Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics

Robert Feldt, Richard Torkar, Tony Gorschek and Wasif Afzal
Dept. of Systems and Software Engineering
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
{rfd|rto|tgo|waf}@bth.se

Abstract

Search-based software testing (SBST) has shown a potential to decrease cost and increase quality of testing-related software development activities. Research in SBST has so far mainly focused on the search for isolated tests that are optimal according to a fitness function that guides the search. In this paper we make the case for fitness functions that measure test fitness in relation to existing or previously found tests; a test is good if it is diverse from other tests. We present a model for test variability and propose the use of a theoretically optimal diversity metric at variation points in the model. We then describe how to apply a practically useful approximation to the theoretically optimal metric. The metric is simple and powerful and can be adapted to a multitude of different test diversity measurement scenarios. We present initial results from an experiment to compare how similar to human subjects, the metric can cluster a set of test cases. To carry out the experiment we have extended an existing framework for test automation in an object-oriented, dynamic programming language.

1. Introduction

Developing good tests for software systems is expensive and much effort in recent years has gone into methods to automate parts of this process. Search-based software testing techniques have surfaced as one of the more promising solutions [13, 16]. By applying local or population-based search algorithms to search for test data and/or test cases we can, potentially, both decrease the human effort needed for developing the tests as well as increase the effectiveness of the tests themselves.

Several previous studies have shown the potential of this approach [12, 21, 22]. The fitness functions used to direct the search is often based on some coverage criteria, like statement or branch coverage, even though other approaches have been reported [3, 14]. However, only a few studies have used relative fitness functions that compares newly found tests to the ones previously in the test set, to optimize the test set as a whole [2]. This is unfortunate since an optimal set of tests is what is ultimately needed.

A fundamental fact of software testing is that tests cannot show the absence of faults just their presence [11]. However, in practice test sets are not only used to uncover faults, they are also used in arguments for the quality of the software. A key to making such dependability arguments is that we have test cases that humans judge as being cognitively dissimilar. It is not likely that including many tests that are regarded as the same or very similar would strengthen such arguments. To be able to search for such cognitively different test cases we need fitness functions that can measure them. Existing proposals to measure software and test differences in the literature are either limited in which types of situations they can be applied, disregard some important aspect of the differences to be measured and/or are very complex [5, 9, 19].

From a practical point of view previous studies in SBST have also been lacking in that they do not integrate with existing testing and specification frameworks. This has hindered more wide-spread use. For real-world use it is likely that software developers and testers will want to mix different types of tests; some handwritten and some found via a search. Systems supporting this will be easier to learn and use if the different parts are well integrated with each other to support different types of test creation within the same framework. With this in mind we have extended an existing, behavior-driven specification framework to be able to trace tests and calculate test diversities.

In this paper we investigate test diversity metrics and evaluate their potential in ranking tests based on cognitive similarity. In particular, the paper:

1. Presents a model for test variability with points of vari-

ations, that gives a framework for specifying a family of different test diversity metrics.

2. Proposes the use of a theoretically optimal diversity metric for testing, and a practically, useful approximation of it, for test diversity calculations.

3. Briefly describes our extension to a testing framework to collect traces for diversity calculations.

4. Presents results from an initial experiment to evaluate one such diversity metric.

The rest of this paper if structured as follows. Section 2 introduces a test variability model, and Section 3 describes a universal diversity metric that can be applied to testing. Based on the model and the metric, Section 4 proposes a practically useful test diversity metric. Sections 5– 6 present an initial experiment to compare one particular metric for clustering to humans. Section 7 discusses the results and Section 8 presents related work. Section 9 concludes the paper.

2. Test Variability Model

Figure 1 shows a simple model of running a software test. There are five main steps when executing the software under test (SUT) in order to test it: test Setup, Invocation, Execution, Outcome and Evaluation. In the figure the rectangular nodes refer to these five phases while the eleven elliptical nodes are variation points, i.e. aspects on which two tests might differ.

