AUSTIN: A tool for Search Based Software Testing for the C Language and its Evaluation on Deployed Automotive Systems

Kiran Lakhotia
King's College London, CREST,
Strand, London, WC2R 2LS, U.K.
kiran.lakhotia@kcl.ac.uk

Mark Harman
King's College London, CREST,
Strand, London, WC2R 2LS, U.K.
mark.harman@kcl.ac.uk

Hamilton Gross
Berner & Mattner Systemtechnik GmbH,
Gutenbergstr. 15, D–10587 Berlin, Germany
hamilton.gross@berner-mattner.com

***Abstract*—Despite the large number of publications on Search–Based Software Testing (SBST), there remain few publicly available tools. This paper introduces AUSTIN, a publicly available SBST tool for the C language. The paper validates the tool with an empirical study of its effectiveness and efficiency in achieving branch coverage compared to random testing and the Evolutionary Testing Framework (ETF), which is used in-house by Daimler and others for Evolutionary Testing. The programs used in the study consist of eight non–trivial, real-world C functions drawn from three embedded automotive software modules. For the majority of the functions, AUSTIN is at least as effective (in terms of achieved branch coverage) as the ETF, and is considerably more efficient.**

***Keywords*-Software Testing; SBSE; SBST**

I. INTRODUCTION:

Search–Based Software Testing (SBST) was the first Software Engineering problem to be attacked using optimization [1] and also the first to which a search–based optimization technique was applied [2]. Recent years have witnessed a dramatic rise in the growth of work on SBST and in particular on techniques for generating test data that meets structural coverage criteria. Yet, despite an increasing interest in SBST and, in particular, in structural coverage using SBST, there remains a lack of publicly available tools that provide researchers with facilities to perform search–based structural testing.

This paper introduces such a tool, AUSTIN, and reports our experience with it. AUSTIN uses a variant of Korel's [3] 'Alternating Variable Method' (AVM) and augments it with techniques adapted from recent work on directed adaptive random testing and dynamic symbolic execution [4], [5], [6], [7]. It can handle a large subset of C, though there are some limitations. Most notably AUSTIN cannot generate meaningful inputs for strings, void and function pointers, as well as union constructs. Despite these limitations, AUSTIN has been applied 'out of the box' to real industrial code from the automotive industry (see Section V) as well as a number of open source programs [8].

This paper presents an empirical study in which AUSTIN has been compared to an Evolutionary Testing Framework (ETF), which was developed as part of the EvoTest project [9]. The Framework represents a state–of–the–art evolutionary testing system and has been applied to case studies from the automotive and communications industry. Three case studies from the automotive industry, provided by Berner & Mattner Systemtechnik GmbH, formed the benchmark which AUSTIN was compared against for effectiveness and efficiency when generating branch adequate test data.

Automotive code was chosen as the benchmark because the automotive industry is subject to testing standards that mandate structural coverage criteria [10] and so the developers of production code for automotive systems are a natural target for automated test data generation techniques, such as those provided by AUSTIN.

The rest of the paper is organised as follows: Section II provides background information on the field of search–based testing and gives an overview of related work. Section III introduces AUSTIN and describes the different techniques implemented, whilst Section IV provides details about the Evolutionary Testing Framework against which AUSTIN has been compared. The empirical study used to evaluate AUSTIN alongside the hypotheses tested, evaluation and threats to validity are presented in Section V. Section VI concludes.

II. BACKGROUND

Test data generation is a natural choice for Search–Based Software Engineering (SBSE) researchers because the search space is clearly defined (it is the space of inputs to the program) and tools often provide existing infrastructures for representing candidate inputs and for instrumenting and recording their effect. Similarly, the test adequacy criterion is usually well defined and is also widely accepted as a metric worthy of study by the testing community, making it an excellent candidate for a fitness function [11]. The role of the fitness 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 condition `a == b` must be executed as true, a possible objective function is $|a - b|$. When this function is 0, the desired input values have been

978-0-7695-4195-2/10 $26.00 © 2010 IEEE
DOI 10.1109/SSBSE.2010.21

found. Different branch functions exist for various relational operators in predicates [3].

McMinn [12] provides a detailed survey of work on SBST up to approximately 2004. It shows that the most popular search technique applied to structural testing problems has been the Genetic Algorithm. However, other search–based algorithms have also been applied, including parallel Evolutionary Algorithms [13], Evolution Strategies [14], Estimation of Distribution Algorithms [15], Scatter Search [16], [17], Particle Swarm Optimization [18], [19] and Tabu Search [20].

Due to the large body of work on SBST, Ali et al. [21] performed a systematic review of the literature in order to asses the quality and adequacy of empirical studies used in evaluating SBST techniques. One of their key findings is that empirical studies in SBST need to include more statistical analysis, in the form of hypothesis testing, in order to account for the randomness in any meta–heuristic algorithm.

Outside the search–based testing community there has been a growing number of publicly available tools for structural testing problems, most notably from the field of Dynamic Symbolic Execution (DSE) [4], [5], [6], [7]. DSE combines symbolic [22] and concrete execution. Concrete execution drives the symbolic exploration of a program, and runtime values can be used to simplify path constraints produced by symbolic execution to make them more amenable to constraint solving. For example, assume two inputs `a` and `b` have the values `38` and `100` respectively, and that a path condition of interest is of the form `(int)log(a) == b`. Further assume that a particular constraint solver cannot handle the call to the `log` function. Now suppose during concrete execution, the expression `(int)log(38)` evaluated to 3. DSE can then simplify the path condition to `3 == b`. The constraint solver can now be used to provide a value for `b` which satisfies the constraint. A more detailed treatment of this approach can be found in the work of Godefroid et al. [4].

