Information and Software Technology 55 (2013) 112–125

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

AUSTIN: An open source tool for search based software testing of C programs

Kiran Lakhotia [a,*], Mark Harman [a], Hamilton Gross [b]

[a] CREST, University College London, Gower Street, London WC1E 6BT, United Kingdom
[b] Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, D-10587 Berlin, Germany

ARTICLE INFO

Article history:
Available online 3 April 2012

Keywords:
Software testing
SBSE
SBST

ABSTRACT

Context: 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 open source SBST tool for the C language.[1] The paper is an extension of previous work [1]. It includes a new hill climb algorithm implemented in AUSTIN and an investigation into the effectiveness and efficiency of different pointer handling techniques implemented by AUSTIN's test data generation algorithms.
Objective: To evaluate the different search algorithms implemented within AUSTIN on open source systems with respect to effectiveness and efficiency in achieving branch coverage. Further, to compare AUSTIN against a non-publicly available, state-of-the-art Evolutionary Testing Framework (ETF).
Method: First, we use example functions from open source benchmarks as well as common data structure implementations to check if the decision procedure for pointer inputs, introduced in this paper, differs in terms of effectiveness and efficiency compared to a simpler alternative that generates random memory graphs. A second empirical study formulates two alternate hypotheses regarding the effectiveness and efficiency of AUSTIN compared to the ETF. These hypotheses are tested using a paired Wilcoxon test.
Results and Conclusion: The first study highlights some practical problems with the decision procedure for pointer inputs described in this paper. In particular, if the code under test contains insufficient guard statements to enforce constraints over pointers, then using a constraint solver for pointer inputs may be suboptimal compared to a method that generates random memory graphs. The programs used in the second study do not require any constraint solving for pointer inputs and consist of eight non-trivial, real-world C functions drawn from three embedded automotive software modules. For these functions, AUSTIN is competitive compared to the ETF, achieving an equal or higher branch coverage for six of the functions. In addition, for functions where AUSTIN's branch coverage is equal or higher, AUSTIN is more efficient than the ETF.

1. Introduction

Search-Based Software Testing (SBST) was the first Software Engineering problem to be attacked using optimization [2] and also the first to which a search-based optimization technique was applied [3]. 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 supports three search algorithms: A random search, a hill climber, and a hill climb algorithm augmented with symbolic execution. The hill climb algorithms are a variant of Korel's 'Alternating Variable Method' (AVM) [4].

AUSTIN 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 [1] as well as a number of open source programs [5].

This paper presents an evaluation of the different search algorithms implemented within AUSTIN on functions taken from open source programs as well as data structure implementations. Then, the hill climb algorithms in AUSTIN are compared to a closed source, state-of-the-art Evolutionary Testing Framework (ETF). The ETF was developed as part of the EvoTest project [6] and has itself been applied to case studies from the automotive and communications industry.

* Corresponding author.
E-mail addresses: k.lakhotia@ucl.ac.uk (K. Lakhotia), mark.harman@ucl.ac.uk (M. Harman), hamilton.gross@berner-mattner.com (H. Gross).

[1] http://code.google.com/p/austin-sbst/.

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.03.009

For the comparison we chose three case studies from the automotive industry, provided by Berner & Mattner Systemtechnik GmbH. They form the benchmark against which AUSTIN was compared 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 [7] 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 2 provides background information on the field of search-based testing and gives an overview of related work. Section 3 introduces AUSTIN and describes the different test data generation techniques it implements. These techniques are evaluated in Section 4. Section 5 then presents an empirical study that compares AUSTIN's hill climber against the ETF and discusses any threats to validity. Section 6 concludes.

2. 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 [8].

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 [4].

McMinn [9] 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. More recently other search-based algorithms have also been applied, including parallel Evolutionary Algorithms [10], Evolution Strategies [11], Estimation of Distribution Algorithms [12], Scatter Search [13,14], Particle Swarm Optimization [15,16] and Tabu Search [17].

Due to the large body of work on SBST, Ali et al. [18] 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) [19–22]. DSE combines symbolic [23] 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 an input needs to satisfy the condition `(int) log (a) == b` in order to execute a target branch. During concrete execution, the values `38` and `100` were used for the inputs `a` and `b` respectively. Further assume that a particular constraint solver cannot handle the call to the `log` function because its source code is unavailable. Now suppose during concrete execution, the expression `(int) log (38)` evaluated to 3. DSE can simplify the path condition to `3 == b` and then use its constraint solver to provide a value for `b`. Assuming the program under test and the implementation of the `log` function are deterministic, DSE is thus able to generate input values that exercise the branch in the program that depends on the condition `(int) log (a) == b` being true. A more detailed treatment of this approach can be found in the work of Godefroid et al. [19].

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 addresses a 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. This is the so-called oracle problem.

