UNIVERSITY *of York*

This is a repository copy of *Selecting Highly Efficient Sets of Subdomains for Mutation Adequacy*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/81418/

Version: Submitted Version

Selecting Highly Efficient Sets of Subdomains for Mutation Adequacy

Matthew Patrick*, Rob Alexander*, Manuel Oriol*† and John A. Clark*

*University of York
Heslington, York
United Kingdom
{mtp, rda, manuel, jac}@cs.york.ac.uk

†Industrial Software Systems
ABB Corporate Research
Baden-Dättwil, Switzerland
manuel.oriol@ch.abb.com

Abstract—Test selection techniques are used to reduce the human effort involved in software testing. Most research focusses on selecting efficient sets of test cases according to various coverage criteria for directed testing. We introduce a new technique to select efficient sets of subdomains from which new test cases can be sampled at random to achieve a high mutation score. We first present a technique for evolving multiple subdomains, each of which target a different group of mutants. The evolved subdomains are shown to achieve an average 160% improvement in mutation score compared to random testing with six real world Java programs. We then present a technique for selecting sets of the evolved subdomains to reduce the human effort involved in evaluating sampled test cases without reducing their fault finding effectiveness. This technique significantly reduces the number of subdomains for four of the six programs with a negligible difference in mutation score.

I. INTRODUCTION

Subdomains are used to define the range from which test input values are sampled. It is important to identify efficient subdomains of input, so as to reduce the human oracle cost and provide a starting point for regression testing. Without detailed analysis, it is difficult to determine the best subdomains to use. For example, Andrews et al. [1] report that the subdomain (0..31) gave the best results in testing a dictionary, but it is not clear how they discovered this 'magic number'. The TriTyp program has three integer inputs (a, b and c) and its branches contain conditions such as $a=b=c$. Michael et al. [2] selected over 8000 test cases from the entire input domain, but exercised less than half of the branches. Duran [3] selected 25 test cases from the subdomains ((1..5), (1..5), (1..5)) and exercised all the branches. This paper introduces a technique for finding an efficient set of subdomains automatically.

We employ mutation adequacy as our selection criterion because mutants have been shown to be representative of real faults in software [4]. Random testing can be applied to generate test cases from the evolved subdomains with minimal computational expense [3]. Experienced practitioners perceive random testing as ineffective because of its inability to handle boundary conditions [5]. Yet with carefully selected subdomains, it is possible to target mutants more efficiently. Test cases sampled from within the subdomains are able to find faults more quickly. As a result, subdomain testing requires fewer test cases than is typical for random testing, thus reducing the human effort required to create test oracles.

Previously, we investigated evolving a single subdomain of test input [6] and multiple subdomains targeted at killing specific sets of mutants [7]. Both techniques perform better than generating test cases at random from within an arbitrary interval (0..100). Sampling test cases from multiple subdomains achieved on average 33% higher mutation score than the single subdomain approach, but required a large number of subdomains for some programs [7]. We found that reducing the number of test cases we sampled from a single subdomain increased test suite efficiency without greatly reducing performance [6]. In this paper, we will explore whether reducing the number of subdomains has a similar effect.

The key to selecting an efficient set of subdomains is to find those that complement each other in killing mutants. Our optimisation process initially trains subdomains against the complete set of mutants. This means that the first subdomains to be identified are reasonably good at killing a large number of mutants. Later in the optimisation process, subdomains are trained against mutants for which no effective subdomain has yet been found. These subdomains are evolved to become highly efficient at killing a smaller number of mutants. An efficient set of subdomains should provide a balance between killing a large number of mutants with a single subdomain and having the specificity required for difficult to kill mutants.

In this paper, we show that it is possible to maintain a high mutation score using fewer test cases, sampled from a smaller more efficient set of subdomains. This is achieved by means of a new sequential technique for subdomain reduction. Subdomains are added and removed one at a time until the highest possible mutation score is achieved for each set size (from a single subdomain up to the complete set). In this way, it is possible to identify the smallest set of subdomains that have a similar fault finding capability to the complete set. From the point of view of a human tester, it requires less time to evaluate test cases sampled from a smaller set of subdomains.

The rest of this paper is organised as follows. Section II gives background information relevant to subdomain optimisation and Section III describes the process we use to optimise subdomains. Section IV explains the methodology we use to select sets of subdomains. Section V details our experiments and Section VI presents the results. Section VII surveys the related work. Section VIII summarises our conclusions and Section IX makes suggestions for further work.

II. BACKGROUND

We use mutation analysis to evaluate subdomain effectiveness. An Evolution Strategy (CMA-ES) is employed to identify sets of subdomains capable of finding artificial faults.

A. Mutation Analysis

Mutation analysis uses artificial faults to make quantifiable predictions of the proportion of real faults that will be found by a test suite [8]. Predictions based upon real faults may indicate effective test data or poorly written software. Mutation analysis has been shown to be more stringent than other testing criteria and a good predictor of real fault finding capability [9][4].

