Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?

Article · September 2009

DOI: 10.1109/TAICPART.2009.15

3 authors, including:

Mark Harman
University College London
437 PUBLICATIONS **12,786** CITATIONS

SEE PROFILE

Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?

Kiran Lakhotia
King's College London
CREST centre
Strand, London
WC2R 2LS, UK

Phil McMinn
University of Sheffield
Regent Court
211 Portobello, Sheffield
S1 4DP, UK

Mark Harman
King's College London
CREST centre
Strand, London
WC2R 2LS, UK

***Abstract*—Whilst there is much evidence that both concolic and search based testing can outperform random testing, there has been little work demonstrating the effectiveness of either technique with complete real world software applications. As a consequence, many researchers have doubts not only about the scalability of both approaches but also their applicability to production code. This paper performs an empirical study applying a concolic tool, CUTE, and a search based tool, AUSTIN, to the source code of four large open source applications. Each tool is applied 'out of the box'; that is without writing additional code for special handling of any of the individual subjects, or by tuning the tools' parameters. Perhaps surprisingly, the results show that both tools can only obtain at best a modest level of code coverage. Several challenges remain for improving automated test data generators in order to achieve higher levels of code coverage.**

I. Introduction

Software testing can be thought of as a sequence of three fundamental steps:

1. The design of test cases that are good at revealing faults, or which are at least adequate according to some test adequacy criterion.
2. The execution of these test cases.
3. The determination of whether the output produced is correct.

Sadly, in current testing practice, often the only fully automated aspect of this activity is test case execution. The problem of determining whether the output produced by the program under test is correct cannot be automated without an oracle, which is seldom available. Fortunately, the problem of generating test data to achieve widely used notions of test adequacy *is* an inherently automatable activity.

Such automation promises to have a significant impact on testing, because test data generation is such a time-consuming and laborious task. This promised impact has attracted much attention, leading to several schools of thought as to how best automate the test data generation process. This paper concerns two widely studied schools of thought: concolic testing and search based testing, both of which have been the subject of much recent attention.

Concolic testing [31], [12], [5] originates in the seminal work of Godefroid *et al.* on Directed Random Testing [12]. It formulates the test data generation problem as one of finding a solution to a constraint satisfaction problem, the constraints of which are produced by concolic execution of the program under test. Concolic execution combines symbolic [18] and concrete execution. Concrete execution drives the symbolic exploration of a program, and dynamic variable values obtained by real program execution can be used to simplify path constraints produced by symbolic execution.

Search based testing [23] formulates the test data adequacy criteria as objective functions, which can be optimized using Search Based Software Engineering [7], [14]. The search space is the space of possible inputs to the program under test. The objective function captures the particular test adequacy criterion of interest. The approach has been applied to several types of testing, including functional [3] and non-functional [37] testing, mutation testing [1], regression testing [39], test case prioritization [21], [35], and interaction testing [8]. However, the most studied form of search based testing has been structural test data generation [27], [19], [29], [26], [36], [33], [15].

Whilst many papers argue that both concolic and search based techniques are better than random testing [31], [36], [15], there has been little work investigating and comparing their effectiveness with real world software applications. Previous work has tended to be small-scale, considering only a couple of programs, or considering only parts of individual applications to which the test generation technique is known in advance to be applicable. Furthermore, the test data generators themselves have tended not to apply test data generation tools 'out of the box'; i.e. without customization, special handling for different test subjects, or parameter tuning. This leaves the literature without convincing answers to several important questions, including:

- How effective are concolic and search based tools when applied to real world software applications?
- How long does it take for concolic and search based tools to achieve certain levels of coverage?

The aim of this paper is to provide answers to these questions. In order for automated test data generation approaches to achieve their full potential it is necessary for them to be evaluated using realistic non–trivial programs without any 'special' (i.e. human) intervention to smooth over the

difficult 'real world' challenges that might be encountered. An empirical study is performed which compares a concolic tool, CUTE [31], and a search based tool, AUSTIN [20]. The test adequacy criterion under investigation is branch coverage. The primary contributions of this paper are the following:

1. An empirical study which determines the level of code coverage that can be obtained using CUTE and AUSTIN on the complete source code of four open source programs. Perhaps surprisingly, the results show that only modest levels of coverage are possible at best, and there is still much work to be done to improve test data generators.
2. An empirical study investigating the wall clock time required for each tool to obtain the coverage that it does, and thus an indication of the efficiency of each approach.

The rest of the paper is organized as follows. Section II provides the background information to the concolic and search based tools, CUTE and AUSTIN, which the empirical study presented in this paper uses. Section III outlines the motivation for our work, the research questions addressed, and the gap in the current literature this paper is trying to close. The empirical study, results and answers to the research questions are presented in Section IV, whilst threats to validity are addressed in Section V. Section VI presents related work, and Section VII concludes.

II. Background

Automated structural test data generation has been a burgeoning interest to researchers since at least the 1970s. In this decade two approaches to the problem emerged - symbolic execution [18], which is the basis of concolic testing; and a method that reformulated the problem of executing a path through a program with floating-point inputs into objective functions [27], which later developed into the field known as search based testing.

