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
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 assess 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 \((\text{int})\log(a) == b\). Further assume that a particular constraint solver cannot handle the call to the \(\log\) function. Now suppose during concrete execution, the expression \((\text{int})\log(38)\) 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

```plaintext
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
        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 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 \( \text{currentSolution} \)

1. Compute path condition \( pc \) for \( \text{currentSolution} \)
2. Compute the approximate path condition \( pc' \) from \( pc \) by dropping all constraints over arithmetic types from \( pc \)
3. Trim \( pc' \) by removing all non critical branching nodes
4. Invert the binary operator (\( \epsilon \in \{=, \neq\} \)) 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 \( n \) do
    Update corresponding element for \( s_i \) in \( \text{currentSolution} \) with \( m \)
end for
end for

return \( \text{currentSolution} \)

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 \([\text{one->key} - 10]\). 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 \( \text{one} \). 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: \( \delta = 2^i \ast \text{dir} \ast 10^{-\text{prec}_i} \), where \( i \) is the repeat iteration of the current move, \( \text{dir} \) either \(-1 \) or \( 1 \), and \( \text{prec}_i \) 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 \texttt{gcc}'s \texttt{typeof} operator [24]. For floating point operations on the other hand, AUSTIN does not check for over- or underflow errors \textit{per se}. Instead, it watches for \( \pm \text{INF} \) or \( \pm \text{NaN} \) values. Whenever a potential move leads to an overflow (underflow) of integral types or results in an input taking on the value \( \pm \text{INF} \) or \( \pm \text{NaN} \), AUSTIN discards the move as invalid and explores the next neighbour. As a consequence, code which explicitly checks for \( \pm \text{INF} \) or \( \pm \text{NaN} \) 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 \((2^l/2) - 1\).
### Example Function

```c
typedef struct item {
    int key;
};

void testme(item* one) {
    if (one != null) {
        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 \( \text{prec}_i \) of the \( i^{th} \) input variable to \( 1 \) limits the smallest possible move for that variable to \( \pm 0.1 \). Increasing the precision to \( 2 \) limits the smallest possible move to \( \pm 0.01 \), 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 \( \text{prec}_i \) 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 \( \text{prec}_i \), the AVM checks if adding the delta \( 10^{-\text{prec}_i} \) to a variable changes its value, or, \( \text{prec}_i \) 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 \( 10^{-\text{prec}_i} \times r \), where \( \text{prec}_i \) 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 \( \text{null} \) 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 $pe^c$ must be inverted.

From this updated $pe^c$ 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 $pe^c$. 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 $pe^c$, 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 $pe^c$.

For each feasible constraint in $pe^c$, 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 evolution 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 $0, \ldots, n - 1$, 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
In order to investigate the effectiveness of AUSTIN compared to the ETF we formulated the following null and alternate hypotheses:

\[ H_0 : \text{AUSTIN is as effective as the ETF in achieving branch coverage.} \]
\[ H_A : \text{AUSTIN is more effective than the ETF in achieving branch coverage.} \]

Effectiveness of AUSTIN

Effectiveness of AUSTIN. 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.

### Table I

<table>
<thead>
<tr>
<th>Case Study</th>
<th>LOC</th>
<th>Functions Tested</th>
<th>Software Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>18,200</td>
<td>02, 03, 06</td>
<td>Adaptive headlight control</td>
</tr>
<tr>
<td>C</td>
<td>7,449</td>
<td>07, 08, 11</td>
<td>Door lock control</td>
</tr>
<tr>
<td>D</td>
<td>8,811</td>
<td>12, 15</td>
<td>Electric window control</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Function Name</th>
<th>LOC</th>
<th>Branches</th>
<th>Nesting Level</th>
<th># Inputs</th>
<th>Pointer Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>919</td>
<td>420</td>
<td>14</td>
<td>80</td>
<td>no</td>
</tr>
<tr>
<td>03</td>
<td>259</td>
<td>142</td>
<td>12</td>
<td>38</td>
<td>no</td>
</tr>
<tr>
<td>06</td>
<td>58</td>
<td>36</td>
<td>6</td>
<td>14</td>
<td>no</td>
</tr>
<tr>
<td>07</td>
<td>85</td>
<td>110</td>
<td>11</td>
<td>27</td>
<td>yes</td>
</tr>
<tr>
<td>08</td>
<td>99</td>
<td>76</td>
<td>7</td>
<td>29</td>
<td>yes</td>
</tr>
<tr>
<td>11</td>
<td>199</td>
<td>129</td>
<td>4</td>
<td>15</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>67</td>
<td>32</td>
<td>9</td>
<td>3</td>
<td>no</td>
</tr>
<tr>
<td>15</td>
<td>272</td>
<td>216</td>
<td>4</td>
<td>28</td>
<td>yes</td>
</tr>
</tbody>
</table>
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) = 1.99, p < 0.05). For all other functions, the null-hypothesis that the coverage achieved by the two tools is equal 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 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 ≤ 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Coverage ETF (%)</th>
<th>StdDev (%)</th>
<th>Coverage AUSTIN (%)</th>
<th>StdDev (%)</th>
<th>t value</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>91.15</td>
<td>0.09</td>
<td>91.81</td>
<td>0.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>85.53</td>
<td>0.00</td>
<td>82.89</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>98.48</td>
<td>1.76</td>
<td>100.00</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

### 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 ≤ 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 ≤ 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, 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

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.
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


<table>
<thead>
<tr>
<th>Function</th>
<th>EvalS ETF (%)</th>
<th>StdDev (%)</th>
<th>EvalS AUSTIN (%)</th>
<th>StdDev (%)</th>
<th>t value</th>
<th>p value</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>100</td>
<td>1.90</td>
<td>108.78</td>
<td>0.08</td>
<td>t(58) = 7.67</td>
<td>2.20 · 10⁻¹⁰</td>
</tr>
<tr>
<td>03</td>
<td>100</td>
<td>11.14</td>
<td>84.23</td>
<td>6.18</td>
<td>t(58) = 7.00</td>
<td>2.00 · 10⁻⁹</td>
</tr>
<tr>
<td>06</td>
<td>100</td>
<td>24.76</td>
<td>86.94</td>
<td>23.42</td>
<td>t(58) = 2.10</td>
<td>0.04</td>
</tr>
<tr>
<td>07</td>
<td>100</td>
<td>52.40</td>
<td>54.09</td>
<td>6.04</td>
<td>t(63) = 4.79</td>
<td>1.00 · 10⁻⁵</td>
</tr>
<tr>
<td>08</td>
<td>100</td>
<td>5.97</td>
<td>113.10</td>
<td>0.00</td>
<td>t(58) = 9.11</td>
<td>8.70 · 10⁻¹³</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>6.90</td>
<td>85.58</td>
<td>15.79</td>
<td>t(58) = 4.41</td>
<td>4.49 · 10⁻⁵</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>78.33</td>
<td>25.25</td>
<td>19.48</td>
<td>t(63) = 5.13</td>
<td>3.02 · 10⁻⁶</td>
</tr>
<tr>
<td>15</td>
<td>100</td>
<td>3.03</td>
<td>93.55</td>
<td>0.90</td>
<td>t(58) = 9.22</td>
<td>5.80 · 10⁻¹³</td>
</tr>
</tbody>
</table>