We artificially introduce small changes in syntax one at a time into the program code (see Figure 1). When test data has been found that causes a mutant to behave differently to the original program, we say the mutant has been killed. Some mutants cannot be killed because they are semantically equivalent to the original program. The proportion of non-equivalent mutants killed by a test suite is known as its mutation score. A given testing approach can be considered effective if it is able to achieve a high mutation score.
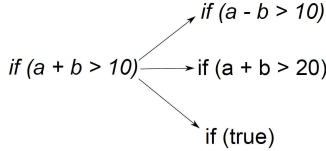

Fig. 1. Examples of three syntactic mutations

B. Evolution Strategies

Evolution strategies are optimisation algorithms inspired by the process of adaptation in nature [10]. In contrast to some genetic algorithms, evolution strategies emphasise mutation over recombination [10]. New candidate solutions are produced by applying a (typically Gaussian) update function (F) to existing sets of values, $x'_1 \ldots x'_n = F(x_1 \ldots x_n)$. Evolution strategies are suited to fine tuning numerical properties, as disruption from crossover is largely avoided. Amongst many other applications, they have been used to optimise image compression [11], network design [12] and web crawling [13].

C. CMA-ES

The evolution strategy applied in this paper (CMA-ES) uses Covariance Matrix Adaptation. CMA-ES represents the search neighbourhood with a multivariate normal distribution [14]. It uses a mean vector for the currently favoured solution, a scaling factor for the step size and a covariance matrix for the shape. Adaptation is performed to achieve fast, but not premature convergence, taking into account pairwise dependencies (covariance) as well as fitness in both time and space [14].

CMA-ES are popular because they can solve difficult optimisation problems without the need for manual parameter tuning. CMA-ES have been shown to be particularly effective at non-linear optimisation. In a recent black-box comparison study with 25 benchmark functions, CMA-ES outperformed eleven other algorithms in the number of function evaluations before the global optimum value is reached [15].

III. SUBDOMAIN OPTIMISATION

Subdomains specify how test data is to be sampled for input parameters to the program under test. We define a candidate solution as a set of subdomains in the following three forms:

Numerical subdomains
> are represented with a lower and upper value. Test inputs are selected inclusively between these values.

Boolean probability values
> are described with an integer value between 0 and 100. This value represents the percentage probability that a generated parameter value is 'true'.

Character array distributions
> are fixed in length (by default to five characters). Each special character (wildcard, closure etc.) is given its own independent probability of inclusion.

Algorithm 1 outlines the main process used in searching for subdomains. Subdomains are preferred that consistently kill the same group of mutants. This is achieved by maximising variance in the number of times each mutant is killed and minimising variance in the number of times the same mutant is killed (see Equation 1). A mutant is 'covered' if it is killed at least 95 times out of 100 by 5 test cases sampled from the subdomains, we refer to this as $TimesKilled(m)$ in Algorithm 1. Once subdomains are found to cover a group of mutants, the search continues with the remaining mutants.

$$
\sum_{s \in S} \sum_{m \in M} \frac{(K_{s,m} - \bar{K}_m)^2}{(\bar{K}_m - \bar{K})^2}
$$
$$
\bar{K}_m = (\sum_{s \in S} K_{s,m})/100 \qquad (1)
$$
$$
\bar{K} = (\sum_{m \in M} K_{s,m})/\#M
$$

(S is the set of test suites, M is the set of mutants and $K_{s,m}$ is the number of times test suite s kills mutant m)

If no new mutants have been covered after 50 generations, the program is stretched to make one of the mutants easier to kill. We terminate the search if, after the stretching process is completed, no new mutants have been covered. Program stretching was originally described by Ghani and Clark [16].

In our research, we use the following three 'stretch' modes:

Path stretching
> forces branch conditions leading up to a mutant to be true or false, depending on whether the branch was taken the last time the mutant was killed.

Mutation stretching
> alters the mutation by an offset of 100, for example $x >= y \rightarrow x > y$ becomes $x > y + 100$ with the aim of increasing its impact on the program.

Branch condition stretching
> adds an offset of 100 to a difficult branch condition in order to make it easier to meet, for example $x == y$ becomes $(x <= y + 100) \&\& (y <= x + 100)$.

Algorithm 1 Synthesising an optimal subdomain $[\alpha_l..\alpha_u]$, $[\beta_l..\beta_u]$, .. $[\Omega_l..\Omega_u]$