A. Concolic Testing

Concolic testing builds on the ideas of symbolic execution. For a given path through a program, symbolic execution involves constructing a *path condition*; a system of constraints in terms of the input variables that describe when the path will be executed. For example, the path condition which executes the `if` statement as true in Figure 1a would simply be $a_0 + 5 = b_0 - 10$, where a_0 and b_0 refer to the symbolic values of the input variables `a` and `b` respectively. To cover the true branch, the path condition is then solved by a constraint solver in order to derive concrete input values.

The path condition can easily become unsolvable, however, if it contains expressions that cannot be handled by constraint solvers. This is often the case with floating-point variables, or non-linear constraints. For example, a linear constraint solver would encounter difficulties with the program of Figure 1b because of the non-linear predicate appearing in the `if` condition.

Concolic testing can alleviate some of the problems of non-linearity by combining *con*crete execution with symb*olic* generation of path conditions. The idea is to simplify the path condition by substituting sub-expressions with concrete values, obtained by actual dynamic executions. This substitution process can remove some of the non-linear sub-expressions in a path condition making them amenable to a constraint solver. Concolic execution originated from the work of Godefroid *et al.* [12]. The term *concolic* was coined by Sen *et al.* [31] in their work introducing the CUTE tool, which is based upon similar principles.

The CUTE Tool: Suppose execution of the path which executes the true branch of the program of Figure 1b is required. CUTE executes the program with some input. The default is to execute a function with all variables of primitive type set to zero, although random values can be used instead. Suppose the function is executed with the random values `536` and `156` for `x` and `y` respectively. The path taking the false branch is executed. The path condition is $x_0 * y_0 < 100$. Since this constraint is non-linear, CUTE will replace x_0 with its concrete value, 536. The path condition becomes $536 * y_0 < 100$, which is now linear and can be passed to the constraint solver to find an appropriate value for `y` (*i.e.* zero or any value that is negative).

CUTE attempts to execute all feasible program paths, using a depth-first strategy. The first path executed is that, which is traversed with all zero or random inputs as described above. The next path to be attempted is the previous path, but taking the alternative branch at the last decision statement executed in the path. The new path condition is therefore the same as the previous path condition, but with the last constraint negated, allowing for substitution of sub-expressions in the new path condition with sensible concrete values (as in the example above). For programs with unbounded loops, CUTE may keep unfolding the body of the loop infinitely many times, as there may be an infinite number of paths. The CUTE tool is therefore equipped with a parameter which places a limit on the depth of the depth-first path unfolding strategy.

B. Search Based Testing

Like symbolic–execution–based testing, the first suggestion of optimization as a test data generation technique also emerged in the 1970s, with the seminal work of Miller and Spooner [27]. Miller and Spooner showed that the series of conditions that must be satisfied for a path to be executed can be reformulated as an *objective function*, the optima of which (*i.e.* the test data that executes the path) could be found using optimization techniques.

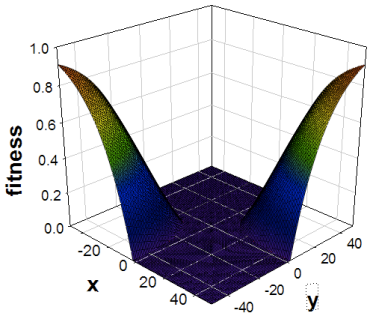
The role of an objective function is to return a value that indicates how 'good' a point in a search space (*i.e.* an input vector) is compared to the best point (*i.e.* the required test data); the global optimum. For example, if a program condition `a == b` must be executed as true, the objective function could be $|a - b|$. The closer the output of this formula is to zero, the 'closer' the program input is to making `a` and `b` equal, and the closer the search technique is to finding the test data of interest.

```
void testme1(int a, int b)
{
    a += 5; b -= 10;
    if (a == b)
        // ...
}
```

(a) *Example for demonstrating symbolic execution. The branch predicate is linear*

```
void testme2(int x, int y)
{
    if (x * y < 100)
        // ...
}
```

(b) *Example for demonstrating concolic execution and search based testing. The branch predicate is non-linear*



(c) *Surface of the objective function for executing the true branch of the program in (b)*

```
void testme3(int a, int b, int c)
{
(1)     if (a == b)
(2)         if (b == c)
(3)             if (a == c)
(4)                 // ...
}
```

(d) *Example for demonstrating objective function calculation for the AUSTIN tool*

Fig. 1. Examples for demonstrating symbolic, concolic and search based testing

Because an optimizing search technique is used rather than a constraint solver, non-linear constraints present fewer problems. For example the surface of the objective function for taking the true path through the `if` condition of Figure 1b can be seen in Figure 1c. The surface is smooth and provides the optimization process with a clear 'direction', guiding the search to the required test data. Furthermore, computation of the objective function by dynamically executing the program alleviates another problem of both symbolic and concolic testing, *i.e* floating-point inputs.

The suggestions of Miller and Spooner were not subsequently taken up until Korel developed them further in 1990 [19], when he proposed the use of a search technique known as the 'alternating variable method'. Since then the ideas have been applied to other forms of testing [3], [37], [1], [39], [21], [35], [8], using a variety of optimizing search techniques, including genetic algorithms [29], [26], [36]. The objective function has been further developed to generate test data for a variety of program structures, including branches, as well as paths [36].

