing abilities of the test suites. We show that DFT achieves better results than focused testing, symbolic execution and random testing by achieving from 3% to 70% improvement in mutation score and up to 100% improvement in fault detection across 105 software subjects.

CCS Concepts: • **Software verification and validation → Empirical software validation**;

Additional Key Words and Phrases: Testing, Focused Testing, GödelTest, Diversity, DFT

ACM Reference Format:
Héctor D. Menéndez, Gunel Jahangirova, Federica Sarro, Paolo Tonella, and David Clark. 2021. Diversifying Focused Testing for Unit Testing. 1, 1 (March 2021), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

State-of-the-art automated testing techniques rely on coverage and random-based test case generation as the most competitive strategies to detect bugs in programs [5]. Although these strategies are, indeed, effective, they lack some attributes that current software development processes require. Nowadays many software projects are in constant

*Corresponding author.

Authors' addresses: Héctor D. Menéndez, h.menendez@mdx.ac.uk, Middlesex University London, The Burroughs, Hendon, London, UK, NW4 4BG; Gunel Jahangirova, gunel.jahangirova@usi.ch, Università della Svizzera italiana, Via Buffi, Lugano, Svizzera, 6900; Federica Sarro, f.sarro@ucl.ac.uk, University College London, Gower Street, London, UK, WC1E 6BT; Paolo Tonella, paolo.tonella@usi.ch, Università della Svizzera italiana, Via Buffi, Lugano, Svizzera, 6900; David Clark, david.clark@ucl.ac.uk, University College London, Gower Street, London, UK, WC1E 6BT.

development and programs grow every day. These modifications often bring new code into existing programs and this code might not be covered adequately by the current test suite of the project. Thus, it is relevant to automatically create test suites focused on specific program parts. These test suites will be able to, for example, help analyse new/changed code or compare new and previous functionality of specific code sections, in order to guarantee that such code is not introducing bugs into the system and it is properly integrated. Coverage strategies alone have limited abilities in detecting real faults, as covering a faulty statement once does not necessarily imply that the fault will be activated, i.e., the program state will be infected [51].

To address this problem, we need to focus the testing process on the components added/modified by the developer. This methodology, called focused testing (Section 2), aims at guaranteeing that multiple tests target a few, specific elements of the program. There are different approaches to focused testing, such as techniques for detecting triggers in malware [58], or those generating directed test cases by traversing specific program elements [2]. However, triggering techniques only generate a single test case [58], while directed test cases come with a high computational cost due to the need to discard possibly many tests during the generation process [2]. Moreover, these techniques are normally guided just by coverage and, therefore, they do not provide any warranty on the diversity of the inputs [26]. Indeed, diversifying the test suite has been shown to improve its fault detection ability, even when the test suite size is small [20].

In this paper, we extend the idea of focused testing by including diversity in the generation process. Our approach leverages the idea behind GödelTest [19], a testing technique based on parametrised test suite generators. We name our technique Diversified Focused Testing (DFT). While the GödelTest approach requires the parametrised generators to be created manually, we produce generators automatically. Moreover, our approach aims at satisfying two objectives at the same time: (1) maximum diversity; and, (2) satisfiability of the condition to reach the target. To the best of our knowledge, our proposal is the first meta-generator technique that can automatically create a parametrised test generator for a given program and a set of program elements (Section 3.1), ensuring diversified focused coverage.

DFT focuses the testing process on a specific target by extracting the program constraints associated with the target. The extraction process is performed by a C-Bounded Model Checker (CBMC), which provides a set of Satisfiability Modulo Theory (SMT) constraints in a bit-vector arithmetic (Section 4.1). The automatically synthesised generators add parameters to the extracted SMT constraints. These parameters have the ability to allow alternative paths to reach the program points of interest, and, at the same time, the ability to randomise the selected inputs (Section 3.1). We optimise these parameters in two directions. The first direction aims to guarantee uniformity, which is our target measure of diversity. This optimisation moves the test case probability distribution associated with the generator closer to a uniform distribution, using statistical tests that quantify such closeness (Section 3.2). The second direction aims to optimise the efficiency of the test generator by reducing the creation of potentially unsatisfiable constraints, a possible consequence of the parametrisation process (Section 3.2). DFT optimises the added parameters using two multi-objective search algorithms: a greedy search and an evolutionary algorithm (Section 4.2).

We evaluate DFT on 105 software subjects drawn from three different sources: R-Project [44], SIR [45] and CodeFlaws [56] (Section 6). Experiments show that our method creates inputs that reach the target program point at least 90% of the time and that these inputs are strongly diverse (Section 7.1). DFT also improves the mutation score by between 3% and 70% and fault detection by up to 100% compared to state-of-the-art approaches in focused and random testing. Moreover, our optimisation process improves the quality of the generator by reducing its error rate (due to unsatisfiable constraints) by an order of magnitude. The main contributions of this paper are:

(1) We propose the first meta-generator for diversified focused testing (DFT) [1], which automatically parametrises the SMT constraints associated with a targeted execution. Such automatically injected parameters are optimised for diversity and satisfiability by a multi-objective algorithm (Section 3).
(2) We make publicly available an open-source implementation of DFT, providing a focused generator guided by diversity and based on two multi-objective search-based strategies that improve the generator's efficiency, while preserving diversity/uniformity of the selected inputs (Section 4).
(3) We provide empirical evidence showing the effectiveness of DFT in improving reachability and diversity in different programs, along with fault detection and mutation killing capabilities (Section 7).

2 PRELIMINARIES

Focused random testing aims to test specific, individual components of a program [2]. In the original work of Alipour et al. [2], the authors define focused testing as a black box method whose aim is to reach a specific target. For example, a target may be the activation of a specific API. Alipour et al. automatically select the parameters of a general test generator to reach that target, by activating or deactivating different options of the generator. Our work increases the granularity of this approach, considering a white box scenario. The components we consider are program points (*pp*), i.e., specific nodes in the control flow graph. Instead of using general purpose generators, our meta-generator automatically creates a generator to test specific *pp*s in programs. The resulting generators are based on SMT solvers, which are well known to affect diversity negatively [11], i.e., they tend to generate inputs which are very similar to each other and are not uniformly spread in the space of possible inputs. Therefore, our main goal is to add diversity to our white box version of the focused random testing process, by making it create, for any program (P) and program point ($pp \in P$), a diverse test suite passing through pp.

Creating a test input generator for a program is a complex process, and it normally requires human intervention [19]. However, based on a representation of the program path conditions (or a bounded subset of them) in conjunctive normal form (CNF), we can leverage SAT or SMT solvers for the meta-generation task [11, 28]. This methodology transforms the program paths into a CNF formula f. In our case, we *focus* this formula on the program point we aim to traverse, considering only those paths, and the associated path expressions, that reach pp (for details about the formula extraction process, see Section 4.1). We denote this formula as f_{pp}.

Given the space of inputs $\mathcal{X}$, let us consider the sub-space of inputs that traverse pp, denoted as $\mathcal{X}_{pp}$. An input $x \in \mathcal{X}$ belongs to $\mathcal{X}_{pp}$ if $f_{pp}(x) = True$, i.e., x is a witness for f_{pp}. The formula gives us the ability to test whether an input passes through pp and solving it gives us the ability to generate inputs passing through pp. In addition to this, we want our generator to produce a *diverse* set of inputs that satisfy f_{pp}.

Although diversity is a well-known concept in the literature, it does not have a precise definition (Section 9). Following the work of Chakraborty et al. [11], we define diversity using entropy, considering a diverse set as a high-entropic one. We define a generator G as an algorithm that creates inputs for a program P. We define a focused generator G_{pp} as a generator whose generated inputs traverse a specific program point pp. Considering the generator as a random variable whose values are inputs traversing pp, we want to maximise the entropy of its probability distribution to make it diverse. According to Theorem 2.6.4 [16], the entropy of a random variable is maximum when its probability distribution is uniform. Therefore, the goal of creating a diverse focused generator is equivalent to creating a uniform

[1] The tool is publicly available in https://github.com/hdg7/DFT

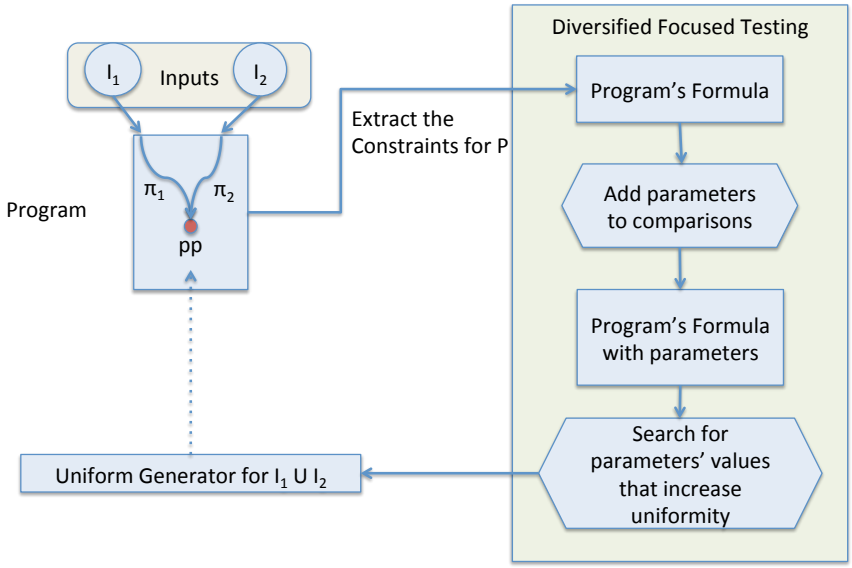

Fig. 1. DFT extracts the program constraints to reach program point *pp*. These are represented as a *program's formula*. It afterwards parametrises the formula. Once parameters are included into the formula, it creates a generator for finding suitable values for these parameters, with a twofold goal: (1) satisfying the original path constraints; (2) sampling diverse inputs uniformly.

focused generator. A uniform focused generator gives the same generation probability to every input in $\mathcal{X}_{pp}$, i.e., $p(G_{pp}(f_{pp}) = x) = 1/|\mathcal{X}_{pp}|, \forall x \in \mathcal{X}_{pp}$.