Test setup is the source code used to setup the SUT for the test. This involves both general setup (SG), which is common to all (or many) different test executions, and setup code specific to the current test (SS). We have chosen not to consider the state of the system as part of the input to the test; this choice increases the level of detail in the model and allows for finer control when measuring test difference. Common to both types of setup is that it only sets up the SUT, it is not concerned with generating or creating the test data to be used in the invocation of the SUT. Instead, creating the arguments (IA) is part of invocation and distinguished from the actual call of the SUT (IC). In test execution we can distinguish several aspects in which tests can differ: in how the flow of control (XC) is transferred and in how state changes happens (XS). The fourth step in our model is the test outcome where we consider both SUT state changes (OS) and the actual return values (OR) as variation points.

In addition to test execution, tests can differ in how the behavior of the SUT is evaluated. This part of the model can involve evaluating both the outcome (OE) or aspects of the execution (EE) such as e.g. performance. Different testers might have different views on which properties of the behavior should be checked, and not all of them might be complete, in the sense of fully describing the desired behavior. We thus avoid any notion of an oracle, and instead note that tests might differ in which properties are checked for and how in the evaluation of a test case.

Apart from the ten variation points above, tests can also differ in what are the goals of running the test (G). We include this variation point since test cases can be used in arguments that the SUT has a certain quality level. Having clear goals that fit with the rest of the tests in a set is important for this type of arguments. For example, two different methods for creating a test, e.g. boundary value analysis and mutation testing, might lead to the exact same test, but their goals with the test might be different. Even though this type of variation might be rarely documented or used in practice we include it for completeness. Also, in place for the actual goals we might find other types of documentation relevant for the test, such as for example comments in the test code or in test plans.

In the following we refer to our model as the VAT model (VAriability of Tests). It is primarily a dynamic model of test case execution, i.e. it is the actually executed code for a certain variation point that we focus on. Depending on the execution model of the programming language or virtual machine or how we choose to use the model static information for a variation point may also (or solely) be used. However, below we focus on measuring distances between information on variation points collected by tracing the test case while it executes.

3. A Universal Test Diversity Metric

The VAT model introduced above has several points of variation that we want to compare between different tests. Given that we choose one or a few variation points that we are interested in, what method should we use to calculate an actual numerical distance value?

The solution in the literature so far has been to devise specialized methods of calculating metrics. Bueno et al. use an Euclidian distance between input vectors [5]. Ciupa et al. specify a number of different factors of interests in comparing object invocations and then weigh them together [9]. Nikolik defines test diversity measures based on the frequency of executing statements or parts of statements when running a test [19].

The problem with these approaches is that for each new variation point and aspect of a test we introduce or consider, we will have to develop a specialized metric for it. As an alternative, we propose that we look at what would be an information-theoretically optimal diversity metric that can be applied in several of these variation points and without us having to adapt it for each aspect we want to measure.

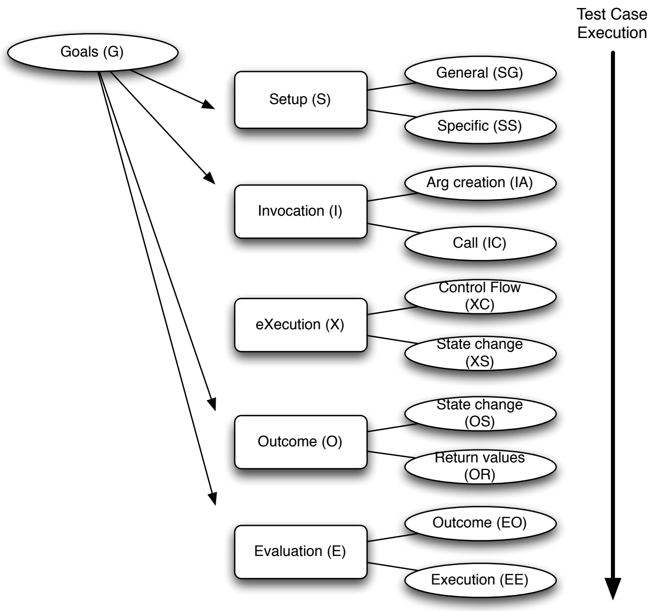

Figure 1. Execution of a software test.

This may sound like a holy grail but recent advances in information theory has produced results of this generality [6]. In the rest of this section we describe them and in the following sections we describe how to apply them for test diversity measurement.

A diversity measure calculates the distance between two or more objects. Measuring the distance between two objects is enough; if we have a method for that we can extend it to measure distances between sets of objects. Bennett et al. have introduced a universal cognitive similarity distance called Information Distance [4]:

> The information distance between two binary strings, x and y, is the length of the shortest program that translates x to y and y to x.