1: Select initial random values for α_l, α_u, β_l, β_u, .. Ω_l and Ω_u.
2: **repeat**
3:     **for** $s = 1 \rightarrow 100$ **do**
4:         Generate 5 test cases from $[\alpha_l..\alpha_u]$, $[\beta_l..\beta_u]$, .. $[\Omega_l..\Omega_u]$.
5:         Count and record the number of times each mutant is killed by the test cases.
6:     **end for**
7:     Calculate subdomain fitness (see Equation 1).
8:     Sample new values from multivariate normal distribution:
$$\alpha_l' = \alpha_l + \epsilon_{\alpha_l},\ \alpha_u' = \alpha_u + \epsilon_{\alpha_u},\ \beta_l' = \beta_l + \epsilon_{\beta_l},\ \beta_u' = b_u + \epsilon_{\beta_u},\ ..\ \Omega_l' = \Omega_l + \epsilon_{\Omega_l},\ \Omega_u' = \Omega_u + \epsilon_{\Omega_u} \text{ where } \epsilon_x = \mathcal{N}(0, \sigma_x^2)$$
9:     **if** $\alpha_l' > \alpha_u'$ **then** swap(α_l',α_u') **end if**;   **if** $\beta_l' > \beta_u'$ **then** swap(β_l',β_u') **end if**; .. **if** $\Omega_l' > \Omega_u'$ **then** swap(Ω_l',Ω_u') **end if**
10: **until** $\exists m \in M, TimesKilled(m) \geq 95$

IV. SUBSET SELECTION

Subset selection is used in testing to reduce computational and human expense [17]. Subsets of test cases can be selected according to the coverage criteria met by each test case (e.g. the mutants they kill). Selecting subsets of subdomains is slightly different because test cases are sampled probabilistically. We therefore an approach that takes into account the expected number of times each subdomain kills each mutant.

Our approach borrows ideas from suboptimal feature selection. Optimal techniques (e.g. branch-and-bound) are provably equivalent to exhaustive search, but computationally prohibitive. Suboptimal techniques typically employ greedy heuristics to quickly select features that provide the greatest improvement for a criteria evaluation. Amongst other applications, they have been used to diagnose Alzheimers disease from EEG data [18], detect emotion from speech [19] and determine steel quality from textural analysis [20].

A. Sequential Floating Forward Selection

Sequential Floating Forward Selection (SFFS) is a suboptimal feature selection technique [21]. In practice, SFFS achieves optimal or near-optimal results [21]. Algorithm 2 describes the technique. Subdomains are selected one at a time that most improve the criterion evaluation (mutation adequacy). After a subdomain is added, other subdomains are removed if they improve the criterion evaluation compared to any previous evaluation on the smaller size of subset. This contrasts with other subset selection techniques (e.g. plus 1 take away r) that only allow a fixed amount of backtracking.

Algorithm 2 Sequential Floating Forward Selection

1: $k = 0$; $D_0 = \{\}$; $D_N = \{all\_identified\_subdomains\}$
2: $J(d)$ is the criteria evaluation for subset d
3: **while** $k < desired\_subset\_size$ **do**
4:     **Maximise** $J(D_k + d^+)$, **where** $d^+ \in D_N - D_k$
5:     $D_{k+1} = D_k + d^+$; $k = k + 1$
6:     **Maximise** $J(D_k - x^-)$, **where** $d^- \in D_k$
7:     **if** $J(D_k - d^-) > J(D_{k-1})$ **then**
        $D_{k-1} = D_k - d^-$; $k = k - 1$
8:     **end if**
9: **end while**

B. Our approach

We apply Sequential Floating Forward Selection (SFFS) to select small but efficient sets of subdomains. The criterion function we use to evaluate each set is shown in Equation 2. The aim is to maximise the probability of killing each mutant whilst using the smallest possible number of subdomains.

$$\sum_{m \in M} \mathbf{max}_{d \in D_x}(killed(d, m)) \qquad (2)$$

(M is the set of mutants, D_x is the current set of subdomains, $killed(d, m)$ is the number of times subdomain d kills mutant m)

SFFS is a sequential technique. Once a subdomain selection has been confirmed and backtracking has been completed, the algorithm will never go back and change it. Our approach identifies the optimal subdomains to include for each size of subset before moving on to the next one. This allows the point to be found at which adding another subdomain will not increase the mutation score significantly further. It is not necessary to decide a size of subset in advance, as our technique is able to determine the smallest size of subset that still provides a mutation score similar to the complete set.

The first step is to sample 100 test cases randomly from within the bounds of each subdomain. Subdomains are then selected for every size of set according to the sum of the maximum number of times they kill each mutant. The subdomain that (on its own) achieves the highest criterion evaluation is selected, then the two subdomains that achieve the highest evaluation and so on. The end result should be a small set of subdomains with a high probability of killing all the mutants.

Our approach is suitable for reducing the human cost of oracle construction and test evaluation. It achieves this by selecting a small set of subdomains that has similar fault finding ability to the complete set. The SFFS technique can be applied with fewer computational resources than an exhaustive search and our approach requires little set up time. All that is necessary is to sample a number of test cases from each subdomain, then apply them to each of the mutants before running the SFFS algorithm. We therefore argue that our approach is suitable for finding the smallest subset from which test cases can be sampled without significant computational expense or a detrimental effect on the fault finding ability.