AUSTIN draws together strands of research on search–based testing for structural coverage and DSE so that it can generate branch adequate test data for integers, floating point and pointer type inputs. Currently AUSTIN only addresses one small, but important part of testing: it generates input values that reach different parts of a program. Whether these inputs reveal any faults is still left for the user to decide.

III. AUSTIN

AUgmented Search–based TestINg (AUSTIN) is a structural test data generation tool which combines a simple hill climber for integer and floating point type inputs with a set of constraint solving rules for pointer type inputs. It has been designed as a unit testing tool for C programs. AUSTIN considers a unit to be a function under test and all the functions reachable from within that function.

Algorithm 1 High level description of the AUSTIN-AVM

```
currentSolution := random
bestSolution := currentSolution
doLocalRestart := true
while not reached stopping criterion do
  if solve pointer constraint then
    if solvePC(currentSolution) = NULL then
      currentSolution := random
    end if
  else if trapped at local optimum then
    if doLocalRestart then
      for i := 0 to currentSolution.length do
        localRestart(currentSolution[i])
      end for
      doLocalRestart := false
    else
      currentSolution := random
      doLocalRestart := true
    end if
  else
    improvement := exploratoryMove(currentSolution)
    restartExploration := false
    while improvement do
      bestSolution := currentSolution
      if reached stopping criterion then
        return bestSolution
      end if
      improvement := patternMove(currentSolution)
      restartExploration := true
    end while
    if restartExploration then
      reset search parameters
    end if
  end if
end while
return bestSolution
```

AUSTIN can be used to generate a set of input data for a given function which achieve (some level of) branch coverage for that function. During the test data generation process, AUSTIN does not attempt to execute specific paths through a function in order to cover a target branch; the path taken up to a branch is an emergent property of the search process. The search is guided by an objective function that was introduced by Wegener *et al.* [23] 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 of the target branch's control dependent nodes 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. The branch

Algorithm 2 High level description of solvePC
Inputs: Equivalence graph of symbolic variables EG and candidate solution $currentSolution$

Compute path condition pc for currentSolution
Compute the approximate path condition pc' from pc by dropping all constraints over arithmetic types from pc
Trim pc' by removing all non critical branching nodes
Invert the binary operator ($\in \{=, \neq\}$) of the last constraint in pc'
for all constraints c_i in pc' **do**
  $left :=$ get lhs of c_i
  $right :=$ get rhs of c_i
  Get node n_{left} from EG which contains $left$ or create a new node n_{left} if no such node exists
  Get node n_{right} from EG which contains $right$ or create a new node n_{right} if no such node exists
  if operator op_i in c_i is $=$ **then**
    if n_{left} has an edge with n_{right} in EG **then**
      return NULL {infeasible}
    else
      Merge nodes n_{left} and n_{right}
      Update EG
    end if
  else if operator op_i in c_i is $\neq$ **then**
    if $n_{left} = n_{right}$ **then**
      return NULL {infeasible}
    else
      Add edge between n_{left} and n_{right}
      if $n_{left} \neq$ null node **then**
        Add edge between n_{left} and null node
      end if
      if $n_{right} \neq$ null node **then**
        Add edge between n_{right} and null node
      end if
      Update EG
    end if
  end if
end for
for all nodes n_i in EG **do**
  if n_i has no edge to null node **then**
    $m :=$ NULL
  else
    if n_i represents the address A of a variable **then**
      $m := A$
    else
      $m := malloc$
    end if
  end if
  for all symbolic variables s_i in n_i **do**
    Update corresponding element for s_i in currentSolution with m
  end for
end for
return currentSolution

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 node (2) in Figure 1 as an example, if the false branch is taken at node (2), the branch distance is computed using `|one->key - 10|`. The branch distance is normalised and added to the approach level.

Similar to DSE [4], [6], [7], AUSTIN also instruments the program under test to symbolically execute the program along the concrete path of execution for a particular input vector. Collecting constraints over input variables via symbolic execution serves to aid AUSTIN in solving constraints over memory locations, denoted by pointer inputs to a function. Consider the example given in Figure 1 and suppose execution follows the false branch at node (1). AUSTIN will use a custom procedure to solve the constraint over the pointer input `one`. On the other hand, if the false branch is taken at node (2), the AVM is used to satisfy the condition at node (2).

A. AVM

The AVM used in AUSTIN works by continuously changing each arithmetic type input in isolation. First, a vector is constructed containing the arithmetic type inputs (*e.g.*, integers, floats) to the function under test. All variables in this vector are initialised with random values. Then, so called *exploratory moves* are made for each element in turn. These consist of adding or subtracting a delta from the value of an element. For integral types the delta starts off at 1, *i.e.*, the smallest increment (decrement). When a change leads to an improved fitness value, the search tries to accelerate towards an optimum by increasing the size of the neighbourhood move with every step. These are known as *pattern moves*. The formula used to calculate the delta added or subtracted from an element is: $\delta = 2^{it} * dir * 10^{-prec_i}$, where it is the repeat iteration of the current move, dir either -1 or 1, and $prec_i$ the precision of the i^{th} input variable. The precision only applies to floating point variables, as will be described in Section III-B (*i.e.*, it is 0 for integral types).