3 DIVERSIFYING FOCUSED TESTING

Our focused diversification process aims to create a parametrised focused generator by automatically including parameters in its program constraints [19]. Then, these parameters are optimised to achieve diversity.

Figure 1 shows the general overview of this process. DFT starts with a program P and a program point pp it aims to traverse. The goal is to select the inputs traversing pp uniformly at random from all the possible inputs. The generation process starts with the extraction of the formula f_{pp} from the program. Then, it parametrises the formula and, finally, it optimises the parameter values so as to obtain a diverse set of witnesses via SMT solving. DFT follows a similar strategy to GödelTest [19]. GödelTest is divided into two steps: the programmer creates a parametrised generator for the program manually and, then the system applies search to find values for parameter values that follow a fitness function. It assumes that the programmer creates a non-deterministic generator. The non-determinism is controlled by some parameters affecting values or paths explored during the execution of the generator. DFT creates the parametrised generator automatically in three steps (Section 4.1): 1) it extracts the program constraints for the paths reaching pp, 2) it selects the comparison operators on these constraints and adds parameters to them and 3) it creates a final formula with the parametrised constraints. Once, the parametrised generator is available, DFT applies search to find suitable parameters that allow the selection of pp-traversing inputs uniformly at random (Section 4.2).

We distinguish between the parameters affecting the execution flow and the ones exploring the input space. We call them *path selection parameters* and *partition exploration parameters*, respectively. Our approach creates the parametrised

Original Program

```
#include<stdio.h>
int main(int argc, char *argv[])
{
  int x,y,c;
  scanf("%d %d",&x,&y);
  c=x*y;
  if(c>5 || (x<5 && c == 3)) c=y;//pp
  else c=x;
  return 0;
}
```

Program Constraints for pp

$$f_{pp}(x,y) = \begin{cases} f^V_{pp}(x,y) & = & \wedge \begin{cases} guard_1 & = & x \cdot y > 5 \\ guard_2 & = & x < 5 \\ guard_3 & = & x \cdot y = 3 \end{cases} \\ f^\pi_{pp}(x,y) & = & \vee \begin{cases} \pi_1 : guard_1 \\ \pi_2 : guard_2 \wedge guard_3 \end{cases} \end{cases}$$

Final formula with parameters

$$f^*_{pp}(x,y) = \begin{cases} f^V_{pp}(x,y) & = & \wedge \begin{cases} guard_1 & = & x \cdot y > 5 \\ guard_2 & = & x < 5 \\ guard_3 & = & x \cdot y = 3 \\ e_1 & = & y = e^1_{val} \\ e_2 & = & x \cdot y = e^2_{val} \\ e_3 & = & x = e^3_{val} \end{cases} \\ f^\pi_{pp}(x,y) & = & \mathbb{1} \begin{cases} guard_1\ [p(\pi_1)] \\ guard_2 \wedge guard_3\ [p(\pi_2)] \end{cases} \\ Exps & = & DoF \begin{cases} e_1\ [p(e_1)] \\ e_2\ [p(e_2)] \\ e_3\ [p(e_3)] \end{cases} \\ e^1_{val} & = & rand(e^1_{\min}, e^1_{\max}) \\ \dots & & \\ e^3_{val} & = & rand(e^3_{\min}, e^3_{\max}) \end{cases}$$

Fig. 2. Example of a program, the formula f_{pp} related to the program point pp, and the manipulated formula including parameters f^*_{pp}.

generator automatically, by adding these two kinds of parameters to f_{pp}, hence transforming the deterministic formula into a parametrised, non-deterministic input generator.

3.1 Creating an Automatic Generator

While a solver can understand the f_{pp} formula and is able to generate inputs for it, as Chakraborty et al. noted [11], the solver's heuristics do not generate diverse/uniformly distributed witnesses. To achieve this goal, we transform f_{pp} into a parametrised generator automatically.

First, we separate f_{pp} into two sets of expressions $f_{pp} = f^{\pi}_{pp} \wedge f^{V}_{pp}$, where f^{π}_{pp} are the path selection expressions and f^{V}_{pp} are the expressions that describe the constraints on the program variables collected on each path (Figure 2). For example, in Figure 2 we have x and y as inputs and pp can be reached under two alternative path conditions: $x \cdot y > 5$ or $x < 5 \wedge x \cdot y = 3$. The path selection expressions f^{π}_{pp} can be obtained by combining the expressions for the individual paths in an `Or` tree. Hence, f^{π}_{pp} can be decomposed as $f^{\pi}_{pp} = \bigvee_i \pi_i$, where π_i is the path condition of a path passing through pp. This allows different executions to follow the same path, under different inputs. To decide which path is chosen, we add a probability to each path π_i, satisfying $\sum_i p(\pi_i) = 1$. Each $p(\pi_i)$ defines a *path selection parameter*. In the example above, we have two path selection parameters: $p(guard_1)$ and $p(guard_2 \wedge guard_3)$.

The variant expression f^{V}_{pp} contains the (possibly unsatisfiable) conjunction of all path condition guards. From the variant expressions f^{V}_{pp}, we extract those expressions that have an inequality involving an input and a constant value, $x \cdot y > 5$, for example. Then, we replace the constant with a parametrised value, and create an expression e replacing the initial inequality, getting $x \cdot y == e_{val}$. We perform the same operation with the individual inputs, e.g. $y == e_{val}$. Some (a subset) of these expressions are set as constraints: given a fixed path to be traversed, these parametrised expressions force the solver to explore different areas of the input space.

The total degrees of freedom (DoF) determines the maximum number of parametrised expressions (e) to be chosen. DoF is another parameter to be optimised by the algorithm to ensure satisfiability: adding more expressions than DoF may produce an unsatisfiable constraint. An expression e has an associated probability $p(e)$, which is its selection probability. Once an expression e is selected, its value e_{val} is uniformly selected in the range $[e_{\text{min}}, e_{\text{max}}]$. The tuple $(p(e), e_{\text{min}}, e_{\text{max}})$ is called a *partition exploration parameter* tuple. The effect of introducing expression e is a reduction in the degrees of freedom of the whole equation system, forcing the solver to return a smaller set of valid witnesses. By properly selecting the parameters of e, the solver can spread the exploration of valid inputs (witnesses) during the test generation process.

The final generator is a variation of the original formula, denoted as f^{*}_{pp}, including the following parameters:

(1) **Degrees of Freedom (DoF)**: up to the number of input variables. This defines how many inequalities can be fixed before they are submitted to the solver.
(2) **Path Selection Parameters**: a set of probabilities $p(\pi_i)$ indicating which path is more/less likely to be chosen.
(3) **Expression Variation Parameters**: a tuple formed of three values $(p(e), e_{min}, e_{max})$.

In Figure 2, f^{*}_{pp} includes three new expressions, e_1, e_2 and e_3, added to f^{V}_{pp}. They control the inputs and the inequalities associated with the guards. The values e_{val} of these expressions are set uniformly at random before they are included. As the program has two input variables, there is a maximum of 2 degrees of freedom. Therefore, the parameter *DoF* will be between 1 and 2 (total number of input variables). DoF determines how many new constraints are activated from the expression set ($Exps = \{e_1, e_2, e_3\}$). The probability of activating each of these expressions is $p(e_i)$. Finally, only one path will be activated from $f^{\pi}_{pp}(x, y)$, i.e. either π_1 or π_2 ($\mathbb{1}$ indicates Kronecker's delta in Figure 2). The probability of selecting each path is denoted in the example as $p(\pi_i)$. Figure 2 shows the final expression of f^{*}_{pp}.