V. Experiments

We applied the new technique to six programs (see Table I). TriTyp and Tcas have numerical and Boolean input parameters (see Section III regarding their representation). Replace and Schedule input text files and strings. We limit strings to 5 characters and text files to 10 characters. SingularValueDecomposition (SVD) and SchurTransformation (Schur) take matrices. We generate diagonals of a four-by-four matrix for SVD and values of a three-by-three matrix for Schur.

TABLE I
TEST PROGRAMS USED IN THE EXPERIMENTS)

Program	Mutants	LOC	Function
Tcas	267	120	Air traffic control
TriTyp	310	61	Triangle classification
Schedule	373	200	Task prioritisation
Replace	1632	500	Substring replacement
Schur	2125	497	Matrix transformation
SVD	2769	298	Matrix decomposition

We set up experiments to answer the following research question in regard to selecting subsets of subdomains for mutation adequacy:

Is it possible to reduce the number of subdomains without significantly affecting the mutation score?

To answer this question, we applied Sequential Floating Forward Selection (SFFS) to select small but efficient sets of subdomains. A Covariance Matrix Adaptation Evolution Strategy (CMA-ES) was used to optimise subdomains for each of the six programs included in this study. The subdomain optimisation process was applied 100 times for each program to produce 100 sets of subdomains. This was done so that an average result could be achieved for the effect of subdomain optimisation on each program. Subdomain selection was therefore applied 100 times to each set of subdomains identified for each program through the process of subdomain optimisation.

We take advantage of the sequential nature of our approach to evaluate every possible size of subset. Starting with the single most efficient subdomain, we add subdomains one at a time, continuing up to the complete set. At each step, the subdomain that achieves the highest criterion evaluation is selected. In practice, this process is halted once a criterion evaluation is reached similar to that of the complete set. Yet, for the purposes of experimentation, we continue the process until the end. We record the minimum, maximum and average mutation score for each program and size of subset.

Subsets that achieve a similar mutation score to the complete set of subdomains indicate the potential for improving fault finding efficiency. Of particular interest is the proportion of subdomains that can be removed without significantly reducing the mutation score. The criterion we use, to determine whether our technique has been successful in improving efficiency for a particular program, is that the number of subdomains can be halved with little effect on the mutation score.

VI. Results

We evaluated the effect reducing the number of subdomains had on the mutation score for each program. Figure 3 and Table II present the results of subdomain optimisation for the six programs in our experiments. Figure 4 and Table III present the results of selecting subsets of optimised subdomains. It is possible to select fewer subdomains with minimal decrease in mutation score for all the programs, except SVD and Schedule. For example, selecting a quarter of the subdomains of Tcas only reduces the mutation score by 3.6%.

After Schur, Schedule and SVD had the smallest number of subdomains identified by the optimisation process. This limits the opportunity for redundant subdomains and makes it more likely for reducing the number of subdomains to have a significant effect on the mutation score. The reason why this is not the case for Schur may be because its mutants are easy to kill even by random testing.

The minimum, maximum and average values are consistently close to each other. This suggests that our interpretations can be relied upon. The minimum mutation score when selecting the first few subdomains of Schur is very low, but this changes quickly after the fifth subdomain is added. It is likely caused by a single optimisation run where no one subdomain has a high mutation score by itself.

It is also worth noting that the graphs are not completely smooth because each optimisation run identified a different number of subdomains. This is why, for example, the mutation score of Schedule appears to decrease at one point when subdomains are added.

There is a relationship between the subdomains that are selected and the order in which they were identified by the optimisation process. Take for example, all the subdomains selected for Tcas in subsets of size 10 (see Figure 2). Subdomains are more likely to be included if they were discovered later in the optimisation process. Subdomains that are discovered earlier are more likely to be redundant because they aim to cover mutants more broadly, before some of the mutants have been put aside. This suggests that it is useful to focus on identifying subdomains for harder to kill mutants. This may even eliminate the need for subset selection.
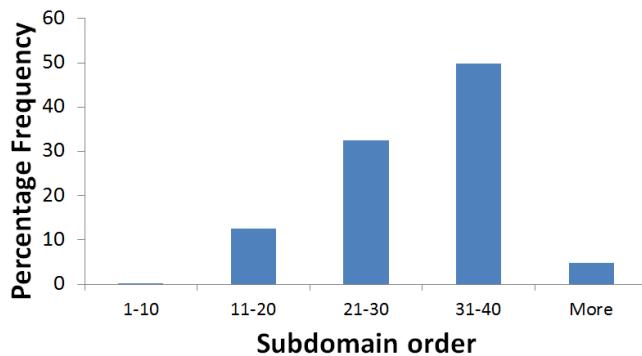