The AUSTIN Tool: AUSTIN is a tool for generating branch adequate test data for C programs. AUSTIN does not attempt to execute specific paths in order to cover a target branch; the path taken up to a branch is an emergent property of the search process. The objective function used by AUSTIN was introduced by Wegener *et al.* [36] for the Daimler Evolutionary Testing System. It evaluates an input against a target branch using two metrics; the *approach level* and the *branch distance*. The approach level records how many nodes on which the branch is control dependent, were not executed by a particular input. The fewer control dependent nodes executed, the 'further away' the input is from executing the branch in control flow terms. Thus, for executing the true branch of statement 3 in Figure 1d; the approach level is

- 2 when an input executes the false branch of statement 1;
- 1, when the true branch of statement 1 is executed followed by the false branch of statement 2;
- zero if statement 3 is reached.

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current 'target' branch. Taking the true branch from statement 3 as an example again, if the false branch is taken at statement 1, the branch distance is computed using $|a - b|$, whilst $|b - c|$ is optimized if statement 2 is reached but executed as false, and so on. The branch distance is normalized and added to the approach level.

The search method used is the Alternating Variable Method (AVM), proposed by Korel [19]. The AVM is a simple search technique, which was shown to be very effective by Harman and McMinn [15] when compared with more sophisticated optimization techniques such as genetic algorithms.

AUSTIN, like CUTE, begins with all primitives set to zero. If the target is not executed, the AVM cycles through each input of primitive type and performs so–called 'pattern moves', guided by the objective function. If a complete cycle of adjustments takes place with no improvement in objective value, the search restarts using random values.

Suppose the program of Figure 1d is executed with the input $< a = 100, b = 200, c = 300 >$, with the aim of executing the true branch of statement 3. AVM takes the first variable, `a`, and performs *exploratory moves*; executions of the program where `a` is decreased and increased by a small amount δ ($\delta = 1$ for integers and 0.1 for floating point variables). An increased value of the variable `a` brings it closer to `b` and results in a better objective value.

AVM then makes *pattern moves* for as long as the objective function continues to yield an improved value. The value added to the variable in the nth pattern move is computed using the formula $2^n \cdot dir \cdot \delta$; where $dir \in \{-1, 1\}$ corresponding to the positive or negative 'direction' of improvement, identified by the initial exploratory moves. Thus consecutive exploratory moves for the variable `a` are 102, 104, 108 and so on. Pattern moves will improve the objective value until a value of 228 is reached for the variable `a`. At this point the minimum ($a = 200$) has been overshot, so the AVM repeats the exploratory-pattern move cycle for as long as necessary until the minimum is reached. When $a = 200$, the true branch of statement 1 is executed. For executing statement 2, exploratory moves on the variable `a` both lead to a worse objective value, because the original outcome at statement 1 is affected and the approach level worsens. The AVM will then consider the next variable, `b`. Exploratory moves here have the same effect, and so the AVM moves onto the variable `c`. Decreasing the value of `c` improves the objective value, and so pattern moves are made. Eventually each input value is optimized to 200.

Should exploratory moves produce no improvement in objective value, with the required test data not found either, the search has hit a local minima from which it cannot proceed. AVM terminates and restarts with a new random input vector. Typically, the search is afforded a 'budget' of objective function evaluations (*i.e.* program executions) in which to find test data, otherwise the search is deemed to have failed. This could be because the branch is infeasible. In some cases, however, the objective function surface can be flat, offering the search no guidance to the required test data. This is sometimes due to the presence of boolean 'flag' variables [2], [25], which can result in two plateaus for a branch condition; one where the flag is true, and one where it is false.

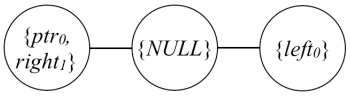
C. Handling Inputs Involving Pointers

The above descriptions explained how concolic testing and search based testing handle inputs of a primitive type only. This section explains how CUTE and AUSTIN handle pointer inputs. The CIL (C Intermediate Language) infrastructure [28] is used to transform and simplify programs such that all predicates appearing in decision statements contain only either pointer types or arithmetic types. In addition, the source contains no compound predicates. Thus all conditions involving pointers are of the form `x == y` or `x != y` (where `x` and `y` could also represent $NULL$).

Pointers in CUTE: The first path explored by the CUTE tool is the path executed where inputs of primitive type are zero (or of a random value, depending on the settings used). If the function involves pointer variables, these are always initially set to $NULL$. However, further paths through the program may require pointers to point to an actual data structure instead. In order to find the 'shape' of this data structure, CUTE incorporates symbolic variables for pointers in the path condition. A graph-based process is used to check that the constraints over the pointer variables are feasible, and finally, a simple procedure is used to actually build the data structure required.

```
        void testme4(item* ptr)
        {
   (1)      if (ptr != NULL)
   (2)          if (ptr->left != NULL)
   (3)              if (ptr->left->right == ptr)
   (4)                  // ...
        }
```

(a) *Code snippet*



(b) *CUTE feasibility graph for path which executes all decisions in (a) as true*