Whenever a delta is assigned to a variable, AUSTIN checks for a possible over- or underflow. For integral types this is done with a set of custom macros that use `gcc`'s `typeof` operator [24]. For floating point operations on the other hand, AUSTIN does not check for over- or underflow errors *per se*. Instead, it watches for $\pm$ INF or $\pm$ NaN values. Whenever a potential move leads to an overflow (underflow) of integral types or results in an input taking on the value $\pm$ INF or $\pm$ NaN, AUSTIN discards the move as invalid and explores the next neighbour. As a consequence, code which explicitly checks for $\pm$ INF or $\pm$ NaN cannot be covered by AUSTIN. So far this has not been observed in practice. To handle possible overflow (underflow) in bit fields, AUSTIN sets a lower bound for signed bit fields at $-(2^l/2)$ (0 for unsigned bit fields), and an upper bound at $(2^l/2)-1$ (2^l-1

Node id	Example Function

	`typedef struct item {`
	`  int key;`
	`};`
	`void testme(item* one) {`

(1)	`  if ( one != null ) {`
(2)	`  if ( one->key == 10 )` `  // target`
	`}`

Figure 1. Example C code used for demonstrating how AUSTIN combines custom constraint solving rules for pointer inputs with an AVM for other inputs to a function under test. The goal is to find an input which satisfies the condition at node (2).

for unsigned bit fields), where l is the length of the bit field. A user can also specify custom bounds for every variable. Again, updates to an input which violate these bounds are discarded as infeasible, forcing the search to move on. The main motivation for including such 'bounds checking' in AUSTIN is to save wasteful moves.

Once no further improvements can be found for an input, the search continues exploring the next element in the vector, recommencing with the first element if necessary, until no changes to an input lead to further improvements. At this point the search restarts at another randomly chosen location in the search space. This is known as a *random restart* strategy and is designed to overcome local optima and enable the search to explore a wider region of the input domain for the function under test.

B. Floating Point Variables

As mentioned in the previous section, each floating point variable has an associated precision. For example, setting the precision $prec_i$ of the i^{th} input variable to 1 limits the smallest possible move for that variable to ± 0.1. Increasing the precision to 2 limits the smallest possible move to ± 0.01, and so forth. In practice, the precision of floating point numbers is limited. Double precision types have about 16 decimal digits, and single precision types about 7 decimal digits of precision.

Initially, the precision variable $prec_i$ of each input is set to 0, and whenever the AVM is stuck at a local optimum it tries to increase this value. To avoid redundant changes to $prec_i$, the AVM checks if adding the delta 10^{-prec_i} to a variable changes its value, or, $prec_i$ is less than 7, or 16 for single or double precision type inputs. If the precision of floating point variables cannot be increased any further, or, no exploratory move with an increased precision results in an improved fitness value, the AVM resorts to a random restart.

C. Random Restarts

The AVM performs two types of random restarts in order to escape local optima. The first type is a global restart while the second is a local restart (see Algorithm 1). In a global restart, all input variables are assigned new random values. This is likely to place the starting point for the next hill climb far away from the local optimum where the search got stuck. While this is desirable in many cases, it is not an ideal strategy when a global optimum is surrounded by many local optima. The chances are the search will just get stuck at the same local optimum again. Therefore the AVM also uses a local restart, which is designed to stay in the vicinity of the current search space while still being able to escape from a local optimum.

In a local restart, a random number r (between 0 and 1) is created for each input variable. This number is then 'scaled' by the formula $10^{-prec_i} * r$, where $prec_i$ is the current precision of the i^{th} input variable. This 'scaled' random number is then added to the existing value of the input variable. If such a local restart does not enable the search to make further progress, a global restart is performed. Thus, the AVM alternates between local and global restarts.

D. Pointer Inputs

We will now describe how the AVM has been extended to add automatic support for pointers and dynamic data structures. The constraint solving procedure for pointer inputs has been adapted from the DSE tool CUTE [6].

A high level description of the algorithm for solving pointer constraints is shown in Algorithm 2. It is an improved version of the approach introduced in our earlier work [25], which did not include feasibility checks or use an equivalence graph. Algorithm 2 first constructs a symbolic path condition pc, describing the path taken by the concrete execution. At this stage pc may contain constraints over both arithmetic type, as well as pointer type inputs. AUSTIN thus constructs a sub–path condition, pc', in which all constraints over arithmetic types are dropped. This includes constraints which contain a pointer dereference to a primitive type. pc' is further simplified by removing all constraints which originated from non–critical branching nodes with respect to the current target branch. Furthermore, AUSTIN uses the CIL [26] framework to ensure that the remaining constraints over memory locations are of the form $x = y$ and $x \neq y$, where both x or y may be the constant $null$ or a symbolic variable denoting a pointer input.

The path conditions pc and pc' describe the flow of execution taken by a concrete input vector, which took an infeasible path with respect to the current target branch. To

generate input values which take the execution 'closer' to the target branch in terms of the control flow graph, the binary comparison operator (*i.e.* $=, \neq$) of the last constraint in pc' must be inverted.

From this updated pc' AUSTIN generates an equivalence graph of symbolic variables, which is used to solve pointer constraints. The equivalence relationship between symbolic variables is defined by the '=' operators in pc'. The nodes of the graph represent abstract pointer locations, with node labels representing the set of symbolic variables which point to those locations. Edges between nodes represent inequalities.