3.2 Parameter Restrictions

The previous step creates a parametrised generator from a formula. The next step of our approach aims to find proper parameters whose associated constraints keep f^{*}_{pp} satisfiable, while the generated inputs are diverse, i.e. follow a uniform distribution (Section 2). The first condition (satisfiability) requires keeping control on the generator error.

Algorithm 1 L2-test for uniformity

Require: A sample $\bar{x}$ of p with, at least, m elements; a domain $[n]$ and an epsilon ϵ
Ensure: "Pass", i.e., $||p - U_{[n]}||_2^2 < \epsilon^2/(2n)$; or "Fail", i.e., $||p - U_{[n]}||_2^2 \geq \epsilon^2/n$
1: Define σ_{ij} as a Kronecker delta of $\bar{x}$, i.e., 1 when samples i and j are equal and 0 otherwise.
2: Let $s = \sum_{i<j} \sigma_{ij}$
3: Let $t = \binom{m}{2} \frac{1+3\epsilon^2/4}{n}$
4: **if** $s \geq t$ **return** "Fail"; **else return** "Pass"

As we add new constraints to f^*_{pp}, the generator might return an error (UNSAT) at some point. It is important to reduce/eliminate this error.

For the second condition (diverse inputs), we need a way to quantify how close G_{pp} is to generating samples from a uniform distribution (Section 2). The literature around uniformity reports different statistical tests measuring this closeness [36]. These tests are divided into several different categories, such as: order statistics, spacing, order spacing and collisions. Some of them are not applicable to discrete distributions [36], and others can not deal with gaps in the domain [42], which is common for constrained domains. For instance, many uniformity tests will not accept a sample $\{1, 4, 16\}$ as uniform, because the test assumes that the domain is the continuous range $[1 - 16]$. During the sampling process, the probability of individual values 2, 3, etc. would be 0, while in a discrete domain we expect a uniform probability of $1/n$, where n is the domain size. Only tests based on collisions can effectively deal with both gaps and discrete distributions [27]. Some of the collision-based tests, such as the L2-test [27], also provide information about the distance between the uniform distribution and the sampled distribution. As our aim is to measure, in practice, whether the produced input set is acceptably close to a uniform distribution, we use the L2-test in our approach.

The test starts with a distribution p that is compared with a uniform distribution $U_{[n]}$ over the domain $[n]$. The test checks whether the distribution p is ϵ-far from uniformity, i.e. it uses the Euclidean distance (or L2-norm) to determine whether $\|p - U_{[n]}\|_2^2 < \epsilon^2/(2n)$ or $\|p - U_{[n]}\|_2^2 \geq \epsilon^2/n$. In the former case, the distribution *passes* the test while, in the latter, it *fails*. Lemma 5 in the study of Diakonikolas et al. [18] bounds the number of samples to $m \leq 6n^{1/2}/\epsilon^2$. Considering this assumption, the test can be computed following Algorithm 1, with an error probability of $1/4$ (see [18], Lemma 5). The test, initially, counts the internal sample collisions (lines 1 and 2). It sets a threshold (line 3) and considers that the test passes when the number of collisions is lower than the threshold (line 4). We use the rate between the threshold and the collisions to guide the search, aiming to reduce the number of collisions to under the threshold.

The two considered restrictions (satisfiability and uniformity) give us a direct way to measure the quality of the parametrisations of the generator. We use them as fitness functions, when applying different search strategies to tune the parameters. Given a parametrisation, the generator's solver produces a test suite (i.e., a set of solutions) *sol* associated with the bi-objective fitness (Eq. 1):

$$fitness(sol) = \left\{ \frac{TotSat}{TotCalls}, 1 - \frac{1}{t/s + 1} \right\}, \tag{1}$$

where $TotSat$ measures the number of satisfiable calls to the solver, $TotalCalls$ the total number of calls, t is the threshold of the L2-Test and s is the number of collisions (Algorithm 1). We normalise the ratio t/s between 0 and 1 and take the complement, such that 1 indicates maximum fitness or minimum collisions. Any multi (bi-) objective search-based metaheuristic algorithm can be used with these two fitness measures to optimise the generator's parameters. In Section 4.2, we define two such search strategies and benchmark them against random search (Section 7).

4 DFT: THE DIVERSIFIED FOCUSED TESTER

We have implemented our approach for diversifying focused random testing (Section 3) as a tool for C programs, named Diversified Focused Testing (DFT). Its current version can be applied to numerical programs without pointer inputs. The following subsections describe the details of the formula extraction process and the search algorithm implementation.

4.1 Formula Extraction Process

Our analysis starts with the identification of possible input variables in the program. Potential inputs are global variables, external variables or read instructions, such as `scanf`. We create a new function header and include all of these input channels in it, so that they can be considered the actual inputs to the program. The `main` function is a special case. If it does not use any program argument, it is discarded from the instrumented header, and a new header is constructed. In the example in Figure 2, we remove the arguments of `main`, as they are not used anywhere inside the function, and add variables `a` and `b` to the instrumented header, as they are a part of a `scanf` expression.

We extract the formula f_{pp} using CBMC [34]. This restricts the programs that are currently handled by DFT to the ones written in C. Other engines such as Java PathFinder [30] or Kudzu [49] may be used to extend the applicability of DFT to more programming languages.

Given the program P, and a program point $pp \in P$, we automatically instrument the program to generate an invalid assert condition immediately before pp. This process was initially introduced by Angeletti et al. [7], who used it to generate tests suites with CBMC. Following the example of Figure 2, our method adds an `assert(0)` statement at line 7, after the expression `c=y`, still inside the `if` then-block. When the block contains only one statement, our instrumentation transforms it from a simple statement to a compound block. Once CBMC triggers the verification error, it provides the verification conditions for the specific program point, distinguishing every possible path traversing that point. CBMC creates these verification conditions using symbolic execution. We set the output of CBMC to be in SMT-LIB version 2 format [8], so that we are able to apply an SMT solver using bit-vector arithmetic to handle such output. The SMT solver we use is Z3 [17].

Initially, CBMC prepares the program for its symbolic execution process, translating expressions into single static assignment and unwinding the loops up to a specific level. The unwinding might reduce the number of constraints related to the loops. This can create an over-approximation of the input set, i.e., some inputs might not reach the program point. The symbolic execution process leverages ϕ-functions to propagate values at join points after branches and it also performs expression simplification based on constant propagation [34]. Additional simplification removes infeasible branches and expressions that are not related to the verification point. At the end of this process, the verification conditions are a set of equations describing the program, formatted as guard statements. Each guard statement is defined as a control flow instruction depending on expressions defined over the input values. To control the flow of the program, statements are combined into logical expressions that are concatenated with variable definitions.

4.2 Search Process

The search process aims to optimise the parametrisation according to the parameter restrictions detailed in Section 3.2 and to the fitness functions described in Equation 1. First, we create an initial population of individuals that are potential solutions to the parametrisation. Each individual is represented as a vector of real numbers with the values between 0 and 1. Figure 3 shows the individual's encoding. The first value in this vector (denoted as % DoF) decides on the degrees of freedom or number of expressions from the expression variation parameters to be added to the solver as constraints.

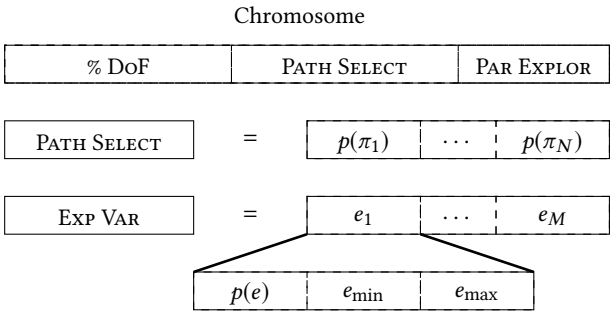

Fig. 3. Chromosome represented as a vector of real numbers, divided into three sections: percentage of degrees of freedom, path selection parameters, and partition exploration parameters. The degrees of freedom vary between 0 (none) to 1 (max). Path selection parameters give the probability for each of the N possible paths, π_i, to be chosen. The expression variation parameters (Partition Exploration) are divided into: expression probability e_p, and minimum and maximum values for the parameter ($e_{\min}$ and $e_{\max}$).

This value is computed as the product of (DoF) and the number of inputs. The path selection parameters (Path Select) are a set of normalised probabilities for each path. The expression variation parameters (Exp Var) correspond to the new expressions created to control the uniformity of the generator. Each Exp Var is a tuple ($p(e)$,$e_{\min}$,$e_{\max}$), described in Section 3.1. Once the degrees of freedom are set, the variation expressions are chosen according to their normalized probability $p(e)$. Then, the e_{val} value is selected uniformly at random from the range of $e_{\min}$ and $e_{\max}$.

We investigate three different search strategies to find solutions to the bi-objective parametrisation problem: random parametrisation, greedy search, and multi-objective evolutionary optimisation.