3. AUSTIN

AUgmented Search-based TestINg (AUSTIN) is a structural test data generation tool for unit testing C programs. AUSTIN considers a unit to be a function under test and all the functions reachable from within that function. It supports three test data generation techniques: A random search, a hill climber (AVM) and a hill climber augmented with symbolic execution (AVM+).

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.

When used with a guided search such as hill climbing, AUSTIN uses the objective function that was introduced by Wegener et al. [24] 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 is a measure for 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 target in control flow terms.

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current target branch. It provides a quantification in the range $[0, 1]$ of a boolean branch condition, such that the value zero is obtained *iff* the condition evaluates to true. Values close to 1 indicate that the condition is far from being satisfied. Intermediate values should be such as to smoothly guide the search toward satisfying the condition. Arcuri [25] presents an evaluation of different normalising functions for the branch distance metric.

3.1. Random search

AUSTIN's random search generates random input values for all arithmetic type inputs to a function, *i.e.*, formal parameters and global variables. Complex data types such as pointers are initialized as follows: Every pointer input has a 0.5 probability of being assigned the constant `NULL`, or a valid memory location. If a pointer is assigned a value, another coin toss chooses between using an existing address (*e.g.*, that of an existing input parameter) and creating a new heap address (via `malloc`). This allows the random search to construct possibly cyclic data structures. Whenever a new memory location is created thru `malloc`, the random search applies its initialization procedure to that new location; for example, initializing the fields of a data structure or assigning a random value in case of pointers to primitive data types. Fig. 1 shows what kind of input graph (middle) the random search might construct

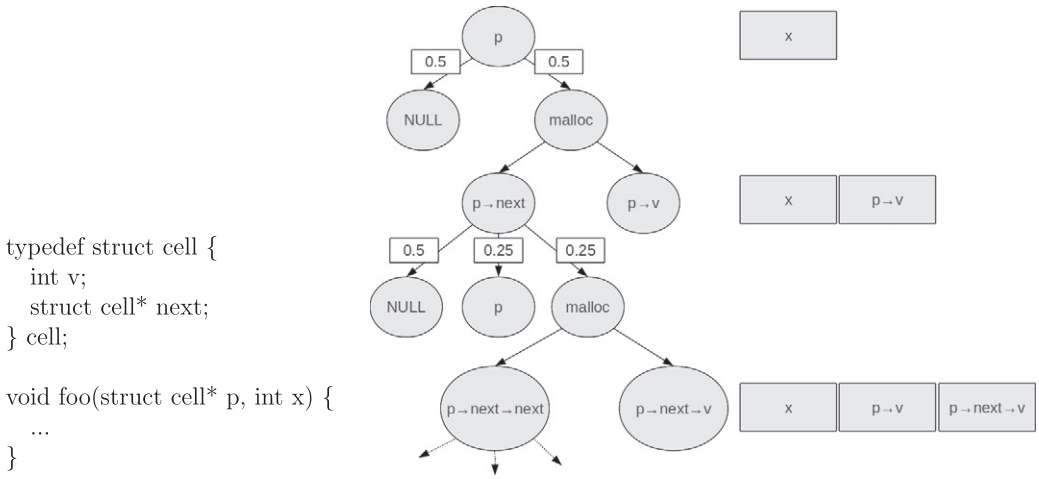

Fig. 1. Example illustrating how a random memory graph is built up for the code on the left. Pointer inputs will be set to `NULL` with a probability of 0.5 to ensure the graph generation terminates within reasonable time. The boxes to the right of the graph show how a vector of arithmetic type inputs is constructed based in part on the shape of the memory graph.

for the code fragment on the left and how this translates to inputs of a function (right).

The random search does not require any fitness function and only uses simple code instrumentation to evaluate the level of branch coverage achieved. If an input covers a new branch it is saved as a test case in an archive. Otherwise the search continues until either all branches have been covered, or, a stopping criterion is reached. The stopping criterion limits the maximum number of inputs that may be generated for a given function. It is typically set to a value an order of magnitude larger than the number of branches within the function under test.

3.2. AVM

AUSTIN also contains a hill climb algorithm, based on Korel's Alternating Variable Method (AVM). The AVM 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 applies to floating point variables only (*i.e.*, it is 0 for integral types). It denotes a scale factor for the size of a neighbourhood move. For example, setting the precision ($prec_i$) of an input to 1 limits the smallest possible move to ±0.1. Increasing the precision to 2 limits the smallest possible move to ±0.01, and so forth. For all experiments carried out in this paper, the precision for floating point variables was set to 2. This provides a reasonable trade-off between accuracy and speed of exploring the search space. The larger the precision of a variable, the bigger the search space for that variable.