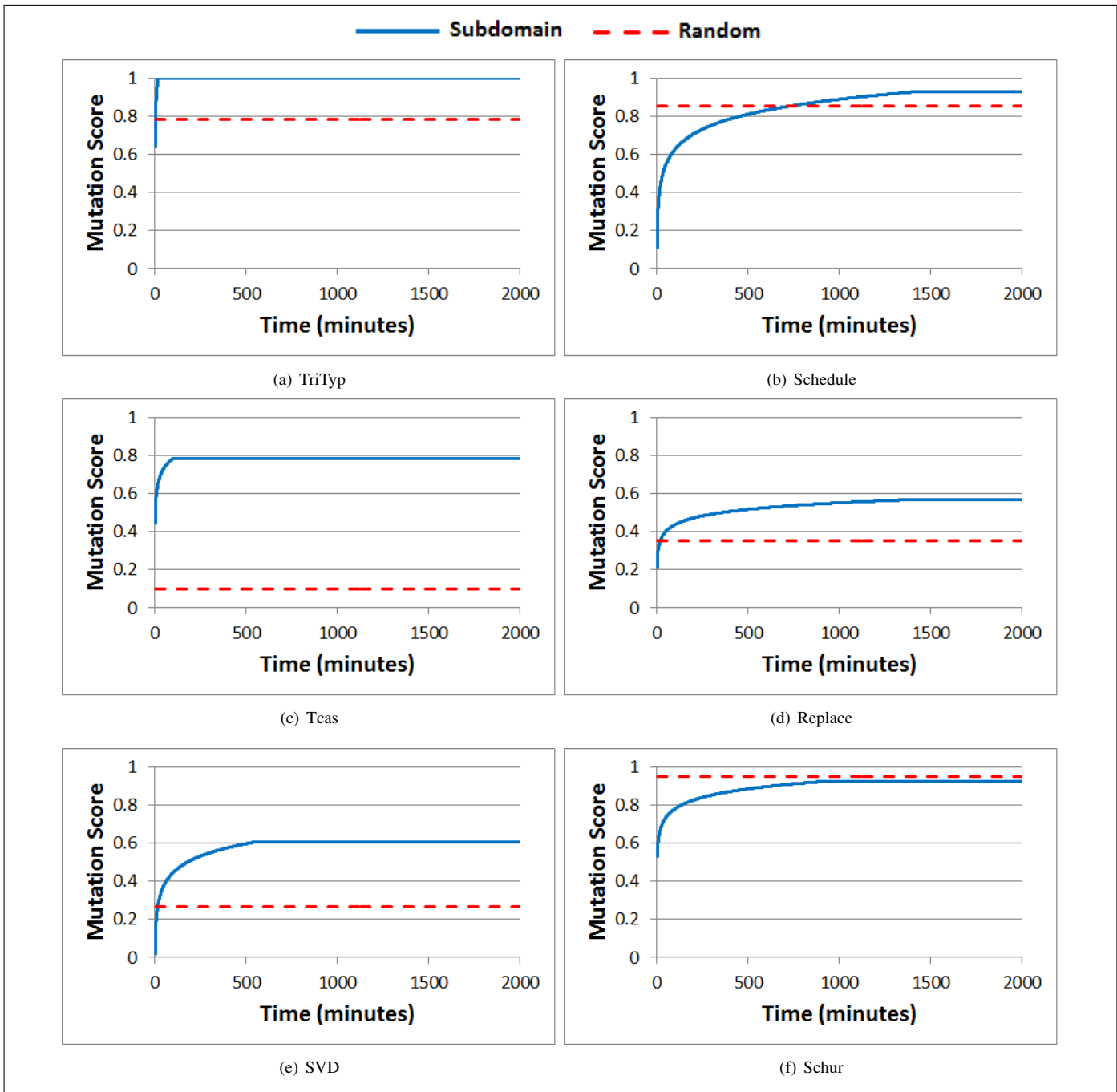Fig. 2. Frequency order of subdomains selected for Tcas subsets of size 10

Fig. 3.   Percentage of mutants covered by evolved subdomains (averaged over 100 trials)

TABLE II
SUMMARY OF RESULTS (AVERAGED OVER 100 TRIALS)