This is based on the notion of Kolmogorov complexity, $K(X)$, which measures the informational content of a binary string x as the length of the shortest program that prints x and then halts [15]. More specifically, it builds on the conditional Kolmogorov complexity, $K(X|Y)$, i.e. the length of the shortest program that can print X given Y as input.

Information distance is universal since it has been proven to be smaller than any other admissible similarity measure, up to an additive term. This means that information distance is as good as any other thinkable similarity measure. In the words of Bennet:

> [information distance] discovers all effective feature similarities or cognitive similarities between two objects; it is the universal similarity metric.

For search-based testing, when we want to find tests that are cognitively different from the ones we have already found, this is a very important result. As long as we devise some way to dump information about a (part of a) test, we can dump this information for two different tests and apply the information distance to get their distance. Thus, we can for example use this as a fitness function in search and optimization algorithms to find better tests.

The generality of information distance, is the reason for why we allowed ourselves to include such a fuzzy element of the VAT model as the test goals. As long as we can generate two strings describing the test goals of two different tests we can measure their similarity (or diversity). We could, for example, use the text stating the different goals as described by the tester or even the customer[1]. Formally we define:

Definition 1 *A complete VAT trace of a test is a string with all the information about the actual execution of a test for all the variation points in the VAT model.*

We propose to use information distance of the complete VAT traces of two tests as the *Universal Test Distance* (UTD). Given the generality and power of information distance, UTD should, in theory, be able to discover all cognitive similarities between two test executions.

[1]There is an important issue of syntactic versus semantic differences here though that needs to be evaluated. See the discussion for more details

Definition 2 *The* Universal Test Distance*, denoted* Δ_{VAT}*, in the VAT model is the information distance between the complete VAT traces of two tests.*

It might not always be the case that we can dump all information during a test execution to get a complete VAT trace. We might only be able to get the information from a few of the variation points. The information distance can be applied also to such traces, so we are really proposing a whole family of different test distances. Depending on which information we decide to include in the traces, our distances will measure different things. We have thus simplified our problem from one of devising a distance measure that captures important differences to one of choosing which information we think is important for uncovering meaningful differences.

However, a big problem with Information Distance is that it, like Kolmogorov complexity, is uncomputable. How should we find the shortest program that can turn two strings into each other? There is no method to calculate Information Distance. In the next section we describe the Normalized Compression Distance (NCD), introduced by Cilibrasi, to approximate the Information Distance [6].

4. A Practical Test Diversity Metric

The uncomputability of the Information Distance metric can be overcome by using data compressors to approximate Kolmogorov complexity. Real-world compressors like gzip and bzip2 will not be as good compressors as the Kolmogorov complexity but can be used to approach it from above [6]. In his thesis, Cilibrasi introduced the Normalized Compression Distance, NCD:

$$NCD(x,y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

where $C(x)$ is the length of the binary string x after compression by the compressor C and $C(xy)$ is the length of the concatenated binary string xy after compression by the compressor C. In practice, NCD is a non-negative number $0 <= r <= 1 + \epsilon$, where ϵ is small and depends on how good an approximation of Kolmogorov complexity the compressor (C) is. For modern compressors like gzip and bzip2, ϵ is typically 0.1 and more recent compressors can even come close to 0, i.e. being excellent approximations of Kolmogorov complexity.

Cilibrasi et al have successfully evaluated NCD in multiple areas as diverse as genomics, virology, languages, literature, music, handwritten digits, and astronomy [8]. NCD is a practical distance metric based on theoretically sound results. We propose to use it as a generally useful test diversity metric. By dumping information about tests into linear strings we can calculate the NCD between tests. This gives a general test diversity metric that is applicable in a large number of contexts. In the following we explore the usability of the NCD as a test diversity metric in an experiment.

5. Experiment

We want to evaluate if NCD can be used as a practical diversity metric to measure the cognitive distances between tests. Based on the arguments about NCD and ID we have formulated the following main hypothesis to be tested in the experiment:

> $H_{\mathbf{cog}}$: Ordering tests based on their Δ_{VAT} distance cannot be distinguished from how a human would order the tests based on their 'cognitive similarity'.