Whenever a delta is assigned to a variable, the AVM 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 [26]. For floating point operations on the other hand, the AVM does not check for over- or underflow errors *per se*. Instead, it looks for ±INF or ±NaN values. Whenever a potential move leads to an overflow (underflow) of integral types or results in an input taking on the value ±INF or ±NaN, the AVM discards the move as invalid and explores the next neighbour. As a consequence, code which explicitly checks for ±INF or ±NaN cannot be covered by the AVM.[2] To handle possible overflow (underflow) in bit fields, the AVM sets a lower bound for signed bit fields at $-(2^l/2)$ (0 for unsigned bit fields), and an upper bound at $(2^l/2) - 1$ ($2^l - 1$ 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 the AVM 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.

Complex data types, such as pointers, are initialized in the same way as with the random search. However memory graphs are not part of the optimization process and their shape remains fixed throughout each hill climbing phase. Once a random restart occurs, complex data types are re-initialized with new values. If initializing a pointer adds a new primitive type to the input space, fields in a dynamic data structure for example, they are added to the AVM's vector of arithmetic type inputs. Thus primitive type values within the memory graph *are* optimised by the AVM.

3.3. AVM+

Constructing random memory graphs may not be efficient when testing functions with pointer type inputs. It can easily result in large graphs for functions with dynamic data structures. At the same time the random search and AVM are unlikely to construct inputs that would reach the target in Fig. 2.

Therefore, the AVM+ uses techniques borrowed from Dynamic Symbolic Execution to lazily instantiate pointer type

[2] So far this has not been observed in practice.

```
void foo(int* w, int* x, int* y, int* z) {
  if( w ! = 0 )
    if( w == x )
      if( y == z )
        if( w == z ) {
          //TARGET
        }
}
```

Fig. 2. Example C code used to demonstrate the benefits of using a decision procedure for pointer inputs as opposed to constructing random memory graphs.

inputs instead. A high level description of the AVM+ algorithm is shown in Algorithm 1. The source code of the unit under test is instrumented to perform a pseudo-symbolic execution in parallel to the concrete execution. Since the AVM+ does not evaluate expressions symbolically it is not a classic symbolic execution. It simply re-writes operations over local variables as operations over input parameters (including globals). This often suffices for the purpose of solving pointer constraints and enables the AVM+ to use a very light-weight 'symbolic engine' that only performs symbolic assignments. Constraints over input parameters can still be collected to form a path condition describing the execution path taken by a concrete input.