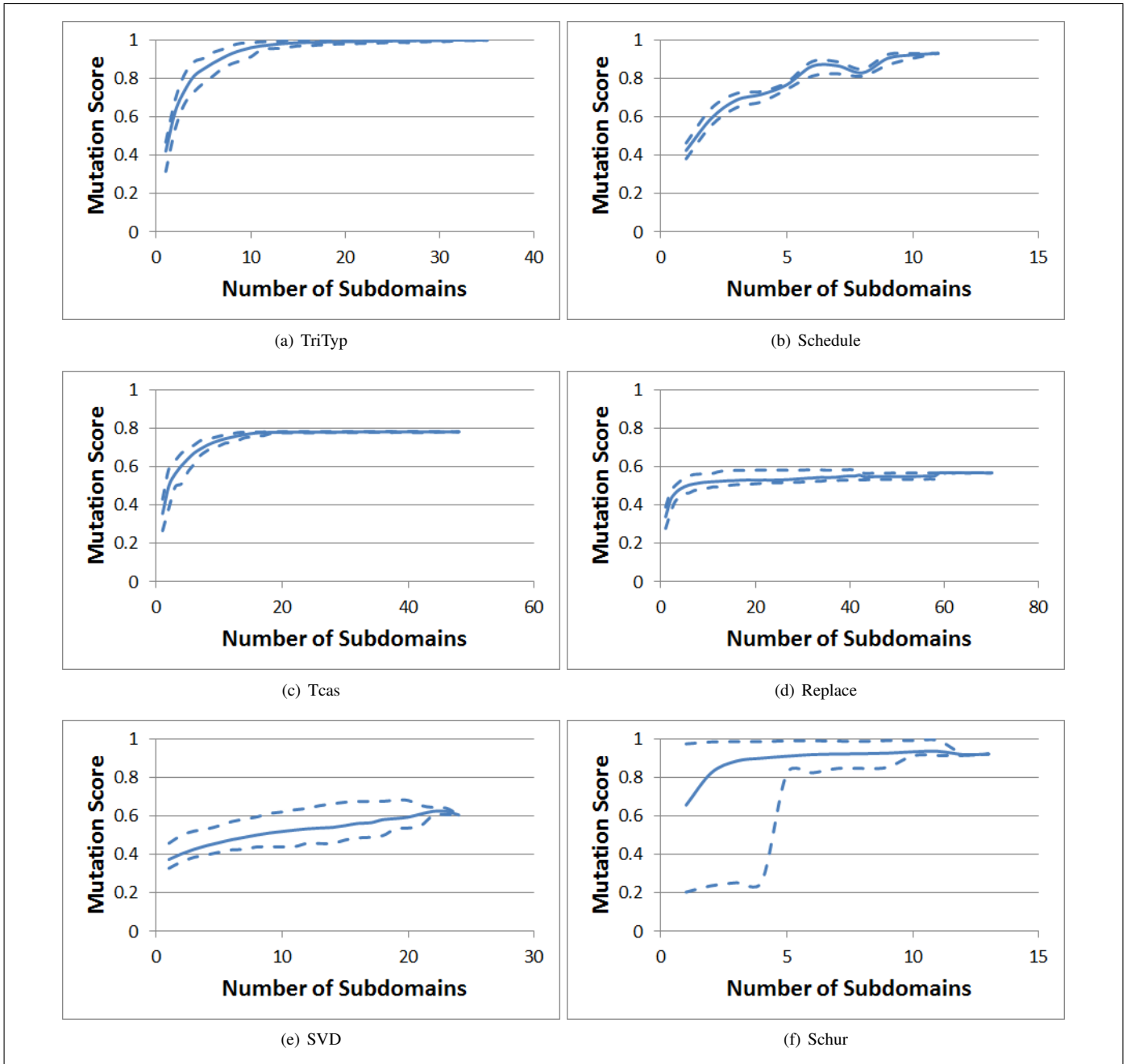| Program	Subdomain Testing		Random Testing	Subdomains	Test Cases
	Mutation Score	Time (mins)	Mutation Score		
Tcas	0.780	50.6	0.0945	40.6	205
TriTyp	0.998	8.00	0.780	26.9	135
Schedule	0.930	1310	0.850	8.01	40
Replace	0.566	1410	0.350	90.7	455
SVD	0.632	546	0.263	25.4	125
Schur	0.920	885	0.95	8.61	45

Fig. 4.   Percentage of mutants covered by evolved subdomains (averaged over 100 trials)

(NB: Dotted lines represent the minimum and maximum mutation scores from 100 trials. Solid lines represent the average)

TABLE III
SUMMARY OF RESULTS (AVERAGED OVER 100 TRIALS)

Program	25%	50%	75%	100%
Tcas	0.752	0.778	0.779	0.780
TriTyp	0.946	0.988	0.994	0.998
Schedule	0.686	0.862	0.828	0.930
Replace	0.523	0.542	0.547	0.566
SVD	0.460	0.531	0.579	0.603
Schur	0.883	0.920	0.921	0.920

VII. Related Work

Selection techniques are used to improve regression test efficiency [17]. Test cases can be selected that cover infrequently met test criteria [22], meet the most number of unmet criteria [23] or consistently contribute to the overall evaluation [24]. Test selection criteria are typically deterministic, since test cases produce the same result each time they are executed. In contrast, our approach to subdomain set selection uses non-deterministic criteria. This is because test cases are sampled probabilistically from within the bounds of each subdomain.

Test selection methods are often based on the greedy algorithm [17] - They add test cases one at a time, selecting at each step the test case that most improves the criteria evaluation. The problem with this approach is that criteria met by earlier test cases are also often met by a combination of test cases selected later in the process, thus making some of the earlier test cases redundant [17]. Tallam et al. [25] addressed this issue by removing redundant test cases before applying the greedy algorithm. In addition to this, Jeffrey and Gupta [26] use a second set of requirements to determine whether a test case really is redundant. Our solution is to select test cases based on the results of multiple evaluations to avoid overfitting the selection. We also use backtracking as often as it helps the evaluation, so as to eliminate redundant subdomains.