The graph is built up incrementally as the search proceeds (*i.e.* with every invocation of the `solvePC` procedure), and always contains a special node to represent the constant $null$. For each constraint c_i in pc', the symbolic variables involved in c_i are extracted. Note that because every symbolic variable is also mapped to a concrete variable, runtime information from concrete executions can be used to resolve aliases between symbolic variables. AUSTIN then checks if the symbolic variables are already contained within the nodes of the equivalence graph. If they are not, a new node for each 'missing' symbolic variable is added to the graph.

Given the node(s) representing the symbolic variables in c_i, AUSTIN checks for satisfiability of c_i. If the symbolic variables in c_i all belong to the same node, and the binary operator in c_i denotes an inequality, the constraint is infeasible. Similarly, if the symbolic variables belong to different nodes connected by an edge, and the binary operator in c_i denotes an equality, the constraint is also infeasible. Given an infeasible constraint, AUSTIN is forced to perform a global random restart, with the hope of traversing a different path through the program. The expectation is that a new path will result in a solvable pc'.

For each feasible constraint in pc', AUSTIN updates the equivalence graph by either adding nodes, adding edges, or merging (unconnected) nodes. Whenever an edge is added between two nodes where neither node labels contain the constant $null$, for example to capture the constraint $x \neq y$, an edge is added from each of the nodes to the node for $null$.

The final step of the algorithm is to derive concrete pointer inputs from the equivalence graph. For every node n in the graph AUSTIN checks if it has an edge to the node for the constant $null$. If no edge exists, all concrete inputs represented by the symbolic variables in n are assigned $null$. Otherwise AUSTIN does the following: if a node n represents the address of another symbolic variable s, all concrete pointer inputs represented by the labels of n are assigned the address of the concrete variable represented by s. Otherwise, a new memory location is created via `malloc`, and each concrete pointer input represented by the labels of n are assigned that memory location.

IV. ETF

The ETF was developed as part of the multidisciplinary European Union–funded research project EvoTest [9] (IST-33472), applying evolutionary algorithms to the problem of testing software systems. It supports both black–box and white–box testing and represents the state–of–the–art for automated evolutionary structural testing. The framework is specifically targeted for use within industry, with much effort spent on scalability, usability and interface design. It is provided as an Eclipse plug-in, and its white–box testing component is capable of generating test cases for single ANSI C functions. A full description of the system is beyond the scope of this document and the interested reader is directed towards the EvoTest web page located at `www.evotest.eu`.

At its core, the ETF contains a user configurable evolutionary engine, which has been integrated from the GUIDE [27] project. The framework also implements a subset of the approach introduced by Prutkina and Windisch [28] to handle pointers and data structures. It maintains different pools of variables, which are used as the target of pointers and whose values are optimized by the evolutionary search. Each pool contains a subset of global variables and formal parameters of the function under test, and all variables in a given pool are of the same type. In addition, for parameters denoting a pointer to a primitive type or data structure, the ETF creates a *temporary* variable whose type matches the target type of the pointer. These temporary variables are also added to the pools.

Each variable in a pool is assigned an index in the range $0, ..., n-1$, where n is the size of its pool (*i.e.*, the number of variables in that pool). The individual (chromosome) describing a pointer input to the function under test contains two fields; one denoting an index and the other a value. The index is used to select a variable from the (correct) pool whose address is used as the target for the pointer input. Note that an index may be negative to denote the constant $null$. If the index corresponds to one of the temporary variables, the value field is used to instantiate that variable. In this way the ETF is able to generate pointers initialised to $null$, pointers to primitive types and pointers to simple data structures (*i.e.* not containing any pointer members). However the approach does not extend to pointers to recursive types, such as lists, trees and graphs.

V. EMPIRICAL STUDY

The objective of the empirical study was to investigate the effectiveness and efficiency of AUSTIN when compared to a state–of–the art evolutionary testing system (the ETF). The study consisted of 8 C functions that are summarised in Table I. They were taken from three embedded software modules and had been selected by Berner & Mattner Systemtechnik GmbH to form part of the evaluation of the ETF within the EvoTest [9] project. The functions had been

Table I
CASE STUDIES. LOC REFERS TO THE TOTAL PREPROCESSED LINES OF C SOURCE CODE CONTAINED WITHIN THE CASE STUDIES. THE LOC HAVE BEEN CALCULATED USING THE CCCC TOOL [29] IN ITS DEFAULT SETTING.

Case Study	LOC	Functions Tested	Software Module
B	18,200	02, 03, 06	Adaptive headlight control
C	7,449	07, 08, 11	Door lock control
D	8,811	12, 15	Electric window control

Table II
TEST SUBJECTS. THE LOC HAVE BEEN CALCULATED USING THE CCCC TOOL [29] IN ITS DEFAULT SETTING. THE NUMBER OF INPUT VARIABLES COUNTS THE NUMBER OF INDEPENDENT INPUT VARIABLES TO THE FUNCTION, *i.e.*, THE MEMBER VARIABLES OF DATA STRUCTURES ARE ALL COUNTED INDIVIDUALLY.

Obfuscated Function Name	LOC	Branches	Nesting Level	# Inputs	Pointer Inputs
02	919	420	14	80	no
03	259	142	12	38	no
06	58	36	6	14	no
07	85	110	11	27	yes
08	99	76	7	29	yes
11	199	129	4	15	yes
12	67	32	9	3	no
15	272	216	4	28	yes

chosen to provide a representative sample of real world automotive code, with particular attention paid to the number of branches and nesting level. Table II gives a breakdown of relevant metrics for the selected functions.