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 bestSolution
```

AUSTIN uses the CIL [27] infrastructure for "C program analysis and transformation" to perform code instrumentation and its symbolic analysis. CIL provides a number of pre-defined modules, such as control and data flow analysis. It also offers an extensive API (in Ocaml) to traverse and manipulate the Abstract Syntax Tree (AST) of C code.

Whenever an input misses a target branch, the AVM+ first identifies the critical branching node n_c in the Control Flow Graph (CFG) where execution took an infeasible path with respect to the target. A critical branching node is a node in the CFG on which the target branch is (transitively) control dependent. The AVM+ then executes the function under test symbolically along the path taken by the concrete input, up to the last occurrence of n_c within the path (node n_c might be executed more than once in case of loops).[3]

During the symbolic execution, the AVM+ makes use of information from the concrete execution to solve the aliasing problem associated with static analysis. It also uses concrete values to replace symbolic values whenever an operation is beyond the scope of its symbolic execution engine. The result of the symbolic analysis is a (partial) path condition *pc* involving constraints over formal parameters of the function under test and global variables. A path condition describes the circumstances under which a CFG-path will be executed. The AVM+ uses the path condition to decide whether it can use its hill climb algorithm to solve the condition at node n_c, or whether it requires a constraint solver to instantiate pointer inputs instead.

Recall that the fitness function used by the hill climb algorithms is composed of a branch distance and approach level. However, the branch distance function is not suitable for conditions that describe constraints over memory locations. Its value would either be 0 (indicating that the condition has been satisfied) or 1 (indicating that the condition is not satisfied). This lack of gradient information deteriorates a guided search like the AVM into a random search.[4]

Therefore, the AVM+ uses a custom algorithm (shown in Algorithm 2) to solve pointer constraints arising from branching nodes in a CFG. Initially *pc* contains both, constraints over arithmetic type inputs as well as pointer type inputs. The AVM+ reduces *pc* to a *partial* path condition *ppc* by dropping all constraints over arithmetic type inputs. This includes constraints which contain a pointer dereference to a primitive type. Then, *ppc* is further simplified by removing all constraints which originate from non-critical branching nodes with respect to the current target branch.

The result is a path condition that only contains constraints over memory locations. Further, by using CIL, the AVM+ also ensures that these constraints 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. If the last constraint in *ppc* corresponds to the critical branching node n_c where execution diverged away from the target, the AVM+ uses a constraint solver. Otherwise, the node n_c contains constraints over arithmetic types that can be solved using a hill climber.

[3] The symbolic execution is inter-procedural for calls to functions whose source code was available during the instrumentation process.

[4] The same problem applies to branching conditions that contain boolean type variables.

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 *ppc* from *pc* by dropping all constraints over arithmetic types from *pc*
Trim *ppc* by removing all non-critical branching nodes to generate *ppc′*
Invert the binary operator ($\in\{=, \neq\}$) of the last constraint in *ppc′*
for all constraints c_i in *ppc′* **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**
      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**
      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 currentSolution

Assume the AVM+ needs to use its pointer constraint solver. To avoid following the same path in the next iteration, the AVM+ *inverts* the condition of the last constraint in *ppc* to generate *ppc′*. Finding an input that satisfies this new path condition will thus get 'closer' to the target branch in control flow terms. The procedure for finding such an input is described in the remainder of this section.

Table 1
Test subjects used to evaluate the different test data generation algorithms within AUSTIN. The LOC have been calculated using the sloccount tool [31] in its default setting.

Test subject	Num. of functions tested	Num. branches tested	LOC
binary tree	6	38	828
binheap	4	22	802
dllist	13	134	1244
gimp	28	292	867
spice	2	142	269

The AVM+ generates an equivalence graph of symbolic variables from *ppc′*. The equivalence relationship between symbolic variables is defined by the '=' operators in *ppc′*. 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 from Algorithm 2), and always contains a special node to represent the constant `NULL`. For each constraint c_i in *ppc′*, the symbolic variables involved in c_i are extracted. The AVM+ then checks if the symbolic variables are already contained within the nodes of the equivalence graph. If they are not, a new node is added for each 'missing' symbolic variable.

Given the node (s) representing the symbolic variables in c_i, the AVM+ 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, the AVM+ 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 *ppc′*.

For each feasible constraint in *ppc′*, the AVM+ 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, the AVM+ 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 the AVM+ 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.

4. Comparing AVM and AVM+

The aim of this section is to investigate what difference the constraint solver in the AVM+ makes in practice, compared to constructing random memory graphs in the AVM. We therefore compared the three search algorithms in AUSTIN, the random search, AVM and AVM+, on a mix of open source programs. Table 1 summarizes the programs and functions tested. The random search was simply used as a sanity check.

The experiments were performed on 628 branches, drawn from two different C programs[5] and common data structure implementations, taken amongst others from SGLIB [30]. `gimp` is an image editor and `spice` an analogue circuit simulator. Both programs are open source. `binheap`, `dllist` and `tree` are data structure implementations of a binary heap, doubly linked list and binary tree respectively.

Each search for test data was performed 30 times. For `gimp` and `spice`, if test data was not found to cover a branch after 30,000 fitness evaluations, the search was terminated. The fitness budget for the three data structure implementations was set to 100 evaluations per branch.

Serendipitous coverage, *i.e.*, branches covered by accident during the test data generation process, was ignored for the AVM and AVM+, so that a distinct search was carried out for every branch. The success or failure of each search was recorded, along with the number of fitness evaluations required to find the test data. The 30 runs were performed using an identical list of fixed seeds for random number generation, so as to provide a basis for assessment with tests for statistical significance. Such tests are necessary to provide robust results in the presence of the inherently stochastic behaviour of the search algorithms.

4.1. Evaluation

Fig. 3 shows the average branch coverage achieved by the three search algorithms for the open source functions from `gimp` and `spice`. Compared to the random search, both the AVM and AVM+ algorithms achieve a higher level of coverage. A branch was counted as covered if it has been executed at least once during the 30 repeat test data generation trials.

We will now explain why the AVM and AVM+ achieve identical levels of branch coverage. For `gimp` and `spice` all formal parameters of pointer type are either pointers to data structures or pointers to an integer. Many `gimp` data structures also contain one or more fields that represent pointers to other data structures. However, all pointer operations in the code are limited to accessing data by dereferencing a pointer. Subsequently, the code only contains pointer constraints of the form `p != 0`, where `p` denotes a pointer input. Furthermore, these constraints are not explicitly captured in guard statements, but one can view them as a form of precondition to the functions under test. For example, passing a `NULL` pointer to these functions will cause a segmentation fault.

AUSTIN does not have the ability to analyse stack traces. Therefore, any preconditions to a function have to be specified manually by using auxiliary functions provided by AUSTIN. For example, to specify that a pointer `int* p` must not be `NULL`, one would write