For the metric to be useable as a stand-in for humans we want it to find differences between tests that are cognitively meaningful. If we can refute $H_{\mathbf{cog}}$ our proposed metric might still be useful as a distance metric but we would have to find other justifications for it than it being a substitute for human judgement.

For an initial experiment to evaluate the potential of our proposed metric we have used the triangle classification problem [17]. This is almost the smallest thinkable, i.e. toy, problem in the testing literature and education. Of course this will not allow us to generalize to larger and more realistic software systems. However, if Δ_{VAT} does not work for this, simple problem, it might not warrant further investigation.

Based on an extensive list of tests for the triangle classification problem we created implementations of these tests in a specification framework for the dynamic programming language Ruby [20]. The framework is called Bacon and is a recent behavior-driven test/specification framework similar to the more well-known RSpec framework [18, 1]. We have chosen Bacon since it is smaller, and thus more malleable, than RSpec, allowing us to more easily extend it to dump test traces to file while running the tests.

Our extension uses a Ruby function to hook into the program interpreter and trace the execution of every important event. When the trace function we have added detects a 'line' event indicating the execution of a new line of code in the program executing it will save information about the line number, the code on that line as well as information about all the values of variables being accessed on this line of code. This information is saved to a file for later analysis. It constitutes a trace of both the code and the data values that are accessed for each line of code. Note however, that it is not a complete VAT trace of the test. We only trace the invocation, execution and evaluation steps in the VAT model and we do not save all accessible information during the trace. For example, we cannot follow the execution of Ruby's internal methods when called by the SUT.

A total of 25 different test cases for the triangle classification problem were included in the tests. There where tests for all the four types of classifications (illegal, scalene, isosceles and equilateral triangles) as well as invalid inputs such as too few or too many arguments or strings instead of numbers. We further used both small integer values as well as very large ones (Ruby supports arbitrary-precision integers).

All the test cases were implemented in a version of Bacon that we extended to save a Δ_{VAT} trace. We ran all the test cases, collected the traces and then used Cilibrasi's et al. CompLearn toolkit [7] to calculated the NCD distances and create a quartet tree representing the hierarchical clustering of the test case traces [6]. The quartet tree method is a powerful technique to visualize the distance matrices produced by an application of NCD to a set of objects. It is a heuristic optimization method that tries to find an unrooted binary tree, clustering the objects so that objects with small distances are closer in the tree than objects with high distances.

All the 25 test cases were also stripped of documentation strings, randomly ordered and labeled before they were given to three human subjects. These subjects worked independently and were told to cluster the test cases based on their subjective judgement of test case similarity. They did not have access to the implementation we used when collecting the traces. Their clustering was thus black-box, i.e. based on the test code only.

6. Results

All three human subjects classified the test cases in a very similar way. They used rooted non-binary trees and divided the test cases into five main clusters based on their output, i.e. either there was an exception because the arguments were not valid or there was one of four different triangle classifications. Only for one of these main clusters did they further divide into sub-clusters. All three of them sub-clustered the test cases with illegal triangles into groups containing zeroes, positive or negative side lengths. Two of them also singled out triangles that violated the triangle property ('one side cannot be longer than the sum of the other two'). Common to all three human clusterings was that they did not use binary trees so within sub-clusters there were several test cases that were not fully ordered on their similarity. This made it hard to test $H_{\mathbf{cog}}$ statistically. Below we discuss some of the features of the clustering produced from the UTD matrix that corresponds to or differs from the clustering done by the human subjects.

Figure 2 shows the quartet tree showing the clustering that CompLearn calculated based on the Δ_{VAT} distance matrix calculated from the test case traces. In the picture, elliptical leaf nodes are the test cases and empty, intermediate nodes, are used to show distance between test case nodes or other intermediate nodes (representing a cluster). The dashed edges on the rim of the figure shows the Δ_{VAT} distances between neighboring test case nodes (or really the traces for the test cases in the nodes). The labels of the test case nodes briefly characterize the test case. The first position can be either S, for short integers, L, for long integers, or F, for floating point side lengths used in the test invocation. The second position of the label indicates the outcome of the test with E representing equilateral, I isosceles, S scalene and X illegal or that an exception was raised. The third position is an underscore which is then followed by a description of the actual input arguments in the call to the triangle classification method. Note that for some of the test cases with long integers we could not use the full value and used an L followed by a number to indicate a particular long integer number. As an example of a label 'SS_3,4,5' is the test case using 3, 4 and 5 as the short integer arguments and that expects the method to classify the triangle as a scalene triangle.