Random parametrisation chooses the parameters uniformly at random and runs the generator. The other two strategies are multi-objective, i.e., they use the two objectives described in Section 3.2 to produce the optimal Pareto front. A *Pareto front* is the (ideal) output of multi-objective optimisation and it can be defined as a set of non-dominated solutions. A solution x is considered dominated by another solution x^* if, and only if, x^* is equally good or better than x with respect to all objectives.

Greedy search is performed in iterations. At each iteration, the algorithm randomly creates a set of solutions, and keeps an archive of solutions that are non-dominated. This archive gets combined with new solutions forming a new archive of non-dominated solutions. Once the maximum number of iterations is reached, the algorithm stops and the final archive becomes the output.

The multi-objective evolutionary optimisation employed uses Strength Pareto Evolutionary Algorithm 2 (SPEA2) [61]. The algorithm starts by creating a population of individuals, uniformly at random, and an empty archive of fixed size. At each generation, the non-dominated solutions of the previous generation's population and archive are included into the archive of the current generation. If the number of individuals in the new archive is bigger than the predefined archive size, the archive gets truncated. Otherwise, the remaining space is filled with dominated solutions. The algorithm uses tournament selection with replacement to create a mating pool. Then, it applies two-point crossover and mutation on the chosen chromosomes to create a new population. The mutation simply selects an allele from the chromosome, according to the mutation probability, and reassigns its value to a value selected uniformly at random from within its range. The process terminates when the maximum number of generations is reached and the Pareto front, composed of the non-dominated solutions of the archive, are provided as output.

For each individual produced by the above approaches, we compute its fitness values using Equation 1, by creating a test suite based on the parameters described by the individual and the generator. We set a total number of inputs that the generator can create and keep track of the number of satisfiable constraints produced during the generation process. This sets the parameters $TotSat$ and $TotCalls$. Then, we calculate the collisions on the test suite to compute s and we set the threshold t based on the test suite size (see Algorithm 2). During the fitness computation, to guarantee that we do not force fictitious uniformity, we allow the generator to produce repetitions. We calculate the second objective using t and s, eventually minimising collisions (Section 3.2). Although there can be several solutions on the Pareto front, our primary objective when selecting the final ones is diversity, which is used as the final test suite selection criterion at the end of the generation process.

In our experiments (see Section 6), we set the population and archive size to 300, the tournament size to 7, and the number of generations to 50. The crossover rate and mutation rate are set to 0.5 and 0.1, respectively. Grid search and random parametrisation are configured to generate the same number of solutions as SPEA2, to allow for fair comparisons. These parameters were chosen, using grid selection, as those providing a good trade off between time, exploration and exploitation.

5 RESEARCH GOALS

DFT generates a set of program inputs that satisfy two properties: uniformity of the input distribution and reachability of the target program point. To evaluate the usefulness of DFT, we consider three key issues: does DFT fulfill its promise of uniformity and reachability? How useful is this focused diversity driven approach in fault detection? How efficient is the test suite generation algorithm? These issues can be elaborated into the following research questions:

RQ1*: What proportion of the inputs generated by DFT reach the target program point and what is their degree of diversity?* To measure diversity, we rely on the studies showing that a diverse set of inputs must maximise entropy [52]. According to Cover and Thomas [16] (Theorem 2.6.4), if p is a probability distribution and $H(p)$ is its entropy, $H(p)$ is maximum if and only if p is uniform. Therefore, a totally diverse test suite will follow a uniform distribution. Hence, we measure the distance between our test suite's distribution and a uniform distribution using the L2-test.

To measure reachability, we instrument every program point at the beginning of a branch and apply DFT to generate inputs passing through these points. As we want to measure different configurations of DFT parameters, we compare different parametrisation strategies with two baselines: a random input generator and the application of a basic solver directly using the constraints created by CBMC, with no search-based parameter tuning.

RQ2*: What is the quality improvement of the test suites generated by DFT with respect to the state-of-the-art, in terms of mutation killing capability and fault detection? What is the time required to produce them?*

Mutation testing introduces small syntactic alterations to the program (i.e., mutants) and measures the ability of a test suite to detect the errors caused by these alterations, i.e. the ability to kill the mutants. Along with the mutations, in our experiments we use also hand-seeded faults and real bugs that have been detected during the development process. We aim to evaluate the ability of DFT, under different configurations, to detect all these types of faults, in comparison with three baselines, namely, a pseudo-random generator, symbolic execution and search-based testing. This comparison not only measures the effectiveness of the focused strategy but also analyses to what extent its combination with diversification affects it in terms of mutation score and fault detection.

RQ3*: Which search process is most successful in reducing the unsatisfiability effect potentially produced through the introduction of parametrised constraints? and what is the residual unsatisfiability rate?*

The process of adding constraints to an existing constraint might produce an unsatisfiability scenario. This problem, well known in universal hashing approaches [11], can be regarded as the *error rate* of the algorithm. It might impact the effectiveness of the generator negatively, as the solver may return an error at a higher frequency when the parametrisation is in place. To evaluate whether any search strategy alleviates this problem, we compare different parametrisation strategies when using DFT.

RQ4*: Is the quality of the Pareto front solutions higher when we use an evolutionary optimisation algorithm (SPEA2) than when we use a greedy search approach?*

The optimisation process aims to reduce the error of the algorithm while maintaining uniform inputs. However, the computational effort of the multi-objective process might not attain significant improvements in the search objectives over a simple, greedy search. To measure such improvements we use the Hypervolume (HV) indicator [22]. HV measures the area defined by the Pareto front using a specific point as a reference. In our case, since the two goals are between 0 and 1, where 1 represents the best result, we aim to maximise the HV whose reference point is the origin (0, 0), so the maximum possible volume will be 1. We chose this indicator as it is a complete indicator of diversity, coverage and spread of solutions in the Pareto front [35] and it is commonly used to assess multi-objective algorithms in software engineering [47, 48].

6 EXPERIMENT DESIGN

To evaluate our proposed approach, DFT, we used 105 software subjects (for which both the buggy and fixed versions are available) selected from three open-source software repositories. Then, to identify the program points on which we should focus the test generation process, we applied a tree edit distance algorithm to buggy and fixed versions of each program. Finally, we evaluated the performance of the generator, so as to answer the research questions discussed in Section 5. The following subsections provide the details about the design of our experiments.

6.1 Dataset

The subject programs for our experiments were drawn from different sources: a repository of real bugs that we have derived from the R-project, the Software-artifact Infrastructure Repository (SIR) [45], and Codeflaws (CF) [56].

From the R-project, we extracted reported bugs that are related to C functions used for mathematical computation by manually reviewing the subversion logs submitted between 2012 and 2017. During this analysis, we found eight bugs related to statistical functions using floating point arithmetic on probability distributions, all of which are reported in the R-Bugzilla repository[2]. Among these bugs, we had to discard 4 bugs as they present incompatibility issues with CBMC. We collected the function version before and after each fix, including all the dependencies required to compile it. On average, there are 25 dependent source files for each function, 14 of which are headers. The considered functions have an average size of 1,545 LoC and an average number of 25 branches. The specific bugs we dealt with are # 16972, 16521, 15620 and 15075, all available on the R-Bugzilla repository.

From the SIR repository, we used the `tcas` program, a program used to avoid collisions in aircraft systems. This repository contains another 14 C programs, but all of them deal with strings. Unfortunately, the current implementation of our tool is not able to deal with string types (see Section 8). The `tcas` program has 135 LoC and 41 seeded errors. This program has 40 branch statements.

[2] https://bugs.r-project.org

The CF repository includes 7,436 C programs in total, each of which has on average 50 LoC (for a total of 35,654 LoC). Among these, 3,902 programs have both a buggy and a fixed version, which was obtained from a programming competition called Codeforces[3]. Thus, we selected 100 pairs of buggy/fixed programs for our experiment, uniformly at random, under the constraint that they were numerical programs, discarding programs with pointers as inputs. These programs have an average of 10 branch statements.

The data we used herein, including the R-project defects and patches, is already publicly available and the tool is available in Github[4].

6.2 Tree Alignment

In our experiments the target program point is the one containing the fault (in the case of errors) or the mutation (in the case of mutants). If the fix of this fault includes adding new lines of code or deleting the existing ones, the identification of the target program point becomes a non trivial task. To address this, we perform a *tree alignment* of the faulty program and its fixed version. The alignment process transforms the programs into their abstract syntax trees and calculates the tree edit distance between them, using Zhang and Shasha's algorithm [60], similarly to Chawathe et al. [13]. This algorithm provides the shortest sequence of edit commands that converts one tree into the other, at the node granularity. Each node is labeled with the corresponding edit operation, which can be `transform`, `insert`, `delete` or `keep`. Using this information, we set the program point in the fixed version, after the modified node. In the presence of multiple modified lines we consider all of them as target program points, selecting the beginning of each area as the program point of interest. This ensures that all modified code is targeted by DFT.

```
1 int func(int x){          1 int func(int x){
2   x--;        //            2   x--;        //
      Keep                        Keep
3   x+=2;       //Del         3
4   //pp                      4   //pp
5   if(x>10)    //            5   if(x>10)    //
      Keep                        Keep
6     x=x/2;    }//           6     x=x/2;    }//
      Keep                        Keep
```

Fig. 4. Example of alignment.