```
Austin__Assume(1, p != (void*)0);
```

Since the functions tested for `gimp` and `spice` both assume all pointers point to valid memory locations, we formulated these assumptions as a list of preconditions. In effect this 'fixes' the shape of any data structure, and thus the search algorithms simply have to optimise arithmetic values. The random search, AVM and AVM+ all satisfy preconditions in the same manner, hence there cannot be any difference in performance between the AVM and AVM+ for `gimp` and `spice`.

Fig. 4 shows the average branch coverage achieved by AUSTIN's algorithms for the data structure implementations. For these functions the AVM and random search perform equal, and the AVM+ achieves a lower coverage than both.

Interestingly, for the data structure implementations, a decision procedure-based approach to handling pointers appears to perform worse than generating random memory graphs. Examination of the results revealed that the problem again occurs in code that does not contain explicit guard statements for pointers. Without these, the approach described in Algorithm 2 is not able to construct a complete path condition which, in turn, would be used as a basis for instantiating pointer inputs.

Consider the example from Fig. 5. The function takes a pointer, to a pointer, to a data structure (`**first`) and another pointer to a data structure (`*second`) as input. Further assume that no, or an incomplete, precondition is specified. The AVM+ will not be able to cover any branches in that function. By default the function will be executed with all pointers set to `NULL`, causing a segmentation fault. Since AUSTIN is not able to identify the cause of such errors, it will never try to assign a value besides `NULL` to the input `first`.

A random memory graph generation technique on the other hand is able to cover both branches with a high probability. Consequently, in some cases it is able to achieve a higher branch coverage. Even though care was taken to specify appropriate preconditions for the functions, dereferencing non-initialized pointers was still a problem and the cause for the low coverage achieved by the AVM+ method.

Finally, for the data structure implementations tested, there appears to be no difference in using a guided search for arithmetic types instead of a random search. Both achieve the same level of coverage. This can be explained by the fact that branching conditions mainly contain constraints over pointer inputs. Recall that the AVM and random search initialize pointers in the same manner. When conditions contain constraints over arithmetic inputs these conditions are often easily satisfied, even by a random search. An example of of such a condition can be seen in Fig. 6, taken from the binary tree implementation.

Nevertheless, one would expect the guided search to be more *efficient* than a random search, even for conditions such as the one shown in Fig. 6. We use the number of fitness evaluations as a measure of efficiency. It reports how many different test data we had to generate in order to achieve the reported branch coverage. Fig. 7 shows the average number of fitness evaluations required by the random search, AVM and AVM+ for functions from the data structure implementations. The data has been normalised with respect to the random search, as this serves as a baseline for comparison.