We should note that the algorithms used for producing this figure are non-deterministic so different runs can give slightly different trees from one and the same distance matrix. We ran the CompLearn tree building command several times to evaluate if these differences were important and they seem not to be. Even though the actual positions on the produced trees might differ, the relative positions, with only a few exceptions, seems to correspond to the figure (see Figure 2).

We can see that the method have found several groupings that are intuitively correct, for example, test cases that are simply permutations of the same side lengths are grouped close to each other (see for example the group of test cases in the lower left corner where all the scalene triangles that are permutations of the sides 3, 4 and 5 are clustered). Closest to this group is the test case with the same side lengths given as floats. This makes sense since the implementation of the triangle classification problem we used did not treat floats differently from integers; they only differ in which Ruby internal methods are executed. Since we only included user-developed code in the trace the executed code is the same at this level and thus the test case is grouped close to its integer counterparts. The method has also correctly grouped the equilateral triangle case (SE_1,1,1) closer to the group of isosceles triangles than the scalene ones. This is intuitive since the former share the property of having two sides of equal length.

In the middle of the figure we can see a fairly sharp divide between a majority of invalid triangle test cases in the right part of the figure while in the left there is a majority of valid triangles. One exception is 'LX_L4,L5,longer'

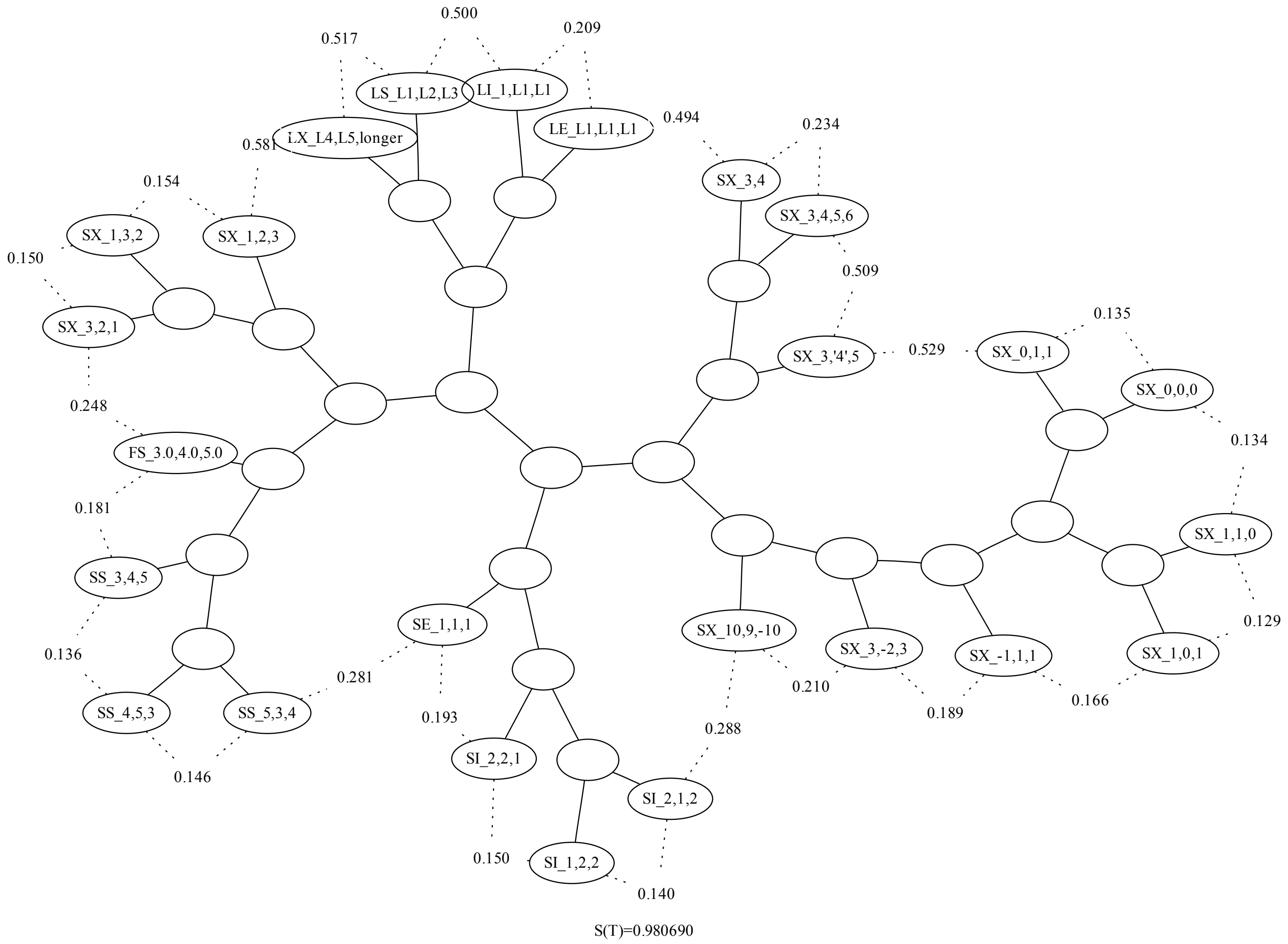

Figure 2. Quartet tree clustering based on Δ_{VAT} distances between test case traces.

which is an illegal triangle in a group of valid triangles. This seems counterintuitive. We studied the traces for this and its neighboring test cases in detail and found that this happens because the triangle sides used in these test cases are very long numbers, and dominates the trace strings. The fact that these test cases share long runs of integer characters seems to overtake the differences they show in the statements executed. However, we will need more empirical data to evaluate this fully. Anyhow, the relatively large Δ_{VAT} distances shown, indicates that these test cases are quite distant even though they share some structure.

Another exception in the left side of the figure is the three invalid triangles in the upper, left corner. All these three test cases have triangles with permutations of sides in which two sides of the triangle have the same length as the third one, which is illegal. Our human subjects have all clustered these three test cases together with the other test cases with illegal triangle side lengths. By analyzing the traces of 'SX_3,2,1' and 'FS_3.0,4.0,5.0' we can see that they differ only in the last statement executed, i.e. the one returning the classification. So they are similar because of our particular implementation of the triangle classification problem. With the condition statements in a different order the traces between these two would have been larger and thus their distances.