Fig. 2. Example for demonstrating pointer handling in CUTE and AUSTIN

For the program of Figure 2a, and the path that executes the true branch at each decision, CUTE accumulates the path constraint:

$$ptr_0 \neq NULL \wedge left_0 \neq NULL \wedge right_1 = ptr_0$$

CUTE keeps a map of which symbolic variable corresponds to which point in the data structure, for example, $left_0$ maps to `ptr->left`. The feasibility check involves the construction of an undirected graph, which is built incrementally at the same time as the path condition is constructed from the conditions appearing in the program. The nodes of the graph represent abstract pointer locations, with node labels representing the set of pointers which point to those locations. A special node is initially created to represent $NULL$. Edges between nodes represent inequalities. After statement 1 in the example, the graph consists of a node for ptr_0 with an edge leading from it to the $NULL$ node. When statement 2 is encountered, a new node is constructed for $left_0$, with an edge to $NULL$. Finally, $right_1$ is merged into the existing ptr_0 node, as they must point to the same location (Figure 2b). Feasibility is checked as each constraint is added for each decision statement. An equality constraint between two pointers `x` and `y` is feasible if and only if there is no edge in the graph between nodes representing the locations of `x` and `y`. An inequality constraint between `x` and `y` is feasible if and only if the locations of `x` and `y` are not represented by the same node.

If the path condition is feasible, the data structure is built incrementally. Each new branching decision adds a new constraint to the path condition, and the data structure is created on the basis of each constraint using the rules of Table I. A more detailed treatment can be found in the work of Sen *et al.*[31].

Pointers in AUSTIN: There has been little work with respect to generating dynamic data structures in search based testing. Korel [19] developed a limited method for simple Pascal data structures. In order to apply search based testing to real

TABLE I
DYNAMIC DATA STRUCTURE CREATION ACCORDING TO INDIVIDUAL CONSTRAINTS ENCOUNTERED ALONG THE PATH CONDITION FOR CUTE AND AUSTIN

Constraint	CUTE	AUSTIN
$m_0 = NULL$	Assign $NULL$ to m_0	
$m_0 \neq NULL$	Allocate a new memory location pointed to by m_0	
$m_0 = m_1$	Make m_1 alias m_0	
$m_0 \neq m_1$	Allocate a new memory location pointed to by m_1	With an even probability, assign $NULL$ or allocate a new memory location pointed to by m_0

world programs this limitation had to be overcome. AUSTIN uses search based techniques for primitive inputs, a symbolic process akin to that of CUTE is used for pointers[20]. As with CUTE, pointer inputs are initially set to $NULL$. During the search process, a branch distance calculation may be required for a condition that involves a pointer comparison. However branch distances over physical pointer addresses do not usually give rise to useful information for test data generation; for example it is difficult to infer the shape of a dynamic data structure. Therefore, instead of computing a branch distance, AUSTIN performs symbolic evaluation for the path taken up to the predicate as executed by the current input generated by the AVM. The result is a path condition of the same form as generated by CUTE. As with CUTE, the constraints added to the path condition are used to incrementally build the dynamic data structure. The rules for doing this appear in Table I. AUSTIN does not perform a feasibility check; if the branching condition at the control dependent node is not executed as required, the AVM process restarts afresh. For a more in-depth treatment, see reference [20].

III. MOTIVATION AND RESEARCH QUESTIONS

One of the first tests for any automatic test data generation technique is that it outperforms random testing. Many authors have demonstrated that both concolic based and search based techniques can outperform purely random test data generation. However, there are fewer studies that have attempted to evaluate concolic and search based approaches on real world programs.

Previous studies have tended to be small-scale [19], [36] or, at least in the case of search based approaches, concentrated on small 'laboratory' programs. Where production code has been considered, work has concentrated solely on libraries [33] or individual units of applications [15]; usually with the intention of demonstrating improvements or differences between variants of the techniques themselves.

Studies involving concolic approaches have also tended to focus on illustrative examples [31], [38], [9], [5], with relatively little work considering large scale real world programs such as the `vim` text editor [4], [22], network protocols [10] or windows applications [11]. Furthermore, no studies have compared the performance of concolic and search based testing on real world applications.

The research questions to be answered by the empirical study are therefore as follows:

RQ 1: How effective are concolic and search based test data generation for real world programs? Given a set of real world programs, how good are concolic and search based test data generators at achieving structural coverage?

RQ 2: What is the relative efficiency of each individual approach? If it turns out that both approaches are more or less equally effective at generating test data, efficiency will be the next issue of importance as far as a practitioner is concerned. If one approach is more effective but less efficient than the other, what is the trade off that the practitioner has to consider?

IV. EMPIRICAL STUDY

The empirical study was performed on a total of 87,589 pre–processed lines of C code contained within four open-source programs. This is the largest study of search based testing by an order of magnitude and is similar in size to the largest previous study of any form of concolic testing.

A. Test subjects

Details of the subjects of the empirical study are recorded in Table II. A total of 387 functions were tested. Since the study is concerned with branch coverage, trivial functions not containing any branches were ignored. In addition, further functions had to be omitted from the study, because they could not be handled by CUTE or AUSTIN. These included functions whose inputs were files, data structures involving function or `void` pointers, or had variable argument lists.