Distribution-based techniques have also been used to select more efficient sets of test cases. Faults typically produce errors for specific ranges of input values [27]. Many techniques have been developed to reduce the time it takes to find the first error by distributing test cases more evenly over the input domain. They maximise the distance between new and existing test cases [27], set up exclusion zones [28], use quasi-random sequences [29] or employ lattices [30]. Nevertheless, the added expense involved with these techniques can often outweigh the benefits [31]. We use mutation analysis to predict and improve fault finding capability for the particular program under test. Our subdomains are evolved and selected to target specific ranges of input values that reveal faults as errors.

There have been other attempts at tailoring the range of test input for specific programs. Andrews et al. [32] use a genetic algorithm to improve statement coverage by optimising the range of values for each scalar type. Poulding and Clark [33] present a Bayesian network representation that allows for dependencies between parameter values. Thus, the sampling distribution for a second parameter may depend on the specific value sampled for the first. The input distribution is specified in terms of bins (similar to our subdomains). Their goal is to maximise the least covered element, a criteria originally developed by Thévenod-Fosse and Waeselynck [34].

We optimise independent subdomains (without dependencies) for each input parameter and use mutation adequacy as our selection criterion. Mutation adequacy is more stringent than statement or branch coverage [9]. There has been much research into test generation for mutation adequacy [35] [36] [37] [38], but our work is the first to optimise and select subdomains from which efficient test cases can be sampled.

VIII. Conclusions

Our subdomain selection technique has two stages. First, subdomains are optimised for their ability to kill mutants consistently, then a small set of subdomains is selected that is able to kill mutants more efficiently. Sampling test cases from optimised subdomains achieved a higher mutation score than random testing, except for one trivially easy to test program. The subdomain selection technique reduced the number of subdomains for four programs with little affect on mutation score. Our is therefore suitable for some, but not all programs.

Subdomain optimisation is computationally expensive compared to random testing. Yet once efficient subdomains have been identified, the cost of sampling new test cases is insignificant and the ability of the new test cases to find faults is significantly increased. Subdomain optimisation is therefore a useful technique for regression testing. In our experiments, subdomain optimisation did receive a lower mutation score than random testing for one program (Schedule), but this was only because random testing had already achieved a perfect (or close to perfect) mutation score in every trial. It seems that subdomain optimisation is applicable whenever the program is complicated enough to merit it.

Our approach to subdomain selection is computationally inexpensive. It adds little additional cost, but can significantly improve the efficiency of a subdomain set. It is, however, not effective in all cases: reducing the number of subdomains had an immediate negative effect on Schedule and SVD. Few test cases were needed for Schedule in the first place, but for SVD there must be some other reason. Subdomain selection is guaranteed to have an effect on mutation score unless there is some overlap in mutant coverage between subdomains. In the case of SVD there is little such overlap. Subdomain selection is therefore only successful for certain programs, but for those programs it has the potential to provide significant improvement.

IX. Further work

Subdomains identified earlier in the optimisation process tend to kill large numbers of easy-to-kill mutants. As a result, they are more likely to prove redundant later, as subdomains are added that target smaller groups of mutants with greater precision. This suggests that focussing on difficult to kill mutants may reduce the computational cost of subdomain optimisation and selection. For this reason, we will focus our further work on three main areas of research:

1) Investigate the potential to improve computational efficiency by identifying and removing redundant subdomains earlier during the optimisation process.

2) Evaluate different orderings of mutants to minimise overlap in the coverage of subdomains optimised for them.

3) Employ static analysis techniques to identify difficult to kill mutants and mutants that should be grouped together and targeted for subdomain optimisation.

REFERENCES

[1] J. H. Andrews *et al.*, "Tool support for randomized unit testing," in *Proc. 1st Int. Workshop Random Testing*, Portland, ME, 2006, pp. 36-45.

[2] C. C. Michael *et al.*, "Generating software test data by evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1085-1110, Dec. 2001.

[3] J. W. Duran, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438-444. IEEE Press, Piscataway, July 1984.

[4] J. H. Andrews *et al.*, "Is mutation an appropriate tool for testing experiments?," in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, 2005, pp. 402-411.

[5] G. J. Myers *et al.*, *The Art of Software Testing*. Hoboken, NJ: Wiley, 2011, pp. 35.

[6] M. Patrick *et al.*, "Using mutation analysis to evolve subdomains for random testing," in *Proc. 8th Int. Workshop Mutation Analysis*, Luxembourg, Luxembourg, 2013.

[7] M. Patrick *et al.*, "Efficient subdomains for random testing," in *Proc. 5th Int. Symp. Search Based Software Engineering*, St. Petersburg, Russia, 2013, pp. 251-256.

[8] Y. Jia and H. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649-678, Sept. 2011.

[9] P. G. Frankl *et al.*, "All-uses versus mutation testing: an experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, no. 3, pp. 235-253, Sept. 1996.

[10] T. Bäck, *Evolutionary Algorithms in Theory and Practice*, Oxford, England: Oxford University Press, 1996, pp. 66-90.