7. Discussion

The results indicate that applying the UTD, i.e. NCD to measure diversity between traces in the VAT model, can be used to cluster test cases by cognitive similarity, i.e. in a way that is intuitive to humans. This has important implications for search-based software testing since we can then use these metrics when trying to find tests that are more diverse and more meaningful to a human software developer, thus increasing their quality and usefulness in dependability and quality arguments.

The unintuitive exceptions in the automatically clustered test cases could be explained by the actual choice of in-data or by the actual implementation of the tested code. This is a reflection of the fact that we included a majority of the steps in the VAT model (except the outcome step, and setup, which is not present for our chosen problem) in our traces. Thus the metric we employ is not simply a black-box diversity metric but includes also white-box elements such as the path taken through the code and internal state during execution. This is in contrast to the human subjects that could only do the clustering based on the inputs themselves. By excluding information from the trace we could possibly get a clustering that is closer to the clustering done by humans. However, this will have to be evaluated in future research, since for many pieces of software, like in the one we experimented with, there might not be enough information in the input arguments to allow differentiation of the tests. If the NCD-based metrics really breaks down when there is only little information available, needs to be explored more in future research.

Our VAT model is important since it allows us to create a whole family of different NCD-based test diversity measures, depending on which steps and to what level of detail information from a step is included in a trace. However, our focus has not been on the model and future work can also explore how the model can be refined and extended, and thus imply other important and useful test diversity metrics. It might also be possible to use NCD for search-based software engineering studies other than testing.

A threat to our study is that the results cannot be generalized to larger and more realistic programs. For example, it is not clear what would happen if the internal state contains very complex and large data structures. Maybe they will dwarf the relatively smaller parts of the trace that the statements make up in a way similar to how our test case with long input numbers seemed more similar than they would intuitively have been judged to be. Such trace dominance problems could potentially limit the usefulness of our approach for programs with complex data. On the other hand, not only data could dominate a trace, but also the code, for example in loops with many iterations.