The programs chosen are not trivial for automated test data generation. `libogg` is a library used by various multimedia tools and contains functions to convert to and from the *Ogg* multimedia container format, taking a bitstream as input. `plot2d` is a relatively small program which produces scatter plots directly to a compressed image file. The core of the program is written in ANSI C, however the entire application includes C++ code. Only the C part of the program was considered during testing because the tools handle only C. `time` is a GNU command line utility which takes, as input, another process (a program) with its corresponding arguments and returns information about the resources used by a specific program, including wall-clock and CPU times. `zile` is a text editor in Unix, and makes heavy use of string operations. It was designed to be a more lightweight version of Emacs.

B. Experimental setup

Each function of each test subject was taken in turn (hereinafter referred to as the 'FUT' – Function Under Test), with the aim of recording the level of coverage that could be achieved by each tool.

TABLE II
DETAILS OF THE TEST SUBJECTS USED IN THE EMPIRICAL STUDY. IN THE 'FUNCTIONS' COLUMN, 'NON-TRIVIAL' REFERS TO FUNCTIONS THAT CONTAIN BRANCHING STATEMENTS. 'TOP-LEVEL' IS THE NUMBER OF NON-TRIVIAL, PUBLIC FUNCTIONS THAT A TEST DRIVER WAS WRITTEN FOR, WHILST 'TESTED' IS THE NUMBER OF FUNCTIONS THAT WERE TESTABLE BY THE TOOLS (I.E. TOP-LEVEL FUNCTIONS AND THOSE THAT COULD BE REACHED INTERPROCEDURALLY). IN THE 'BRANCHES' COLUMN, 'TESTED' IS THE NUMBER OF BRANCHES CONTAINED WITHIN THE TESTED FUNCTIONS

Test	**Lines**	**Functions**				**Branches**	
Object	**of Code**	Total	Non-Trivial	Top Level	Tested	Total	Tested
`libogg`	2,552	68	33	32	33	290	284
`plot2d`	6,062	35	35	35	35	1,524	1,522
`time`	5,503	12	10	8	10	202	198
`zile`	73,472	446	339	312	340	3,630	3,348
Total	87,589	561	417	387	418	5,646	5,352

Since CUTE and AUSTIN take different approaches to test data generation, care had to be taken in setting up the experiments such that the results were not inadvertently biased in favour of one of the tools. The main challenge was identifying suitable stopping criteria that were 'fair' to both tools. Both tools place limits on the number of times the function under test can be called, yet this is set on a per-function basis for CUTE and a per-branch basis for AUSTIN. Furthermore, one would expect CUTE to call the function under test less often than AUSTIN, because it carries out symbolic evaluation. Thus, setting a limit that was 'equal' for both tools was not feasible. Therefore it was decided that each limit would be set to a high value, with a time limit of 2 minutes of wall clock time per FUT used as an additional means of deciding when a tool's test data generation process should be terminated.

CUTE's limit was set to the number of branches in the FUT multiplied by 10,000. CUTE can reach this limit in only two cases; firstly if it keeps unfolding a loop structure, in which case it won't cover any new branches; or secondly if the limit is less than the number of interprocedural branches (which was not the case for any of the test subjects considered). AUSTIN's limit was set to 10,000 FUT executions per branch, with branches in the FUT attempted sequentially in reverse order. Oftentimes the search process did not exhaust this limit but was terminated by the overall time limit instead.

Thirty 'trials' were performed for each tool and each function of each test subject. AUSTIN is stochastic in nature, using random points to restart the search strategy once the initial search, starting with all primitives as zero, fails. Thus, several runs need to be performed to sample its behaviour. Since some test subjects exhibit subtle variations in behaviour over different trials (*e.g.* in the `time` program), CUTE was also executed thirty times for each function, so that AUSTIN did not benefit unfairly from multiple executions.

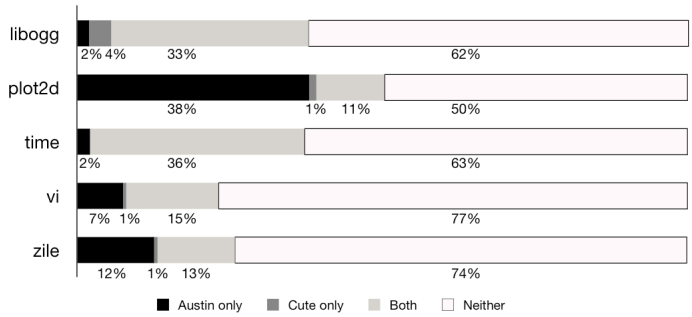
Coverage was measured in two ways. The first is respective to the branches covered in the FUT only. A branch is counted as covered if it is part of the FUT, and is executed at least once during the thirty trials. The second measure takes an interprocedural view. A branch is counted as covered if it is part of the FUT or any function reachable through the FUT. Interprocedural coverage is important for CUTE, since path conditions are computed in an interprocedural fashion. Any branches covered interprocedurally by AUSTIN, however, are done so serendipitously, as the tool only explicitly targets branches in the FUT.

Apart from the settings necessary for a fair comparison, as discussed above, both tools were applied 'out of the box', *i.e.* with default parameters and without the writing of special test drivers for any of the test subjects. As mentioned in Section II-A, CUTE has an option to limit the level of its depth-first search, thus preventing an infinite unfolding of certain loops. However, as it is not generally known, *a priori*, what a reasonable restriction is, CUTE was used in its default mode with no specified limit, *i.e.* an unbounded search.