Let us first consider the efficiency of the AVM+. For functions where the AVM+ can make use of its constraint solving algorithm it is more efficient than either the random search or the AVM. The AVM+ is only less efficient for functions that do not check for `NULL` pointers and where no adequate precondition exists. Let us now examine the figures for the AVM. Contrary to our expectation the AVM is overall *less* efficient than the random search. This means conditions, such as the one shown in Fig. 6, are more easily satisfied by a random search than a hill climber.

Generalizing this example, consider a condition of the form $x < y$ where both x and y are random variables denoting integers. Arcuri [32] showed that the probability of $x < y$ is

$$\frac{1}{2} - \frac{1}{2n}$$

where n is the problem size. In our case n represents the valid value range of a 32-bit integer. Thus, the random search has a high chance of satisfying this condition in two trials $\left(\frac{1}{\frac{1}{2}-\frac{1}{2n}} \approx 2\right)$. For the AVM, either the starting point satisfies the condition $x < y$, or, the search is guided by the branch distance function. Arcuri [32] shows that the expected time for the AVM to satisfy this condition is thus $O(\log n)$.

4.1.1. Statistical significance

To check if there is a statistically significant difference in the branch coverage achieved by either the random search, AVM or AVM+ we used a one-sided Wilcoxon test and specified a confidence level of 99%. We grouped all the coverage data for the

[5] Programs were chosen arbitrarily. However, all branches used for this study have been used to evaluate search-based testing techniques in the past [28,29].

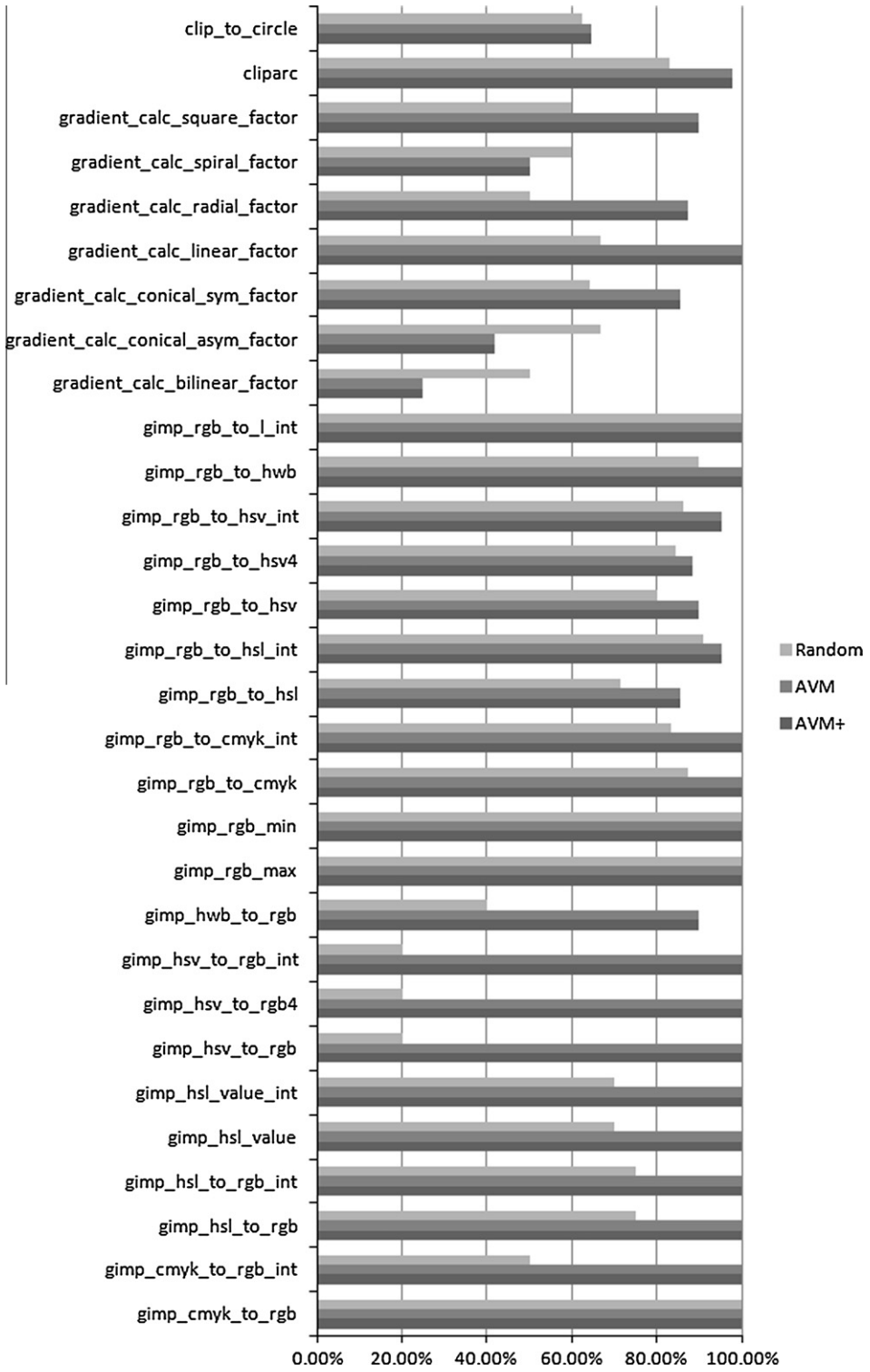

Fig. 3. Branch coverage achieved by the random search, AVM and AVM+ in AUSTIN for the open source functions from `gimp` and `spice`. The horizontal axis denotes the percentage of branches covered, averaged over 30 trials.

subjects from Table 1 into two samples. Then we compared the AVM against the random search, the AVM+ against the random search, and finally the AVM against the AVM+. We chose the Wilcoxon test over a t-test because the result of the Shapiro-Wilk normality test indicated that the data does not follow a normal distribution (given a confidence level of 99%). For the AVM-AVM+ samples we obtained a p-value of 3.881×10^{-13}, for the random-AVM+ samples a p-value of 0.001003, and for the random-AVM samples a p-value of 5.798×10^{-08}. Each test was performed with the statistical tool R [33].

In all cases our null hypothesis is that there is no difference in branch coverage between the random search, AVM and AVM+.

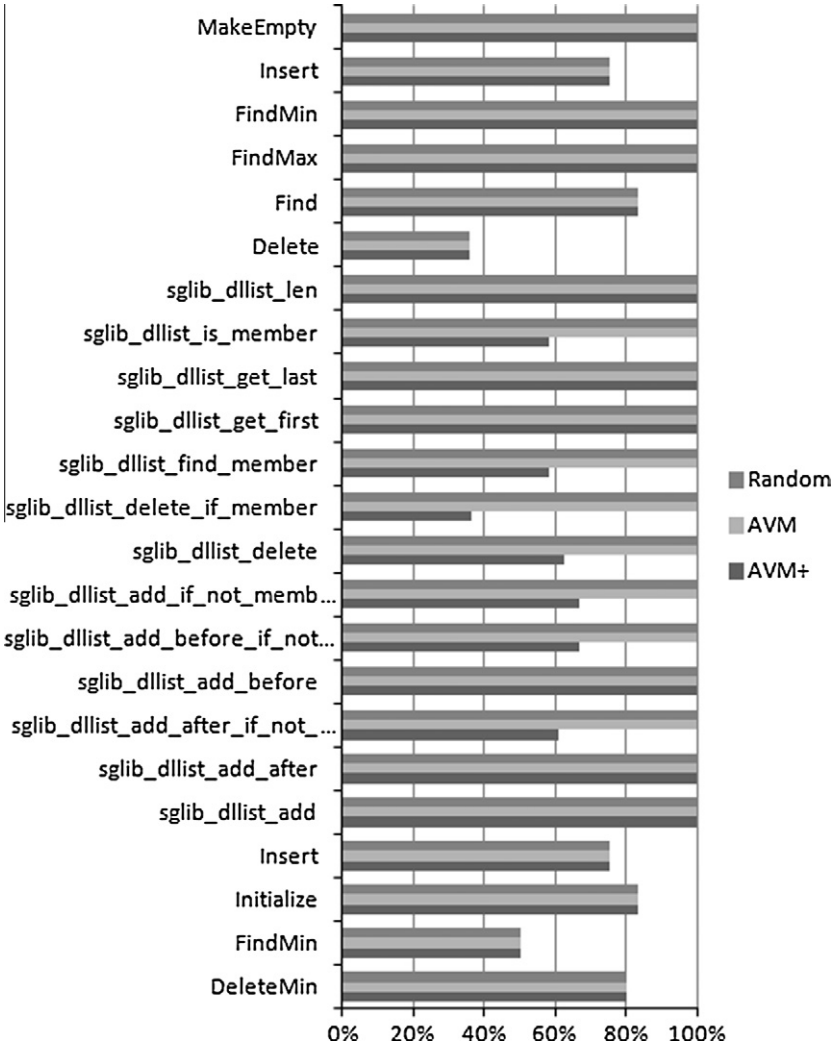

Fig. 4. Branch coverage achieved by the random search, AVM and AVM+ in AUSTIN for the data structure implementations. The horizontal axis denotes the percentage of branches covered, averaged over 30 trials.