Effectiveness of AUSTIN

In order to investigate the effectiveness of AUSTIN compared to the ETF we formulated the following null and alternate hypotheses:

H_0 : *AUSTIN is as effective as the ETF in achieving branch coverage.*
H_A : *AUSTIN is more effective than the ETF in achieving branch coverage.*

Efficiency of AUSTIN

Alongside coverage, efficiency is also of paramount importance especially in an industrial setting. To compare the efficiency of AUSTIN (in terms of fitness evaluations) to the ETF, we formulated these null and alternative hypotheses:

H_0 : *AUSTIN is equally as efficient as the ETF in achieving branch coverage of a function.*
H_A : *AUSTIN is more efficient than the ETF in achieving branch coverage of a function.*

A. Experimental Setup

The data for the experiments with the ETF on the functions listed in Table II had already been collected for the evaluation phase of the EvoTest project [9]. This section serves to describe how the ETF had been configured and how AUSTIN was adapted to ensure as fair a comparison as possible between AUSTIN and the ETF.

Every branch in the function under test was treated as a goal for both the ETF and AUSTIN. The order in which branches are attempted differs between the two tools. AUSTIN attempts to cover branches in level-order of the Control Flow Graph (CFG), starting from the exit node, while the ETF attempts branches in level-order of the CFG starting from the entry node. In both tools, branches that are covered serendipitously while attempting a goal are removed from the list of goals. The fitness budget for each tool was set to $10,000$ evaluations per branch.

The ETF supports a variety of search algorithms. For the purpose of this study, the ETF was configured to use a Genetic Algorithm (GA) whose parameters were manually tuned to provide a good set which was used for all eight functions. The GA was set up to use a population size of 200, deploy strong elitism as its selection strategy, use a mutation rate of 1% and a crossover rate of 100%.

The ETF also provides the option to reduce the size of the search space for the GA by restricting the bounds of each input variable, or even completely excluding variables from the search. Reducing the size of the input domain will improve the efficiency of search–based testing [30]. The input domain reduction in the ETF is a manual process and was carried out by members of the EvoTest project for each of the eight functions. AUSTIN was configured to apply the same input domain reduction in order to ensure a fair comparison. Finally, due to the stochastic nature of the algorithms used in both tools, each tool was applied 30 times to each function.

B. Evaluation

Effectiveness of AUSTIN. Figure 2 shows the level of coverage achieved by both the ETF and AUSTIN with error bars in each column indicating the standard error of the mean. The results provide evidence to support the claim that AUSTIN can be equally effective in achieving branch coverage than the more complex search algorithm used as part of the ETF. In order to test the first hypothesis, a test for statistical significance was performed to compare the coverage achieved by each tool for each function. A two–tailed test was chosen such that *reductions* in the branch coverage achieved by AUSTIN compared with the ETF could also be tested. Since the samples were often distinctly un–normally distributed and possessed heterogeneous variances and skew, the samples were first rank–transformed as recommended by Ruxton [31]. Then a two–tailed t–test for unequal variances with ($p \leq 0.05$) was carried out on the ranked samples. The two–tailed t-test was used instead of the Wilcoxon–Mann–Whitney test because the latter is sensitive to differences in the shape and variance of the distributions.

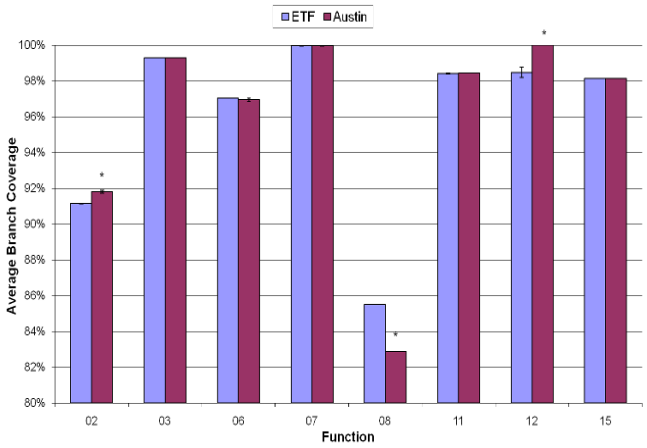

Figure 2. Average branch coverage of the ETF versus AUSTIN. The $y - axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$).

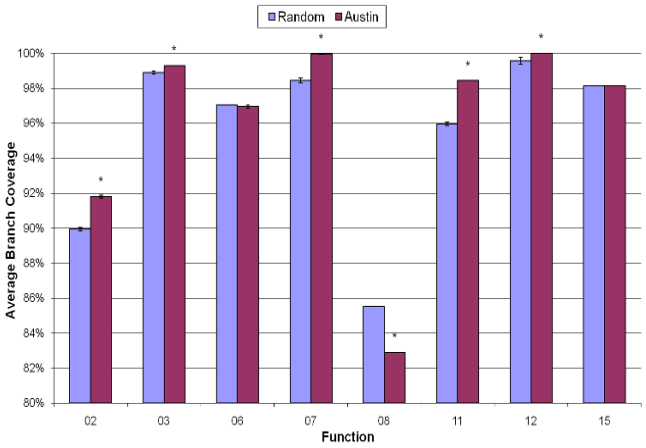

Figure 3. Average branch coverage of random search versus AUSTIN. The $y - axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$).