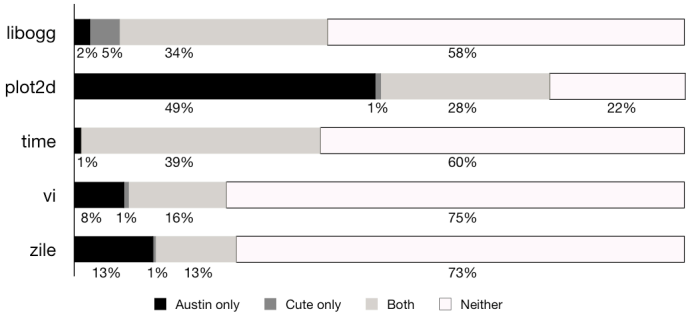
The test driver for both tools is not only responsible for initializing input parameters, but also the place to specify any pre–conditions for the function under test. AUSTIN generates a test driver automatically by examining the signature of the FUT. The test drivers for CUTE had to be written manually but were constructed using the same algorithm as AUSTIN. Writing pre–conditions for functions without access to any specifications is non–trivial. For the study only the source code was available with no other documentation. Therefore it was decided the only pre–condition to use was to require top level pointers to be non-*NULL* (as described in Section II-C).

C. Answers to Research Questions

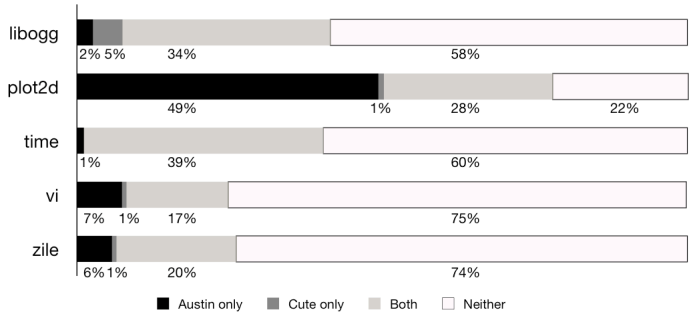
RQ 1: How effective are concolic and search based test data generation for real world programs? Figure 3 plots three different 'views' of the coverage levels obtained by CUTE and AUSTIN with the test subjects. The first view, Figure 3a, presents coverage of branches in the FUT only. However, CUTE places an emphasis on branches covered in functions called by the FUT, building up path conditions interprocedurally. For AUSTIN interprocedural branch coverage is incidental, with test generation directed at the FUT only. Therefore, Figure 3b plots interprocedural coverage data which, in theory, should be favourable to CUTE. Finally, CUTE could not attempt 81 functions, containing 1,148 branches. Some of these functions involved `void` pointers, which cannot be handled by CUTE. However, a number of functions could not be tested by CUTE because the test subject



(a) *Branches covered only as part of the function under test*



(b) *Branches covered in the function under test and interprocedurally*



(c) *Branches covered in functions that CUTE can handle only*

Fig. 3. Branch coverage for the test subjects with CUTE and AUSTIN. CUTE explicitly explores functions called by the function under test, whereas AUSTIN does not. Therefore the graph (a) counts only branches covered in each function tested individually. Graph (b) counts branches covered in the function under test and branches covered in any functions called. Graph (c) is graph (b) but with certain functions that CUTE cannot handle excluded

TABLE III
COMPARING WALL CLOCK TIME AND INTERPROCEDURAL BRANCH COVERAGE FOR A SAMPLE OF BRANCHES

Test Object	Function	Branches		CUTE			AUSTIN		
				Time	Branches covered		Time	Branches covered	
		Function Under Test	(Inter-procedural)	(s)	Function Under Test	(Inter-procedural)	(s)	Function Under Test	(Inter-procedural)
libogg	ogg_stream_clear	8	(8)	0.84	7	(10)	134.75	5	(7)
	oggpack_read	14	(14)	0.24	2	(2)	0.18	2	(2)
plot2d	CPLOT_BYTE_MTX_Fill	4	(8)	131.25	4	(4)	130.05	1	(1)
	CPLOT_DrawDashedLine	56	(56)	130.43	13	(13)	130.4	37	(37)
	CPLOT_DrawPoint	16	(16)	0.51	13	(13)	131.82	13	(13)
time	resuse_end	6	(6)	0.42	4	(4)	131.84	5	(5)
zile	astr_rfind_cstr	6	(6)	0.45	2	(2)	0.17	2	(2)
	check_case	6	(6)	0.38	1	(1)	130.49	6	(6)
	expand_path	82	(84)	0.37	0	(1)	0.16	0	(1)
	find_window	20	(20)	0.4	1	(1)	131.4	1	(1)
	line_beginning_position	8	(8)	0.27	0	(0)	0.18	0	(0)
	setcase_word	40	(74)	0.31	0	(0)	0.18	0	(0)
Total		266	306						

did not compile after CUTE's instrumentation. For certain functions of the `zile` test subject, the instrumentation casts a data structure to an unsigned integer, and subsequently tries to dereference a member of the data structure, which results in an error. Since CUTE's exploration behaviour is interprocedural, all functions within this source file became untestable. Thus, Figure 3c plots interprocedural branch coverage, but removing these branches from consideration.