Figure 4 shows an example of identification of the target program point where the fix of the fault requires deleting a line of code. The tree edit distance algorithm labels the node at Line 5 with a `delete` operation. Using this information, we set the program point, automatically, after the deleted node. If the modified node is the exit statement of a function (for example, a `return` statement), we set the target program point just before this node. An execution of such a program point guarantees the execution of the exit statement. When the modification is in a conditional statement, we create two program points, one inside the true branch and the other inside the false branch, as we want inputs activating both paths.

6.3 Evaluation Procedure

The evaluation was performed in three parts, and each of them focused on a different research question. The first part is the uniformity and reachability evaluation (Research Question 1). This part was performed using the instrumentation

[3] https://codeflaws.github.io and http://codeforces.com
[4] https://github.com/hdg7/DFT

process and the L2-test [18]. For this evaluation, we created as many tests as the L2-test requires to measure the distance from a uniform distribution, allowing repetition.

The second part compares the ability of DFT to detect mutants and real faults with respect to the other baseline techniques (Research Question 2), while the last part measures the efficiency of the search process in terms of errors incurred by the solver and optimisation performance (Research Questions 3 and 4). For these two parts of the evaluation (second and third part), we first applied the alignment algorithm to identify the portion of code that changes between the faulty (or mutated program) and the fixed one. Then, we used DFT to create a test suite of 500 tests for each program under test. We compare the inputs produced by our tool with those generated by CBMC, CAVM [33] (a search-based test generator for C) and a random generator. CBMC generates inputs by sampling an SMT solver with the generated constraints that are focused on the program point. This guarantees a fair comparison with the focused search of DFT. In the case of CBMC, once an input is included, we add the negation of this input to the set of constraints, to be able to generate a different one. CAVM applies a whole test suite generation process that aims to generate an input per program branch, maximising branch coverage.

We also included an extra experiment, related to Research Question 2, in which we provided the same time budget (20 minutes) for every technique and directly compared each implementation's performance and effectiveness given that time budget, rather than comparing the methodologies *per se*. This experiment aims to show how the tools work in practice although improvements in any tool's implementation could alter the comparison results. For this reason, in this research question we present the results for both a fixed size and a fixed time budget.

Every experiment was repeated 30 times per program to account for the non determinism of the algorithms involved. We used aggregate measures such as median, mean and standard deviation (SD) to summarise the resulting distributions.

7 EXPERIMENTAL RESULTS

We started our experiments by measuring uniformity and reachability of DFT using different configurations, and then comparing it with the baselines: a (pseudo-)random generator, CAVM[5] and a symbolic executor (using CBMC's constraints). We then measured the ratio of faults detected and the mutation score. Finally, we compared the efficiency of the algorithms in terms of error rate and optimisation quality.

7.1 Uniformity and Reachability

For the first experiment, we chose a program point at the beginning of each branch in every program of our corpus. For CodeFlaws and SIR we used the complete program, for the R-functions we used each specific function.

We measured the uniformity of the approach by using the L2-test in order to calculate how close DFT is to producing a uniform distribution. The L2-test includes a notion of distance, defined in terms of the epsilon parameter of the test. The value of epsilon affects the sample size (Section 4.2). For our experiments, we checked three epsilon distances: 0.1, 0.05, 0.01. These distances are smaller than the traditional distances from the state-of-the-art experiments, where it is normally around 0.25 [18], hence setting a stricter criterion for the L2-test. Based on the domains, we bound the generation of samples for the experiments to 6,000, 12,000 and 24,000, respectively, allowing repetition during the sampling process.

[5]During the experiments, CAVM went out-of-memory on 52% of CF programs (results with * in the tables are restricted to the programs where it worked). Restricting DFT to those programs where CAVM works shows almost the same results as the general ones. CAVM has type compatibility limitations, which prevents it from generating inputs for some R-functions. It fails with abstract numeric labels such as infinite or NaN.

Epsilon	0.1	0.05	0.01
Inputs upper-bound	6,000	12,000	24,000
R CAVM	-	-	-
R SymEx	▼0.00%	▼0.00%	▼0.00%
R DFT/Random	▼83.5%	▼83.5%	▼70.5%
R DFT/Greedy	87.5%	87.5%	87.5%
R DFT/SPEA2	**90.0**%	**87.5**%	**87.5**%
SIR CAVM	▼0.00%	▼0.00%	▼0.00%
SIR SymEx	▼0.00%	▼0.00%	▼0.00%
SIR DFT/Random	**100.0**%	**100.0**%	▼92.5%
SIR DFT/Greedy	**100.0**%	**100.0**%	▼95.0%
SIR DFT/SPEA2	**100.0**%	**100.0**%	**100.0**%
CF CAVM(*)	▼0.00%	▼0.00%	▼0.00%
CF SymEx	▼0.00%	▼0.00%	▼0.00%
CF DFT/Random	▼81.2%	▼75.3%	▼42.5%
CF DFT/Greedy	85.3%	85.0%	61.2%
CF DFT/SPEA2	**86.9**%	**87.2**%	**63.1**%

Table 1. L2-test/diversity of the different approaches: percentage of test generation runs that pass the test. The ▼ symbol highlights the results which are significantly worse than DFT/SPEA2. Bold highlights the best results.

Table 1 shows the results of the L2-test for the three different values of epsilon on the three repositories. Percentages show the proportion of test suites generated for each program point that have passed the L2-test. The random generator of inputs passes the test in all cases, as it samples directly from the uniform distribution. Hence, it is not shown in Table 1 (its value being always 100%, by construction). On the other hand, CAVM and the symbolic execution generator (which uses directly the constraints of CBMC), always fail the L2-test. DFT can produce good diversification in terms of uniformity even with random parameters. However, its generation productivity significantly improves when greedy search or SPEA2 parametrisation strategies are applied.

We checked whether there is a statistically significant difference between these distributions by using the Wilcoxon test [31], a non-parametric test that compares probability distributions. We use the SPEA2 search as our baseline for the test and compare it with the other techniques. When the p-value is smaller than 0.05, we consider that there is a significant difference. The results are denoted in Table 1, where the presence of the arrow points out the cases when the difference is statistically significant. The color/direction of the arrow indicates when results are significantly worse (red downward arrow). As we can see, DFT achieves significant improvements in almost every repository w.r.t. the search-based and symbolic execution approaches. We have noticed that the generated test suites tend not to pass the L2-test in small domains, e.g., when the domain is limited to tens of possible inputs. However, in those cases we may get all possible inputs exhaustively.

To measure reachability, we read the traces produced by the instrumentation and verified that the flag of the program point is active for the given test case. For each test suite, we calculated the percentage of tests that reach their target program points and show their descriptive statistics in the first three columns of Table 2 (median, mean and standard deviation). The table shows that the pseudo-random generator and CAVM have the worst reachability results (lower than 50% and 70% in median, respectively). We can notice that symbolic execution is also not able to reach all target program points. Our analysis of the traces indicates that this is a consequence of the unwinding process. In fact, loops

Method	Reachability			Unsat	HV
	Median	Mean	SD		
R Random	▼18.9%	18.5%	± 9.1	-	-
R CAVM	-	-	-	-	-
R SymEx	▼80.8%	69.0%	± 38.3	-	-
R DFT/Random	88.5%	88.0%	± 10.3	▼54.8%	-
R DFT/Greedy	**98.1**%	**90.7**%	± 16.1	▼25.7%	▼0.63
R DFT/SPEA2	93.8%	90.6%	± 12.0	**1.8**%	**0.85**
SIR Random	▼00.0%	14.3%	± 36.3	-	-
SIR CAVM	▼58.4 %	57.3%	± 32.2	-	-
SIR SymEx	**98.7**%	78.3%	± 33.5	-	-
SIR DFT/Random	90.0%	81.8%	± 27.4	▼88.9%	-
SIR DFT/Greedy	96.6%	**82.7**%	± 31.3	▼9.6%	**1.00**
SIR DFT/SPEA2	93.0%	81.2%	± 24.6	**3.1**%	**1.00**
CF Random	▼49.8%	54.2%	± 35.0	-	-
CF CAVM(*)	▼65.6%	58.4%	± 37.1	-	-
CF SymEx	**100**%	73.3%	± 43.6	-	-
CF DFT/Random	**100**%	99.8%	± 2.3	▼75.8%	-
CF DFT/Greedy	**100**%	**99.9**%	± 1.6	▼6.9%	▼0.64
CF DFT/SPEA2	**100**%	**99.9**%	± 1.6	**2.3**%	**1.00**

Table 2. Reachability, percentage of unsatisfiable calls (only for DFT variants) and HV (only for DFT with search guidance). The ▼ highlights the results which are significantly worse than DFT/SPEA2. Bold highlights the best results.