The problem of trace dominance is important since it relates to what it is we really want to investigate the distance between. Human subjects may be more likely to focus on the black-box aspects of a test than to consider the implementation important. Or maybe simplified traces of the execution might be enough, e.g. only incorporating the calls to other methods, not the statements within them. We believe that the UTD itself might not be the one and only solution to measuring test distances, however by choosing different variation points and levels of details we can get different clusterings of tests that show different but important aspects of how the tests differ. Such a multi-metric approach might be the most long-term useful. Potentially it could also consider calculating distances between static information about the tests.

Another threat to our study is that the humans we used in our study are not typical of the general (software engineer) population. On this small example we do not think this is a serious threat. However, for larger programs we need to evaluate if different humans might differently value test differences and thus group the associated tests.

We note that the wording of 'cognitive diversity' might not be well chosen since there is already such a concept in psychology. However, we use the term in the same sense as it is used in [4] and we think there is little risk of confusion with the psychological term.

There are many issues that need to be further investigated. For example, it is not clear how we can select which information from the VAT model to include when applying

UTD. There is a risc that we have shifted the problem of calculating a test distance to one of selecting which information to include. And without any information selection some other part of the system searching for tests must select relevant from irrelevant information. Even though the NCD, or in particular the NID, has theoretically interesting properties and should be able to find all relevant differences we need more empirical research to evaluate this in practice. Also it is unclear if differences that are relevant from a information-theoretical standpoint is relevant for humans or for the actual quality of the developed software. While non-relevance for humans could be an advantage, the method could find test differences important that humans missed, non-relevance for software quality needs to be investigated in detail.

Special attention in future studies should be given to the issue of clustering humanly-written test goal specifications. One counter-argument to using a method based on NCD would be that words that are syntactically similar but semantically different might be co-clustered. This might invalidate the use of UTD for clustering of tests based on descriptions written by humans, not only in test goal descriptions but also in, for example, source code comments. There is some evidence that this would not be the case. For example, Cilibrasi, have clustered works of fiction and found that they cluster in a cognitively 'natural' way [6]. However, it might be the case that these clusters are based on ways of expression and choice of words etc. and not on any semantic differences.

The basic premise for our paper, that tests that appear diverse to humans are more effective in improving confidence in program correctness, also needs to be studied in more detail. For example, in program reliability arguments based on statistical usage testing the most heavily used modules are the ones that are the most heavily tested. Tests for these heavily tested modules might be more similar than tests that execute rarely used modules. Thus, we need to better understand different levels of scale and test diversity. Even though test diversity on module level might be important it is not clear that this is true for the system level. It also points to the fact that UTD-based test search is not a silver bullet; it can be but one component in a battery of testing tools that we need to address pressing software quality challenges. Investigating how it fares compared to and in combination with other test creation approaches is thus important.

8. Related Work

Bueno also discusses different approaches to diversity-based test set optimization and introduces structural and input domain perspectives on diversity [5]. Our framework extends this by also considering output domain diversity, including both changes to the state and the return values. We also emphasize that the input to the software under test is not simply the data used in the invocation but also any test code and data used in setting it up for the test. Furthermore our method and evaluation is more general since we consider complex, and not only numerical, test data. The metric we propose to use, as a general solution for diversity metrics, can handle any type of data and structure, as long as it is dumped to a linear string.

Ciupa et al. introduces a measure of object distance by defining an elementary distance between atomic object types (strings, numbers etc.) and then basing the object distance on its type and values of its fields and by recursively applying this on sub-objects in the fields [9]. It is based on the values and not on the structure of how these values are connected [9]. This is in contrast to our NCD-based test metrics, which can take both structure and values into account, if we can find a way to represent them both in a linear string.

Several test diversity measures for control and data flow coverage were defined by Nikolik [19]. They are all based on tracing and collecting information on how many times each branch in the flow graph is executed when running a test. The diversity is then calculated by a formula that relates how skewed the distribution over the branches is compared to a uniform distribution. Thus they measure how focused the test is on a specific part of the SUT. Our proposed metric is related to this since the statements executed in the chosen branch would all be listed in our traces, and listed multiple times if the branch is executed multiple times. Thus, in an indirect way, similar information as the one collected by Nikolik would be available in our trace. However, the actual calculations involved are quite different and our proposed metrics is motivated by the theory of Information Distance. Furthermore, our traces contain much other information that is not collected by Nikolik.