Strikingly, all three views of the coverage data show that in most cases, the majority of branches for an application were not covered by *either* tool. The only exception is the `plot2d` test subject. Here, AUSTIN managed 77% coverage taking interprocedural branches into account, compared to CUTE's 50%. Code inspection revealed that 13 functions in `plot2d` contained unbounded loops. CUTE therefore never attempted to cover any more branches preceding the body of the loop (both intraprocedural and interprocedural) and instead kept increasing the number of loop iterations by one until its timeout or iteration limit was reached. For all other subjects, coverage for either tool does not exceed 50% whatever 'view' of the data is considered. It has to be noted, however, that AUSTIN does seem to cover a higher number of the branches.

When a modified path condition falls outside the supported theory of CUTE's constraint solver, unlike AUSTIN, CUTE does not try a fixed number of random 'guesses' in order to find test data. AUSTIN on the other hand will spend 10,000 attempts at covering the branch. In the worst case this is equal to performing a random search. Nevertheless, it has a higher chance of finding test data than CUTE, simply because it spends more effort per branch.

To conclude, these data suggest that the concolic and search based approaches are not as effective for real world programs as researchers may have previously been led to believe by previous smaller–scale studies. Next the efficiency of each of the tools is examined with respect to a subset of the branches.

RQ 2: What is the relative efficiency of both approaches? In order to answer this research question, a random sample of 12 functions were taken and the performance of each individual tool analysed further. These functions are listed in Table III and comprise 266 branches, with a further 40 reachable interprocedurally.

CUTE times out (reaching the 120 second limit) on two occasions. This is because CUTE gets stuck unfolding loops in called functions. AUSTIN times out on five occasions. For example, the function `ogg_stream_clear` from `libogg` takes as input a pointer to a data structure containing 18 members, one of which is an array of 282 unsigned characters, while 3 more are pointers to primitive types. Since AUSTIN does not use any input domain reduction, it has to continuously cycle through a large input vector in order to establish a direction for the search so it can start applying its pattern moves. Secondly, unbounded loops cause problems for AUSTIN too with respect to the imposed timeout. Whenever AUSTIN performs a random restart, it has a high chance of increasing the number of loop iterations by assigning a large value to the termination criterion, thus slowing down execution time of the function under test.

The table does not reveal a prevailing pattern that would allow us to simply conclude that 'CUTE is more efficient' or vice versa. The results are very much dependent on the function under consideration. However, unless the tool gets 'trapped' (*e.g.* CUTE continually unfolding a loop), each tool can be expected to terminate within a second, which is an entirely practical amount of time.

V. Threats to Validity

Any attempt to compare two different approaches faces a number of challenges. It is important to ensure that the comparison is as fair as possible. Furthermore, the study presented here, as we are comparing two widely studied approaches to test data generation, also seeks to explore how well these approaches apply to real world code. Naturally, this raises a number of threats to the validity of the findings, which are briefly discussed in this section.

The first issue to address is that of the *internal validity* of the experiments, *i.e.* whether there has been a bias in the experimental design that could affect the obtained results. One potential source of bias comes from the settings used for each tool in the experiments, and the possibility that the setup could have favoured or harmed the performance of one or both tools. In order to address this, default settings were used where possible. Where there was no obvious default (*e.g.* termination criteria), care was taken to ensure that reasonable values were used, and that they allowed a sensible comparison between performance of both tools.

Another potential source of bias comes from the inherent stochastic behaviour of the metaheuristic search algorithm used in AUSTIN. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments were repeated 30 times. Due to the stochastic nature of some of the test subjects in the study, experiments were also repeated 30 times for CUTE, so as not to bias the results in favour of the AUSTIN tool.

A further source of bias includes the selection of the programs used in the empirical study, which could potentially affect its *external validity*; *i.e.* the extent to which it is possible to generalise from the results obtained. The rich and diverse nature of programs makes it impossible to sample a sufficiently large set of programs such that all the characteristics of all possible programs could be captured. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real world open source programs. It should also be noted that the empirical study drew on over 561 functions comprising of over 5,646 branches, providing a large pool of results from which to make observations.

The data were collected and analysed in three different ways; taking into account coverage in the FUT only, interprocedural coverage and removing functions that CUTE could not handle from the sample. No matter which analysis was conducted, the results always showed a consistently poor level of coverage. Nevertheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all empirical experiments in software engineering, further experiments are required in order to replicate the results here.

VI. Related Work

There have been several tools developed using Directed Random Testing. The first tool was developed by Godefroid *et al.* [12] during their work on directed random testing and the DART tool. Unlike CUTE, DART does not attempt to solve constraints involving memory locations. Instead, pointers are randomly initialized to either $NULL$ or a new memory location. DART does not transform non linear expressions either and simply replaces the entire expression with its concrete value. Cadar and Engler independently developed EGT [5]. EGT starts with pure symbolic execution. When constraints on a programs input parameters become too complex, symbolic execution is paused and the path condition collected thus far instantiated with concrete inputs. Runtime values are then used

to simplify symbolic expressions so that symbolic execution can continue with a mix of symbolic variables and constants. CREST [4] is a recent open-source successor to CUTE. Its main difference to CUTE is a more sophisticated, CFG based, path exploration strategy.