Method	R Mut Sc	R Faults	SIR Mut Sc	SIR Faults	CF Mut Sc	CF Faults	Time (R / SIR / CF)	Memory (R / SIR / CF)
Random	▼55.17%	▼00.00%	▼00.00%	▼07.14%	▼72.86%	▼74.07%	29 s / 31 s / 35 s	42 M / 38 M / 32 M
CAVM	-	-	▼07.03%	▼15.33%	▼78.22%	▼77.72 %	- / 6 m / 28 m	- / 130 M / 128 G
SymEx	▼43.99%	▼25.00%	▼40.91%	▼35.71%	▼76.88%	81.48%	16 m / 2 m / 31 m	97 M / 61 M / 56 M
DFT/Random	**67.10**%	**50.00**%	**69.70**%	**46.43**%	79.36%	81.48%	17 m / 3 m / 33 m	112 M / 83 M / 67 M
DFT/Greedy	▼56.35%	**50.00**%	▼59.09%	**46.43**%	80.59%	**83.33**%	35 m / 7 m / 68 m	121 M / 97 M / 80 M
DFT/SPEA2	66.03%	**50.00**%	**69.70**%	**46.43**%	**80.90**%	**83.33**%	33 m / 6 m / 63 m	122 M / 99 M / 81 M

Table 3. Mutation score and faults detected for the three repositories: Codeflaws, SIR and the R-functions. The ▼ symbol highlights the results which are significantly worse than DFT/SPEA2. Bold highlights the best results.

preceding the program points of interest might change the behaviour of the program depending on the unwinding or might produce a timeout as they do not terminate in some cases. As the solutions of the solver tend to be close to each other, this effect has large impact when it is not mitigated by the search guidance. The diversity-based techniques implemented in DFT show better results, especially on CodeFlaws and on the R-functions. For all DFT approaches, reachability is always higher than 80%.

RQ1: *DFT produces suites close to uniformly distributed and it reaches the target program point 90% of the time on average, improving the results for the baselines.*

7.2 Mutation Score and Faults Detected

To measure the effectiveness of DFT, we evaluate it in terms of mutation score and number of faults detected, and compare it with the baseline techniques. We created up to 100 mutants per program (or function, for the R-functions),

Method	R Mut Sc	R Faults	SIR Mut Sc	SIR Faults	CF Mut Sc	CF Faults
Random	▼60.22%	▼00.00%	▼00.00%	▼09.56%	▼74.01%	▼75.12%
CAVM	-	-	▼12.22%	▼16.07%	▼75.11%	▼72.03 %
SymEx	▼45.20%	▼25.00%	▼45.73%	▼42.40%	▼68.21%	▼75.33%
DFT/Random	▲**68.55%**	**50.00%**	**75.23%**	**55.20%**	77.19%	79.83%
DFT/Greedy	▼55.11%	**50.00%**	**75.23%**	**55.20%**	76.95%	79.68%
DFT/SPEA2	65.45%	**50.00%**	**75.23%**	**55.20%**	**77.25%**	**80.22%**

Table 4. Mutation score and faults detected for the three repositories, Codeflaws, SIR and the R-functions, in a fixed time budget (20 minutes). The ▼ symbol highlights the results that are significantly worse than DFT/SPEA2, while the ▲ symbol highlights the results that are significantly better. Bold highlights the best results.

using Milu [32]. For each of the mutants, we ran the same test suite on the mutant and on the original program to check whether they produce different outputs. When the testing outputs are different, we consider that the test suite strongly kills the mutants. The mutation score is the percentage of mutants killed. In this experiment, we used the alignment algorithm described in Section 6.2 to create a test suite passing through the program point where the mutation is located. For every program point we created a test suite, with a size of 500 tests, per technique. We follow a similar approach to detect the real faults, using the buggy version as we used the mutation.

Table 3 shows the median mutation score and percentage of faults detected for each technique and repository. We compare the pseudo-random generator, symbolic execution, CAVM, and the three configurations of DFT. For every repository, DFT performs significantly better (using again the Wilcoxon test, and considering a p-value threshold of 0.05) than the baselines in detecting faults and killing mutants. The improvement is strongly representative with respect to random testing and CAVM in the SIR and R-functions repositories, where the random baseline is not able to detect a single mutant in the former or a single bug in the latter, and CAVM fails due to compatibility issues. Although there are not statistically significant differences between the alternative strategies of DFT, the SPEA2 search shows better results in general. Considering the latter technique, our improvements are 3% for CF, 70% for SIR and 20% for the R-functions with respect to the best baseline on the mutation score. On real fault detection, improvements are 2% for CF, 30% for SIR and 100% for the R-functions.

We also evaluated how these results change depending on the size of the test suite. Figure 5 shows that on CodeFlaws data there is a large difference between the asymptotic behaviour of the DFT-based strategies and the asymptotic behaviour of symbolic execution and random approach. The figure shows the median mutation score computed over 30 executions of each test suite with suite sizes ranging from 10 to 250. Starting from size 10, we can notice a significant gap between DFT and random/symbolic execution, while the gap with CAVM is smaller. This gap remains consistently stable across the whole size range.

The execution time of DFT is similar between Greedy and SPEA2 searches. Both take approximately twice the time of DFT/Random. The memory consumption is similar across the DFT variants (slightly better for DFT/Random). CAVM exceeded the memory limits in several occasions, resulting in out-of-memory errors.

Due to the fact that the execution time is quite different across the compared approaches, we designed an experiment where each of them has been allocated the same budget (20 minutes) to generate inputs. In the case of Greedy and SPEA2 versions of DFT these 20 minutes were divided into 30% for search and 70% for generation. Table 4 shows the obtained results. During these experiments, Random generates around 300,000 inputs while the other tools generate a similar amount of inputs to each other, usually in the range from 200 to 500, depending on the program. We can see

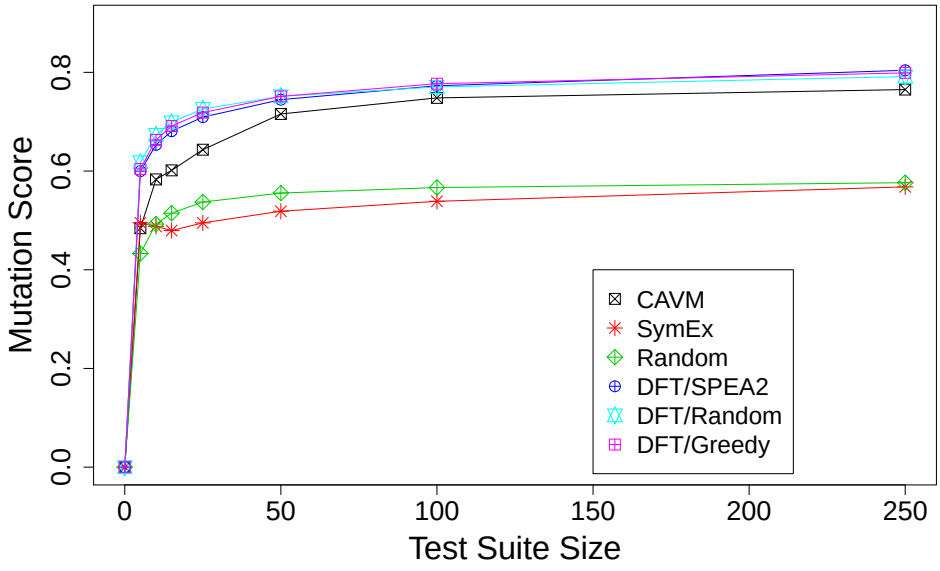

Fig. 5. Mutation score vs. test suite size for the random baseline (Random), symbolic execution (SymEx), CAVM and DFT using random parameters (DFT/Random), greedy search (DFT/Greedy), and SPEA2 (DFT/SPEA2).

that the asymptotic behaviour of the different generators shown in Figure 5 reaches a plateau and remains stable as the test suite size grows. These results confirm that DFT outperforms the rest of the tools, achieving better results even on the SIR repository.

RQ2: *DFT performs better than the baseline approaches, reaching between 3% and 70% improvement in mutation score and up to 100% improvement in detecting real faults. Execution time and memory consumption are acceptable (order of 30 m, 100 M, respectively).*

7.3 Unsatisfiability of the Amended Constraints

DFT requires adding new constraints to those extracted from CBMC. The main problem with these amended constraints is the potential loss of satisfiability [11]. We measured our error rate using the first experiment's data, counting, for each program point, program and repository, the median number of times the solver returns an error. Table 2 shows these results under Column *Unsat*. We can see that both random parametrisation and greedy search are significantly worse than SPEA2 on every repository. This means that the effort by SPEA2 to obtain the same number of solutions is lower, since it incurs a substantially lower number of constraint solving errors. So, with a given solver, it will take less time to generate the same number of witnesses. We can see that the SPEA2 method is able to reduce the error rate from 89% to 3% in the best case and from 54.8% to 2% in the worst case.