Baudry et al. have introduced a relative fitness measure [2] where the fitness of one test case is related to that of a set of other test cases. However, the base fitness functions used to calculate the relative fitness judges the fitness of a whole set of test cases together while the metrics we propose in this paper applies to pairs of individual test cases.

Our approach has similarities to the clustering of execution profiles that Dickinson and Leon used to select a subset of a set of potential test cases to evaluate for conformance to requirements [10]. However, their dissimilarity metrics was based on euclidian distance and the execution profiles were based on caller/callee counts. It is not clear how their method could be expanded to include more complex profile information like the one supported by NCD (any information).

9. Conclusion

We have proposed a family of test diversity metrics based on a theoretically optimal, even universal, cognitive similarity measure. The metrics are easy to implement and are flexible in the sense that they can adapt to the information about the tests that is chosen or is available. They require little thought on the part of its users.

In an initial experiment we evaluated one of these metrics and compared it to three human subjects on the task of clustering 25 test cases for the triangle classification problem. The metric found many of the same and intuitive clusterings that were uncovered by the humans. There were a few exceptions which could be explained by the fact that the automated clustering was based on not only the black-box but white-box information. The human subjects only clustered the test cases based on black-box information.

The proposed metrics can be used in search-based software testing to search for tests that are cognitively different from each other. They might also be more generally useful in search-based as well as normal software engineering. However, based on this study it is not clear if the proposed metrics will scale to real-world software systems. More evaluation is needed and should be the topic of future research.

10. Acknowledgements

We thank the participants in this study for their valuable time. We would also thank the anonymous reviewers, whose comments helped increase the quality of this paper. This research is supported by the Knowledge Foundation in Sweden (under a research grant for the project BESQ).

References

[1] D. Astels, S. Baker, D. Chelimsky, A. Helesøy, D. North, and B. Takita. RSpec. `http://rspec.info`, January 2008.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. Automatic Test Case Optimization: A Bacteriologic Algorithm. *IEEE Software*, 22(2):76–82, 2005.

[3] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. From Genetic to Bacteriological Algorithms for Mutation-Based Testing. *Software Testing, Verification & Reliability*, 15(2):73–96, 2005.

[4] C. H. Bennett, P. Gács, M. Li, P. M. B. Vitányi, and W. H. Zurek. Information Distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[5] P. M. S. Bueno, W. E. Wong, and M. Jino. Improving Random Test Sets Using the Diversity Oriented Test Data Generation. In *RT '07: Proceedings of the 2nd International Workshop on Random Testing*, pages 10–17, New York, NY, USA, 2007. ACM.

[6] R. Cilibrasi. *Statistical Inference Through Data Compression*. PhD thesis, Universiteit van Amsterdam, Amsterdam, Holland, February 2007.

[7] R. Cilibrasi, A. L. Cruz, S. de Roij, and M. Keijzer. CompLearn. `http://www.complearn.org/`, January 2008.

[8] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Object Distance and Its Application to Adaptive Random Testing of Object-Oriented Programs. In *RT '06: Proceedings of the 1st International Workshop on Random Testing*, pages 55–63, New York, NY, USA, 2006. ACM.

[10] W. Dickinson, D. Leon, and A. Podgurski. Finding Failures by Cluster Analysis of Execution Profiles. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, Washington, DC, USA, 2001. IEEE Computer Society.

[11] E. W. Dijkstra. Chapter I: Notes on Structured Programming. pages 1–82, 1972.

[12] R. Feldt. An Interactive Software Development Workbench Based on Biomimetic Algorithms. Technical Report 02-16, Dept. of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002.

[13] M. Harman. The Current State and Future of Search Based Software Engineering. In *FOSE '07: Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[14] K. Inkumsah and T. Xie. Evacon: A Framework for Integrating Evolutionary and Concolic Testing for Object-Oriented Programs. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 425–428, New York, NY, USA, 2007. ACM.

[15] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1997.

[16] P. McMinn. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[17] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[18] C. Neukirchen. Bacon. `http://chneukirchen.org/repos/bacon/README`, January 2008.

[19] B. Nikolik. Test Diversity. *Information & Software Technology*, 48(11):1083–1094, 2006.

[20] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[21] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 73–81, New York, NY, USA, 1998. ACM.

[22] A. Windisch, S. Wappler, and J. Wegener. Applying Particle Swarm Optimization to Software Testing. In *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 1121–1128, New York, NY, USA, 2007. ACM.