```
void sglib_dllist_concat(dllist **first, dllist *second) {
    if ((*first) == ((void *) 0))
       ...
}
```

Fig. 5. Example used to illustrate how Algorithm 2 might be outperformed by random generation of memory graphs.

```
SearchTree Insert( ElementType X, SearchTree T ) {
    ...
    if( X < T->Element )
       ...
}
```

Fig. 6. Example illustrating a condition that is easily satisfiable with a random search, but for which a hill climber might be less efficient.

AVM vs Random: Our alternative hypothesis is that the AVM achieves a higher branch coverage than the random search. For this comparison we obtained a p-value of 0.001582 ($p \leqslant 0.01$), indicating that we can reject the null hypothesis in favour of our alternative hypothesis.

AVM+ vs Random: Again, our alternative hypothesis is that the AVM+ achieves a higher branch coverage than the random search. Using the Wilcoxon test we obtain a p-value of 0.0836 ($p > 0.01$), indicating that we cannot reject the null hypothesis (*i.e.*, there is no difference, on average, in branch coverage). Here the result is clearly influenced by the low coverage achieved by the AVM+ for the data structure implementations.

AVM vs AVM+: Finally, our alternative hypothesis is that the AVM+ achieves a higher branch coverage than the AVM. We obtain a p-value of 0.9395, indicating once more that we cannot reject the null hypothesis.

4.2. Comments

Surprisingly we did not find any statistically significant difference in branch coverage between the AVM+ and random search, or the AVM+ and the AVM implemented in AUSTIN. For the functions from `gimp` and `spice` the hill climbers are, on average, better than a random search. However for the data structures there is either no difference (AVM) or the coverage is worse (AVM+).

Caution is required when interpreting the statistical evaluation of the findings. One has to question whether it makes sense to evaluate different techniques across the set of all possible functions, or whether one should focus on comparisons within a certain application domain. Clearly in some cases the AVM+ with its decision procedure for pointer inputs will outperform the AVM, both in

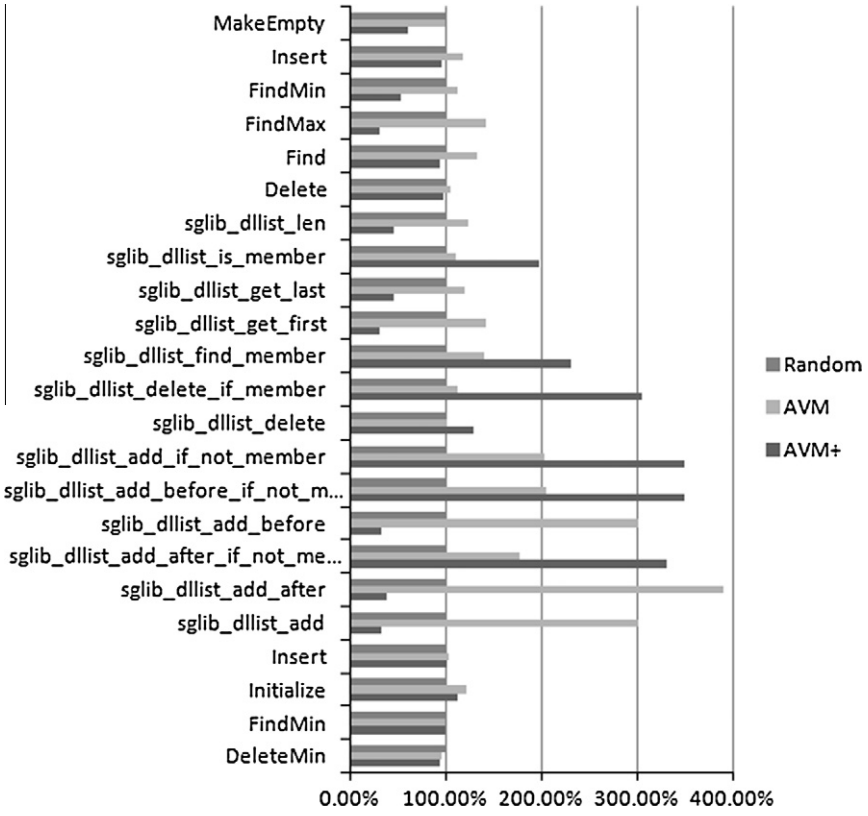

Fig. 7. Normalised average number of fitness evaluations required by the random search, AVM and AVM+ for the data structure implementations.

terms of effectiveness and efficiency. It is easy to come up with synthetic examples to prove this.

We already explained why the AVM and AVM+ perform equally for the functions from `gimp` and `spice`. Since the shape of any input data structure is fixed by preconditions, the AVM and AVM+ end up only optimising arithmetic type values, for which both techniques are identical. Where the AVM+ performed worse, *i.e.* for the data structure implementations, the cause was a lack of guard statements for pointer inputs. Without these, the decision procedure within the AVM+ is not able to build up a path condition and thus solve pointer constraints. Other tools using similar techniques to deal with pointers, such as CUTE [21] and CREST [34], suffer from the same problem.