RQ3: *The comparison of the different search-based parametrisation strategies shows that SPEA2 search significantly reduces the error rate incurred due to the additional constraints, keeping it as low as around just 3% on average.*

7.4 Multi-objective vs. Greedy Strategies

To compare the multi-objective search strategies and to understand the quality of the resulting Pareto front, we compute the HV of the generated fronts. Table 2 shows the median HV for every program point and program of the first experiment. We can observe that the HV of the Pareto front generated by SPEA2 is always higher than the one generated by the greedy approach. As Table 1 shows, this has not a dramatic impact on the uniformity of the solution. However, the impact is significant on the error rate of DFT (see Table 2, Column *Unsat*). Even if this affects the generation productivity, it does not affect the quality of the results either in terms of killing mutants or detecting real faults (see Table 3).

RQ4: *The quality of the Pareto front solutions is higher when adopting a multi-objective genetic search process. While this has no dramatic consequence on the final results of DFT (uniformity, reachability, fault detection), it improves the efficiency of the algorithm, by substantially reducing the solver's error rate.*

8 EXTENSIONS AND LIMITATIONS

DFT is a methodology for the testing of specific sections of a program by manipulating the associated program constraints. The current implementation is limited by the technicalities of the SMT solver, however, it can be extended to deal with more types. DFT can be extended to handle branch condition expressions that contain calls to helper functions on complex data types, but with primitive return types, as follows:

Numerical arrays. Different properties of arrays are often used in branch condition expressions to check whether a certain operation can be performed on an array. For instance, the length of an array might be checked before adding a new element to it. This can be achieved with the use of helper functions, that provide some meta-information on the array (i.e. its length, sum of elements and mean, among others). DFT can be extended so that it handles such helper functions as part of its constraints. Programs with branch conditions involving array access via a symbolic index would require the adoption of an SMT solver that supports a theory of arrays.

Strings. Strings work similarly to numerical arrays but have an embedded semantics. There are several examples of helper functions that determine whether a string is lower or upper case, whether it is empty or it contains a specific substring. There are also operations, such as Levenshtein or redex distances, used to compare strings. DFT can manipulate the constraints related to these comparisons but, as for the numerical arrays, an SMT solver supporting a theory of strings (and possibly of regular expressions) is required to find solutions to constraints that involve the content of a string.

Static Structures. Static structures consist of subcomponents that might have primitive types such as integer or floating point. These primitive types are already considered in the current implementation of DFT. Therefore, DFT can be easily extended to deal with the branch comparison expressions using the primitive type subcomponents of static structures.

Pointer Arithmetic. Arithmetic conditions on pointers involve the comparison of different memory addresses as integer values, by calculating distances between them, for instance, to calculate the size of a buffer or the existence of aliasing between pointers. These comparisons are similar to integer comparisons, which are already part of DFT, so the extension required to handle pointer arithmetic may be minimal. However, this is no longer true if we want to deal also with the dereference content.

Pointer Dereference Content. The content of the pointer dereferences are complex for SMT solvers to handle. To make their representation more in line with the requirements of DFT, we need to build a memory model in form of an abstract heap [53] containing a NULL pointer, relative memory positions in a symbolic address space, and values in this memory model at the specific places to which a pointer may point. Once this extension is provided, if the memory

content of a pointer dereference is an atomic value, it would behave as primitive type, a numerical array or a string. If the content is another pointer, it would behave according to pointer arithmetic. In both cases, DFT would require the adoption of an SMT solver supporting symbolic pointers.

Dynamic Structures. Extending the idea of memory models, dynamic structures such as trees, linked lists and queues also contain operations that deal with comparisons, for instance: number of elements, depth of a tree, etc. DFT could modify these operations into utility functions with primitive return types, to generate specific instances that reach specific program points related to them. However, to handle the general case in which the pointers used to create such structures are also symbolic, the tool would require a symbolic memory model and an SMT solver supporting symbolic pointers.

Another potential extension of the tool that could improve its execution time, by reducing the amount of constraints it needs to deal with, is concolic execution [50]. The combination of concolic execution with DFT would work as follows: a concrete execution would provide the specifics of a path and the symbolic constraints that lead to the program point along a specific execution path. Then, DFT would modify these constraints to find another input that gets as close as possible to the target, but along a different path, and so on, until it gets a necessarily under-approximated set of different path constraints to reach the same program point. This would improve the current scalability of our tool, but at the same time it would reduce the number of program paths taken into account to reach the target program point.

Extending DFT to object-oriented languages would not be difficult for testing specific methods if they work with primitive values or static structures. However, if they require objects (i.e., dynamic structures) as inputs, DFT would need to also control the order of method calls performed to construct and change the state of these objects. The symbolic constraints would work in a similar way to the dynamic structure extension.

8.1 Threats to Validity

Construct Threats. A potential threat to construct validity can be our definition of diversity. We considered uniformity as a measure of diversity, due to its justification via Information Theory. However, other authors use similarity metrics between the test cases. Use of similarity metrics might have led to different results.

Internal Threats. The use of CBMC is a threat to internal validity because of the bounded loop unwinding process. We are aware that there are other techniques developed to cover code targets, especially in the area of malware triggering [58]. However, these techniques either aim to find just a single test case that covers the target [58] or cannot be adapted to a non-binary parametrisation [2]. It is important to remark that our methodology does not need to discard tests to improve diversity. Another threat to the internal validity of our study is the chosen SMT solver. We used Z3, a mature and robust SMT solver often adopted in software testing research. However, a different solver might exhibit different behaviours.

External Threats. Our experiments are performed on open source code repositories. Although our subjects have been previously used in the literature [14, 54, 59], our results might not generalise to other subjects and/or programming languages. The use of solvers imposes a potential limitation to the software size our approach can be applied to, as solvers can not handle programs that have several branches, a large number of lines of code or extremely complex constraints. Nevertheless, the solver technology is under continuous development and improvement.

9 RELATED WORK

The DFT approach touches on various aspects of testing, including automatic test suite generation, focused testing, GödelTest and diversity of test suites, thus we discuss related work within these fields.

Automatic and Focused Test Generation The field of automatic test case generation has grown in several directions such as symbolic execution, combinatorial testing, model-based testing, search-based testing and adaptive random testing [5]. The main goal of these techniques is to achieve a high coverage according to specific coverage criteria.

Shamshiri et al. [51] evaluate three automated unit test case generators (EvoSuite [23, 24], Randoop [41] and Agitar [1]) in terms of effectiveness in real fault detection. The results show that none of the tools could achieve more than 40.1% of fault detection individually, while the overall fault detection of the tools together is only 55.7%. The qualitative analysis conducted by the authors identifies the presence of *failed error propagation* (fault execution and infection with no propagation) and *coincidental correctness* (fault execution with no infection and no propagation) as one of the reasons for the low fault detection rate. They report cases when the fault was always fully covered but never detected, as it requires specific values in order to be detected.

Evidence of high failed error propagation and coincidental correctness rates was provided in previous work [6, 37–39, 57]. To address this, one needs to test those sections of the program where the error fails to propagate. This is possible by focusing test inputs at specific elements of the program. Alipour et al. [2] and Gotlieb and Petit [29] use search algorithms and solvers, respectively, to reach faulty program points. As reaching the program point is not enough to trigger the faulty behavior, we also included diversity in our input generation algorithm.

Diversity, Uniformity and GödelTesting Diversity shows significant improvements on fault detection when combined with other testing criteria [14, 26]. These improvements were originally reported in the diversified version of Random Testing, named Adaptive Random Testing, where Chen et al. [14] spread the inputs through the space using uniform sampling instead of pseudo-random generators. Chakraborty et al. [11] stated that the effectiveness of diversity comes from the fact that every test is equally likely to activate or detect a bug. In the literature there are several works aiming to diversify test suites [9–11, 14, 21, 25, 43, 52], or use diversification as a filter [20].

The initial work of Chakraborty et al. aimed at creating uniform inputs for testing electrical circuits. Their diversification tool, UniGen, transforms the circuits into a set of constraints in Conjunctive Normal Form (CNF) and applies SAT solvers to transform the witnesses of these constraints into inputs [9, 11]. Diversity methods have focused also on incrementing the diversity of outputs, as in the work by Alshahwan and Harman [3, 4], which introduces a novel test suite adequacy criterion based on output uniqueness. The work by Matinnejad et al. improves diversity on Simulink models by reducing the similarities among their outputs [40]. None of the aforementioned approaches combines diversity with focused test input generation.