Pex [32] is a parameterized unit testing framework developed by Microsoft. Contrary to the majority of structural testing tools, it performs instrumentation at the .NET intermediate language level. As a result it is able to handle all 'safe' .NET instructions and can also include information from system libraries. Pex can be fully integrated into the Visual Studio development environment. Its tight coupling with the .NET runtime also allows it to handle exceptions, *e.g.* by suggesting guarding statements for objects or preconditions to functions.

Several tools have also been developed for search based testing. ET-S, developed by Daimler [36], uses evolutionary algorithms to achieve various coverage types, including path, branch and data flow coverage. IGUANA [24] is a tool developed for researchers, and incorporates different search approaches, as well as an API for the development of different objective functions. The eToc tool [33], implements an evolutionary strategy for JAVA classes. The tool evolves sequences of method calls in order to achieve branch coverage.

Xie *et al.* [17] were the first to combine concolic and search based testing in a framework called EVACON, which aims to maximize coverage of JAVA classes using both eToc and jCUTE, a JAVA version of CUTE [30].

There have been a number of previous empirical studies involving concolic and search based approaches.

Burnim and Sen [4] considered different search strategies to explore program paths in concolic testing and evaluated their findings on large open source applications including the Siemens benchmark suite [16], `grep` [13], a search utility based on regular expressions, and `vim` [34], a common text editor. An extended version of CUTE [22] has also been applied to the `vim` editor. Since its introduction, DART has been further developed and used in conjunction with other techniques to test functions from real world programs in an order of magnitude of 10,500 LOC [6], [10]. Concolic testing has also been used to search for security vulnerabilities in large Microsoft applications as part of the SAGE tool [11].

Studies in search based software testing have largely involved small laboratory programs, with experiments designed to show that search based testing is more effective than random testing [26], [36]. There are comparatively fewer studies which have featured real world programs; those that have, considered only libraries [33] or parts of applications [15] in order to demonstrate differences between different search based approaches.

The present paper complements and extends this previous work. It is the first to compare both approaches on the same set of unadulterated, non–trivial, real world test subjects. It is also the largest study of search based testing by an order of magnitude.

VII. Conclusions

This paper has investigated the performance of two approaches to automated structural test data generation, the concolic approach embodied in the CUTE tool, and the search based approach implemented in the AUSTIN tool. The empirical study centred on four complete open source applications. The results show that there are many challenges remaining in making automatic test data generation tools robust and of a standard that could be considered 'industrial-strength'. This is because with the exception of one of the test subjects chosen, neither tool managed to generate test data for over 50% of the branches in each application's code.

Out of the many open challenges in automated test data generation, two seem to be the most prominent for future work. On a practical level, tools need to be able to prevent or recover from segmentation faults, so that they may continue the test data generation process to any effect. Secondly test data generation tools need to become much more heterogenious in nature. Instead of performing pure static or dynamic analysis, a combination of both is required in order to tackle problems such as testing uninstrumented code, overcoming limitations of a constraint solver, and preventing flat or rugged fitness landscapes in dynamic testing. Work has already begun investigating how search based algorithms can be used to find solutions in concolic testing in the presence of floating point computations.

VIII. Acknowledgments

The authors would like to thank Nicolas Gold for his helpful comments on conducting the empirical study. Kiran Lakhotia is funded by EU grant IST-33472. Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference). Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London.

References

[1] B. B., F. F., J. J.-M., and L. T. Y. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15:73–96, 2005.

[2] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA, 2004. ACM.

[3] O. Buehler and J. Wegener. Evolutionary functional testing of an automated parking system. In *International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, USA, 2003.

[4] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering (ASE 2008)*, pages 443–446. IEEE, 2008.

[5] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[6] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In S. L. Min and W. Yi, editors, *EMSOFT*, pages 262–271. ACM, 2006.

[7] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 38–48, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 281–290. ACM, 2008.

[10] P. Godefroid. Compositional dynamic test generation. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 47–54. ACM, 2007.

[11] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *RT '07: Proceedings of the 2nd international workshop on Random testing*, pages 1–1, New York, NY, USA, 2007. ACM.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[13] G. grep. http://www.gnu.org/software/grep/.

[14] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[15] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 73–83, London, UK, 2007. ACM Press.

[16] M. J. Harrold and G. Rothermel. http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/Tools/subjects.

[17] K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 425–428, November 2007.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[19] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[20] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1759–1766, Atlanta, GA, USA, 12-16 July 2008. ACM.

[21] Z. Li, M. Harman, and R. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*. To appear.

[22] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, pages 416–426. IEEE Computer Society, 2007.

[23] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[24] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical Report CS-07-14, Department of Computer Science, University of Sheffield, 2007.

[25] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 13–24, Portland, Maine, USA, 2006. ACM.

[26] C. C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng*, 27(12):1085–1110, 2001.

[27] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[28] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[29] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[30] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2006)*, pages 419–423, 2006.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[32] N. Tillmann and J. de Halleux. Pex-white box test generation for.NET. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[33] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128, Boston, USA, 2004. ACM Press.

[34] VIM. http://www.vim.org/.

[35] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 1 – 12, Portland, Maine, USA., 2006. ACM Press.

[36] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[37] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, 2001.

[38] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[39] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 140–150, New York, NY, USA, 2007. ACM.