As shown in Figure 2 and detailed in Table III, AUSTIN delivered a statistically significant increase in coverage for functions 02 and 12. For function 08 AUSTIN achieved a statistically significant lower branch coverage than ETF (82.9% vs. 85.5%, $t(58) = \text{Inf}$). For all other functions, the null–hypothesis that the coverage achieved by the two tools comes from the same population was not rejected.

Function 08 is interesting because it is the only function for which AUSTIN performs significantly worse than the ETF. Therefore the results were analysed in more detail. The first point of interest was the constant number of fitness evaluations AUSTIN used during the 30 runs of this function. This can only occur in one of two cases: 1) AUSTIN is able to find a solution for each target branch from its initial starting point, and the starting points are all equidistant from the global optima; 2) for all targets which require AUSTIN to perform a random restart, it fails to find a solution, *i.e.*, the random restart has no effect on the success of AUSTIN. In this case it will continue until its fixed limit of fitness evaluations has been reached. For function 08 the latter case was true.

Analysing AUSTIN's coverage for function 08 revealed that it was unable to cover thirteen branches. These branches were guarded by a 'hard to cover' condition. Manual analysis showed that the difficult condition becomes feasible when traversing only two out of 63 branches prior to it. The other 61 branches lead to a 'killing' assignment to the input variable, whose value is checked in the difficult guarding condition. The paths which contain one of the two branches, which make the difficult condition feasible, are themselves hard to cover. To check if AUSTIN's failure was due to the inherent difficulty of the problem or, because not enough resources had been allocated, we repeated 30 runs for AUSTIN for function 08, this time without any input domain reduction, and a fitness budget of $100,000$ evaluations per branch. The results show that, given this larger fitness budget, AUSTIN is on average able to cover 97.60% of the branches. This is a marked increase from the average coverage of 82.89% shown in Figure 2. We could not repeat the experiments for the ETF with the extended fitness budget of $100,000$ evaluations per branch, because the fitness budget of $10,000$ evaluations per branch is currently hard coded in the ETF, and we do not have access to its source code. Therefore it is not possible to say how the ETF would have performed given a larger fitness budget.

Efficiency of AUSTIN. Figure 4 shows the average number of fitness evaluations used by both ETF and AUSTIN when trying to achieve coverage of each function. In order to test the second hypothesis, a two–tailed test for statistical significance was performed to compare the mean number of fitness evaluations used by each tool to cover each function. Since the difference in achieved coverage between the two tools was generally very small, it was neglected when comparing their efficiency. A two–tailed test was carried out to also test for functions in which AUSTIN's efficiency was worse than that of the ETF. The distributions of the samples were sufficiently normal (as determined by bootstrap re–sampling each sample–pair over 1000 iterations and visual inspection of the resulting distribution of mean values) to proceed with a two–tailed t–test for unequal variances on the raw un–ranked samples ($p \leq 0.05$).

As shown in Figure 4, AUSTIN delivered a statistically significant increase in efficiency compared with the ETF for functions 03, 06, 07, 11, 12, 15. For functions 02 and 08, AUSTIN used a statistically significant larger number of

Table III
SUMMARY OF FUNCTIONS WITH A STATISTICALLY SIGNIFICANT DIFFERENCE IN THE BRANCH COVERAGE ACHIEVED BY AUSTIN AND THE ETF. THE COLUMNS STDDEV INDICATE THE STANDARD DEVIATION FROM THE MEAN FOR ETF AND AUSTIN. THE T VALUE COLUMN SHOWS THE DEGREES OF FREEDOM (VALUE IN BRACKETS) AND THE RESULT OF THE T–TEST. A P VALUE OF LESS THAN 0.05 MEANS THERE IS A STATISTICALLY SIGNIFICANT DIFFERENCE IN THE MEAN COVERAGE BETWEEN ETF AND AUSTIN.

Function	Coverage ETF (%)	StdDev (%)	Coverage AUSTIN (%)	StdDev (%)	t value	p
02	91.15	0.09	91.81	0.42	$t(58) = 7.15$	$1.6 \cdot 10^{-9}$
08	85.53	0.00	82.89	0.00	$t(58) = Inf$	0
12	98.48	1.76	100.0	0.00	$t(63) = 4.93$	$6.3 \cdot 10^{-6}$

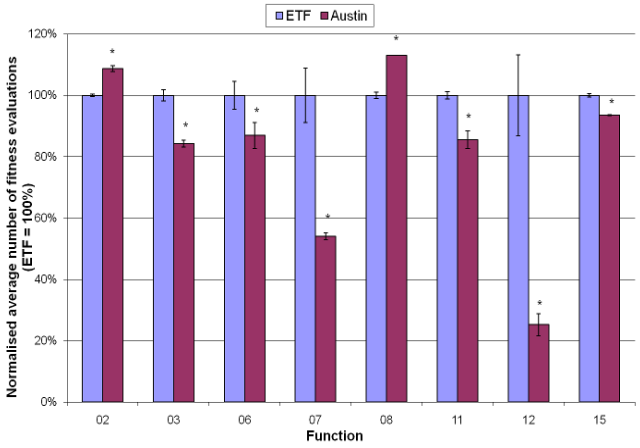

Figure 4. Average number of fitness evaluations (normalised) for ETF versus AUSTIN. The $y - axis$ shows the normalised average number of fitness evaluations for each tool relative to the ETF (shown as 100%) for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$).