[11] B. Babb *et al.*, "State-of-the-art lossy compression of martian images via the CMA-ES evolution strategy," *Int. Soc. Optics Photonics*, vol. 8305, pp. 22-26, Mar. 2012.

[12] V. Nissen and S. Gold, "Survivable network design with an evolution strategy," in *Success in Evolutionary Computation (Studies in Computational Intelligence)*, J. Jung *et al.*, Eds. Berlin, Germany: Springer, 2008, pp. 263-283.

[13] J. Jung, "Using evolution strategy for cooperative focused crawling on semantic web," *J. Neural Comput. Appl.*, vol. 18, no. 3, pp. 213-221, Feb. 2009.

[14] N. Hansen, "The CMA evolution strategy: a comparing review," *Towards a New Evolutionary Computation*, vol. 192, pp. 75-102, 2006.

[15] N. Hansen *et al.*, "Comparing results of 31 algorithms from BBOB-2009," in *Proc. 12th Genetic Evolutionary Computation Conf.*, Portland, OR, 2010, pp. 1689-1696.

[16] K. Ghani and J. Clark, "Widening the goal posts: program stretching to aid search based software testing," in *Proc. 1st Int. Symp. Search Based Software Engineering*, Windsor, England, 2009.

[17] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67-120, Feb. 2012.

[18] K. Akrofi *et al.*, "Classification of Alzheimers disease and mild cognitive impairment by pattern recognition of EEG power and coherence," in *Proc. 35th Int. Conf. Acoustics Speech Signal Processing*, Dallas, TX, 2010, pp. 606-609.

[19] M. Brendel *et al.*, "A quick sequential forward floating feature selection algorithm for emotion detection from speech," in *Proc. 11th Annu. Conf. Int. Speech Communication Association*, Makuhari, Japan, 2010, pp. 1157-1160.

[20] D. Kim *et al.*, "Determination of steel quality based on discriminating textural feature selection," *Chemical Engineering Science*, vol. 66, no. 23, pp. 6264-6271, Dec. 2011.

[21] P. Pudil *et al.*, "Floating search methods for feature selection with nonmonotonic criterion functions," in *Proc. 12th Int. Conf. Pattern Recognition*, Jerusalem, Israel, 2013, pp. 279-283.

[22] M. J. Harrold *et al.*, "A methodology for controlling the size of a test suite," *Trans. Softw. Eng. Meth.*, vol. 2, no. 3, pp. 270285, July 1993.

[23] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135141, Nov. 1996.

[24] J. Offutt *et al.*, "Procedures for reducing the size of coverage-based test sets," in *Proc. 12th Int. Conf. Testing Computer Software*, Washington D.C., 1995, pp. 111123.

[25] S. Tallam, N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35-42, Jan. 2006.

[26] D. Jeffrey, N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 108123, Feb. 2007.

[27] T. Y. Chen *et al.*, "Adaptive random testing," in *Proc. 9th ASIAN Computing Science Conf.*, Chiang Mai, Thailand, 2004, pp. 320-329.

[28] K. P. Chan *et al.*, "Restricted random testing," in *Proc. 7th Int. Conf. Software Quality*, Helsinki, Finland, 2002, pp. 321-330.

[29] T. Y. Chen and R. G. Merkel, "Quasi-random testing," *IEEE Trans. Reliab.*, vol. 56, no. 3, pp. 562-568, Sept. 2007.

[30] J. Mayer, "Lattice-based adaptive random testing," in *Proc. 20th Int. Conf. Automated Software Engineering*, Long Beach, CA, 2005, pp. 333-336.

[31] A. Arcuri and L. Briand, "Adaptive random testing: an illusion of effectiveness?" in *Proc. 20th IEEE Int. Symp. Software Testing Analysis*, Toronto, Canada, 2011, pp. 265-275.

[32] J. H. Andrews *et al.*, "Genetic algorithms for randomized unit testing" in *IEEE Trans. Software Eng.*, vol. 37, no. 1, pp. 80-94, Jan. 2011.

[33] S. Poulding and J. A. Clark, "Efficient software verification: statistical testing using automated search," *Trans. Software Eng.*, vol. 36, no. 6, pp. 763-777, Nov. 2010.

[34] P. Thévenod-Fosse, and H. Waeselynck, "An investigation of statistical software testing," *J. Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 526, 1991.

[35] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. 21st IEEE Int. Symp. Software Reliability Engineering*, San Jose, CA, pp. 147-158.

[36] L. Zhang *et al.*, "Test generation via dynamic symbolic execution for mutation testing," in *Proc. IEEE Int. Conf. Software Maintenance*, Timişoara, Romania, 2010, pp. 533-543.

[37] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution." in *Proc. 21st IEEE Int. Symp. Software Reliability Engineering*, San Jose, CA, 2010, pp. 121-130.

[38] M. Harman *et al.*, "Strong higher order mutation-based test data generation," in *ACM SIGSOFT Symp. Foundations Software Engineering*, Szeged, Hungary, 2011, pp. 212-222.