Existing work defines diversity in different ways. Considering an information theoretic perspective, some work connects it with the normalised information distance (NID) [21], while others connect it with the entropy of the test suite [52]. NID can be used to define the diameter of the test suite, i.e., the average distance between the test cases, with the aim of maximizing it [20]. Entropy leverages its connection with the uniform distribution, since a distribution is uniform if and only if its entropy is maximum [46]. This connection gives us a good approximation to a proper definition of diversity, where we consider a test suite diverse when it is uniform or has maximum entropy. However, measuring the uniformity of a test suite is a difficult task, as it requires statistical tests. This was one of the main problems that has been identified in the evaluation of diversity in previous work [9–11, 28]. Our approach addresses this problem by using a collision-based L2-test to guide and evaluate the generators (Section 3.2). This test was chosen amongst different possible tests for uniformity, e.g., continuous distribution tests [36] (not suitable as they assume no gaps between inputs) or discrete tests [12, 15, 18, 27, 55]. Only collision-based tests guarantee an appropriate evaluation in our scenario, as they give us also a notion of distance from uniformity [18] (Section 3.2). Although collision-based

tests require a big sample size for small epsilon values [18], our generators were able to satisfy this. Poulding and Feldt [43] aimed to create diversification by construction, applying a technique named GödelTesting. However, they require the creation of a manual test generator per program, so as to (manually) define the inputs to generate, based on program-specific constraints. In contrast, our technique leverages a constraint generator to obtain a meta-generator, and uses a solver as a general purpose generator, by including parameters in its constraints.

10 CONCLUSIONS AND FUTURE WORK

Diversified focused testing improves the quality of fault detection and mutation killing. By adding parametrisation to program constraints, our approach improves the diversity of the test suite, making the input distribution close to a uniform distribution. Regardless of the search strategy used to tune the parameters, empirical results show that the quality of the test suites generated by DFT significantly outperforms random generation, constraint solving and search-based generation. Moreover, in addition to making the test suites diverse, the search algorithm that tunes the parameters effectively reduces the error rate of the solver by orders of magnitude, hence improving the efficiency of the generation process quite substantially.

There are several future applications for DFT. A significant application is to study the effect of Failed Error Propagation at specific program points, extending the work of Androutsopoulos et al. [6]. Our technique can measure the reference probability of killing a mutant by ensuring that the generated inputs always traverse the mutation point and at the same time follow a uniform distribution. It could also be used to estimate the total number of inputs that traverse a given point [10]. Finally, from a security perspective, DFT may help developers identify potential triggers of suspicious code in malware analysis.

ACKNOWLEDGMENTS

This research was funded by the EPSRC InfoTestSS grant EP/P005888/1, the EPSRC SeMaMatch grant EP/K032623/1, and the ERC Advanced fellowship EPIC grant no. 741278. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research.

REFERENCES

[1] [n.d.]. Agitar, [Online]. Available: https://www.agitar.com, [Accessed: 06-Mar-2018].

[2] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating Focused Random Tests Using Directed Swarm Testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/2931037.2931056

[3] Nadia Alshahwan and Mark Harman. 2012. Augmenting Test Suites Effectiveness by Increasing Output Diversity. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1345–1348. http://dl.acm.org/citation.cfm?id=2337223.2337414

[4] Nadia Alshahwan and Mark Harman. 2014. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 181–192.

[5] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[6] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. 2014. An Analysis of the Relationship Between Conditional Entropy and Failed Error Propagation in Software Testing. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 573–583. https://doi.org/10.1145/2568225.2568314

[7] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. 2009. Automatic test generation for coverage analysis using CBMC. In *International Conference on Computer Aided Systems Theory*. Springer, 287–294.

[8] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.

[9] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation.. In *TACAS*. 304–319.

[10] Supratik Chakraborty, Kuldeep S Meel, Rakesh Mistry, and Moshe Y Vardi. 2016. Approximate Probabilistic Inference via Word-Level Counting.. In *AAAI*, Vol. 16. 3218–3224.

[11] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2013. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 200–216.

[12] Siu-On Chan, Ilias Diakonikolas, Gregory Valiant, and Paul Valiant. 2014. Optimal algorithms for testing closeness of discrete distributions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1193–1203.

[13] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. In *Acm Sigmod Record*, Vol. 25. ACM, 493–504.

[14] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[15] Vartan Choulakian, Richard A Lockhart, and Michael A Stephens. 1994. Cramér-von Mises statistics for discrete distributions. *Canadian Journal of Statistics* 22, 1 (1994), 125–137.

[16] Thomas M Cover and Joy A Thomas. 2012. *Elements of information theory*. John Wiley & Sons.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[18] Ilias Diakonikolas, Themis Gouleakis, John Peebles, and Eric Price. 2016. Collision-based testers are optimal for uniformity and closeness. *arXiv preprint arXiv:1611.03579* (2016).

[19] R. Feldt and S. Poulding. 2013. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 350–359. https://doi.org/10.1109/ISSRE.2013.6698888

[20] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 223–233.

[21] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. 2008. Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*. 178–186. https://doi.org/10.1109/ICSTW.2008.36

[22] Carlos M Fonseca, Luís Paquete, and Manuel López-Ibánez. 2006. An improved dimension-sweep algorithm for the hypervolume indicator. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 1157–1163.

[23] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In 11^{th} *International Conference on Quality Software (QSIC)*, Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo (Eds.). IEEE Computer Society, Madrid, Spain, 31–40.

[24] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419.

[25] G. Fraser and A. Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (Feb 2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[26] Gregory Gay. 2017. The fitness function for the job: search-based generation of test suites that detect real faults. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 345–355.

[27] Oded Goldreich and Dana Ron. 2011. On testing expansion in bounded-degree graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer, 68–75.

[28] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2007. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Advances In Neural Information Processing Systems*. 481–488.

[29] Arnaud Gotlieb and Matthieu Petit. 2010. A uniform random test data generator for path testing. *Journal of Systems and Software* 83, 12 (2010), 2618–2626.

[30] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[31] Myles Hollander and Douglas A. Wolfe. 1999. *Nonparametric Statistical Methods*. John Wiley & Sons.

[32] Y. Jia and M. Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*. 94–98. https://doi.org/10.1109/TAIC-PART.2008.18

[33] Junhwi Kim, Byeonghyeon You, Minhyuk Kwon, Phil McMinn, and Shin Yoo. 2017. Evaluating CAVM: A New Search-Based Test Data Generation Tool for C. In *International Symposium on Search-Based Software Engineering (SSBSE 2017)*. 143–149.

[34] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[35] Miqing Li, Tao Chen, and Xin Yao. 2018. A critical review of: a practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering: essay on quality indicator selection for SBSE. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 17–20.

[36] Y Marhuenda, D Morales, and MC Pardo. 2005. A comparison of uniformity tests. *Statistics* 39, 4 (2005), 315–327.

[37] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. 2009. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 1–5.

[38] Wes Masri and Rawad Abou Assi. 2010. Cleansing test suites from coincidental correctness to enhance fault-localization. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 165–174.

[39] Wes Masri and Rawad Abou Assi. 2014. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Trans. Softw. Eng. Methodol.* 23, 1 (2014), 8:1–8:28.

[40] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *Proceedings of the 38th international conference on software engineering*. ACM, 595–606.

[41] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[42] Robin L Plackett. 1983. Karl Pearson and the chi-squared test. *International Statistical Review/Revue Internationale de Statistique* (1983), 59–72.

[43] Simon Poulding and Robert Feldt. 2014. Generating structured test data with specific properties using Nested Monte-Carlo Search. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1279–1286.

[44] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[45] Gregg Rothermel, Sebastian Elbaum, Alex Kinneer, and Hyunsook Do. 2006. Software-artifact infrastructure repository. *UR L http://sir. unl. edu/portal* (2006).

[46] Reuven Y Rubinstein and Dirk P Kroese. 2016. *Simulation and the Monte Carlo method*. Vol. 10. John Wiley & Sons.

[47] F. Sarro, F. Ferrucci, M. Harman, A. Manna, and J. Ren. 2017. Adaptive Multi-Objective Evolutionary Algorithms for Overtime Planning in Software Projects. *IEEE Transactions on Software Engineering* 43, 10 (Oct. 2017), 898–917. https://doi.org/10.1109/TSE.2017.2650914

[48] Federica Sarro, Alessio Petrozziello, and Mark Harman. 2016. Multi-objective Software Effort Estimation. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2884781.2884830

[49] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 513–528.

[50] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 571–572.

[51] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[52] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu. 2016. Measuring the Diversity of a Test Set With Distance Entropy. *IEEE Transactions on Reliability* 65, 1 (March 2016), 19–27. https://doi.org/10.1109/TR.2015.2434953

[53] Carsten Sinz, Stephan Falke, and Florian Merz. 2010. A precise memory model for low-level bounded model checking. In *Proceedings of the 5th international conference on Systems software verification*. USENIX Association, 7–7.

[54] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 130–140.

[55] Mike Steele, Janet Chaseling, and Cameron Hurst. 2005. Simulated power of the discrete Cramér-von Mises goodness-of-fit tests. In *Proceedings of the MODSIM 05 International Congress on Modelling and Simulation. Advances and Applications for Management and Decision Making*. 1300–1304.

[56] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 180–182.

[57] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 45–55.

[58] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[59] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979.

[60] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.

[61] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-report* 103 (2001).