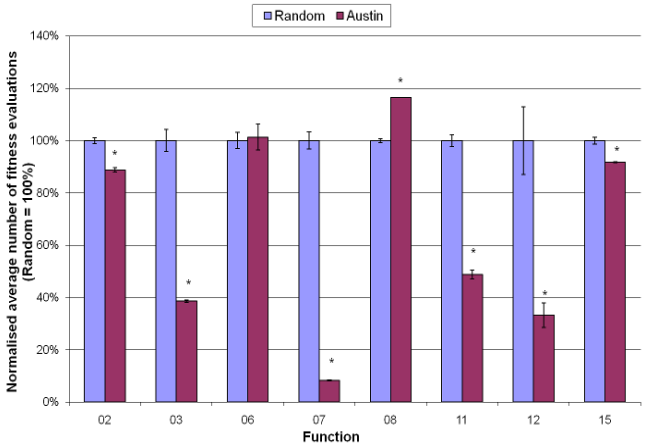

Figure 5. Average number of fitness evaluations (normalised) for random versus AUSTIN. The $y - axis$ shows the normalised average number of fitness evaluations for each tool relative to the random search (shown as 100%) for each of the functions shown on the $x-axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$).

evaluations to achieve its respective level of branch coverage. The results are summarised in Table IV.

Comparison with random search. As a sanity check, the efficiency and effectiveness of AUSTIN was also compared with a random search. Since the random search was performed using the ETF, the same pointer handling technique and input domain reduction were applied as described in Sections IV and V-A respectively. For each branch the random search was allowed at most $10,000$ evaluations. Any branches covered serendipitously by random during the test testing process were counted as covered and removed from the pool of target branches.

The coverage data is presented in Figure 3 and efficiency in Figure 5. Results show that, using the same tests for statistical significance as described in the previous paragraphs, AUSTIN covers statistically significantly more branches than random for functions 02, 03, 07, 11 and 12. For functions 06 and 15 there is no statistically significant difference in coverage, while for function 08, AUSTIN performs statistically significantly worse than random. Recall though, that given a larger fitness budget as mentioned above, AUSTIN is able to achieve a higher coverage for function 08 than the one shown in Figures 2 and 3.

Comparing AUSTIN's efficiency with that of a random search, AUSTIN is statistically significantly more efficient than random for functions 02, 03, 07, 11, 12 and 15. For function 06 we cannot say that either random or AUSTIN is more efficient, while for function 08 random is statistically significantly more efficient than AUSTIN.

C. Threats to Validity

Naturally there are threats to validity in any empirical study such as this. This section provides a brief overview of the threats to validity and how they have been addressed. The paper studied two hypotheses; 1) that AUSTIN is more effective than the ETF in achieving branch coverage of the functions under test and 2) that AUSTIN is more efficient than the ETF. Whenever comparing two different techniques, it is important to ensure that the comparison is as reliable as possible. Any bias in the experimental design that could affect the obtained results poses a threat to the *internal validity* of the experiments. One potential source of bias comes from the settings used for each tool in the

Table IV
SUMMARY OF FUNCTIONS WITH STATISTICALLY SIGNIFICANT DIFFERENCES IN THE NUMBER OF FITNESS EVALUATIONS USED. THE COLUMNS STDDEV INDICATE THE STANDARD DEVIATION FROM THE MEAN FOR ETF AND AUSTIN. THE T VALUE COLUMN SHOWS THE DEGREES OF FREEDOM (VALUE IN BRACKETS) AND THE RESULT OF THE T–TEST. A P VALUE OF LESS THAN 0.05 MEANS THERE IS A STATISTICALLY SIGNIFICANT DIFFERENCE IN THE MEAN NUMBER OF FITNESS EVALUATIONS BETWEEN ETF AND AUSTIN.

Function	Evals ETF (%)	StdDev (%)	Evals AUSTIN (%)	StdDev (%)	t value	p
02	100	1.90	108.78	0.08	$t(58) = 7.67$	$2.20 \cdot 10^{-10}$
03	100	11.14	84.23	6.18	$t(58) = 7.00$	$2.00 \cdot 10^{-9}$
06	100	24.76	86.94	23.42	$t(58) = 2.10$	0.04
07	100	52.40	54.09	6.04	$t(63) = 4.79$	$1.00 \cdot 10^{-5}$
08	100	5.97	113.10	0.00	$t(58) = 9.11$	$8.70 \cdot 10^{-13}$
11	100	6.90	85.58	15.79	$t(58) = 4.41$	$4.49 \cdot 10^{-5}$
12	100	78.33	25.25	19.48	$t(63) = 5.13$	$3.02 \cdot 10^{-6}$
15	100	3.03	93.55	0.90	$t(58) = 9.22$	$5.80 \cdot 10^{-13}$

experiments, and the possibility that the setup could have favoured or harmed the performance of one or both tools.

The experiments with the ETF had already been completed as part of the EvoTest project, thus it was not possible to influence the ETF's setup. It had been manually tuned to provide the best consistent performance across the eight functions. Therefore, care was taken to ensure AUSTIN was adjusted as best as possible to use the same settings as the ETF.

Another potential source of bias comes from the inherent stochastic behaviour of the meta–heuristic search algorithms used in AUSTIN and the ETF. 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 at least 30 times. To check if one technique is superior to the other a test for a statistically significant difference in the mean of the samples was performed. Care was taken to examine the distribution of the data first, in order to ensure the most robust statistical test was chosen to analyse the data.

A further source of bias includes the selection of the functions 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 functions used in the study had been selected by Berner & Mattner Systemtechnik GmbH on the basis that they provided interesting and worthwhile candidates for automated test data generation. Particular attention was paid to the number of branches each function contained, as well as the maximum nesting level of `if` statements within a function. Finally, all the functions from the study contain machine generated code only. While the overall number of branches provides a large pool of results from which to make observations, the number of functions itself is relatively small. Therefore, caution is required before making any claims as to whether these results would be observed on other functions, in particular hand written code.

VI. CONCLUSION

This paper has introduced and evaluated the AUSTIN tool for search–based software testing. AUSTIN is a free, publicly available tool for search–based test data generation. It uses an alternating variable method, augmented with a set of simple constraint solving rules for pointer inputs to a function. Test data is generated by AUSTIN to achieve branch coverage for large C functions. In a comparison with the ETF, a state–of–the–art evolutionary testing framework, AUSTIN performed as effectively and considerably more efficiently than the ETF for 7 out of 8 non–trivial C functions, which were implemented using code–generation tools.

ACKNOWLEDGMENT

We would like to thank Bill Langdon for his helpful comments and Arthur Baars for his advice on the Evolutionary Testing Framework. Kiran Lakhotia is funded by EPSRC grant EP/G060525/1. Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London.

REFERENCES

[1] W. Miller and D. L. Spooner, "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, September 1976.

[2] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of Genetic Algorithms to Software Testing," in *Proceedings of the 5th International Conference on Software Engineering and Applications*, Toulouse, France, 7-11 December 1992, pp. 625–636.

[3] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[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*, vol. 3639. Springer, 2005, pp. 2–23.

[6] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*. ACM, 2005.

[7] N. Tillmann and J. de Halleux, "Pex-white box test generation for.NET," in *TAP*, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153.

[8] K. Lakhotia, P. McMinn, and M. Harman, "Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?" in *4th Testing Academia and Industry Conference - Practice and Research Techniques*, 2009, pp. 95–104.

[9] H. Gross, P. M. Kruse, J. Wegener, and T. Vos, "Evolutionary white-box software test with the evotest framework: A progress report," in *ICSTW '09*, Washington, DC, USA, 2009, pp. 111–120.

[10] Radio Technical Commission for Aeronautics, "RTCA DO178-B Software considerations in airborne systems and equipment certification," 1992.

[11] M. Harman and J. Clark, "Metrics are fitness functions too," in 10^{th} *International Software Metrics Symposium (METRICS 2004)*. Los Alamitos, California, USA: IEEE Computer Society Press, Sep. 2004, pp. 58–69.

[12] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[13] E. Alba and F. Chicano, "Observations in using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing," *Computers & Operations Research*, vol. 35, no. 10, pp. 3161–3183, October 2008.

[14] ——, "Software Testing with Evolutionary Strategies," in *Proceedings of the 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE '05)*, vol. 3943. Heraklion, Crete, Greece: Springer, September 2005, pp. 50–65.

[15] R. Sagarna, A. Arcuri, and X. Yao, "Estimation of Distribution Algorithms for Testing Object Oriented Software," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*. Singapore: IEEE, 25-28 September 2007, pp. 438–444.

[16] R. Blanco, J. Tuya, E. Daz, and B. A. Daz, "A Scatter Search Approach for Automated Branch Coverage in Software Testing," *International Journal of Engineering Intelligent Systems (EIS)*, vol. 15, no. 3, pp. 135–142, September 2007.

[17] R. Sagarna, "An Optimization Approach for Software Test Data Generation: Applications of Estimation of Distribution Algorithms and Scatter Search," Ph.D. dissertation, University of the Basque Country, San Sebastian, Spain, January 2007.

[18] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," in *Proceedings of the First International Conference on Software Testing, Verfication and Validation (ICST 2008)*. Lillehammer, Norway: IEEE Computer Society, 9-11 April 2008, pp. 525–528.

[19] A. Windisch, S. Wappler, and J. Wegener, "Applying Particle Swarm Optimization to Software Testing," in *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. London, England: ACM, 7-11 July 2007, pp. 1121–1128.

[20] E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado, "A Tabu Search Algorithm for Structural Software Testing," *Computers & Operations Research*, vol. 35, no. 10, pp. 3052–3072, October 2008.

[21] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering*, To appear.

[22] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[23] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[24] Free Software Foundation, "Gcc, the gnu compiler collection," 2009. [Online]. Available: http://gcc.gnu.org/

[25] 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*. Atlanta, GA, USA: ACM, 12-16 Jul. 2008, pp. 1759–1766.

[26] 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*, vol. 2304, pp. 213–228, 2002.

[27] L. D. Costa and M. Schoenauer, "Bringing evolutionary computation to industrial applications with GUIDE," 2009.

[28] M. Prutkina and A. Windisch, "Evolutionary Structural Testing of Software with Pointers," in *Proceedings of 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008*. Lillehammer, Norway: IEEE, 9-11 April 2008, pp. 231–231.

[29] T. Littlefair, "An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment," Ph.D. dissertation, Faculty of computing, health and science, Edith Cowan University, Australia, 2001.

[30] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," in *ESEC/SIGSOFT FSE*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 155–164.

[31] G. D. Ruxton, "The unequal variance t-test is an underused alternative to Student's t-test and the Mann-Whitney U test," *Behavioral Ecology*, vol. 17, no. 4, pp. 1045–2249;1465–7279, Jul 2006.