A further threat to the validity of the findings arises from how the data structure functions were tested. We generated test data for each function in isolation. Thus, invalid input may be passed to the functions that can cause segmentation faults or infinite loops. When testing data structures it is recommended to either test the code by generating sequences of method calls, or, by solving data structure invariants [21].

The first option was not feasible because AUSTIN is not able to generate method call sequences. To use the second option one has to rely on 'helper' functions, such as SGLIB's `check_consistency` methods to encapsulate data structure invariants. A limitation of AUSTIN is that it is not able to use the return value of these functions to guide the test data generation process towards producing valid inputs. This is because the hill climbers and decision procedures only consider constraints arising from critical branching nodes; a general weakness in the underlying fitness function used by the AVM and AVM+. Consequently the AVM+ cannot use functions such as `check_consistency` to build up data structures. Future work will consider how AUSTIN can be improved to make use of such functions.

5. Comparing AUSTIN against state-of-the-art

The objective of this section is to investigate the effectiveness and efficiency of AUSTIN's hill climbers compared to a state-of-the art Evolutionary Testing Framework (ETF). The ETF was developed as part of the multidisciplinary European Union-funded research project EvoTest [6] (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 http://evotest.iti.upv.es/.

5.1. Empirical study

In order to compare AUSTIN's hill climbers against the ETF we considered 8 C functions that are summarised in Table 2. They were taken from three embedded software modules and had been

Table 2
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 [35] in its default setting.

Case study	LOC	Functions tested	Software module
B	18,200	02, 03, 06	Adaptive headlight control
C	7449	07, 08, 11	Door lock control
D	8811	12, 15	Electric window control

Table 3
Test subjects. The LOC have been calculated using the CCCC tool [35] 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
02	919	420	14	80
03	259	142	12	38
06	58	36	6	14
07	85	110	11	27
08	99	76	7	29
11	199	129	4	15
12	67	32	9	3
15	272	216	4	28

selected by Berner & Mattner Systemtechnik GmbH to form part of the evaluation of the ETF within the EvoTest [6] project. The functions had been chosen to provide a representative sample of real world automotive code, with particular attention paid to the number of branches and nesting level. Table 3 gives a breakdown of relevant metrics for the selected functions.

As with `gimp` and `spice`, any functions that contain a pointer input assume the pointer has been initialized. This constraint was encoded in all functions for both, the ETF and AUSTIN. Therefore there will be no difference in coverage between AUSTIN's AVM and AVM+. Furthermore, the study will not examine differences in the way pointer inputs are treated within the ETF compared to AUSTIN. Hence we chose to carry out the experiments using AUSTIN's AVM+. In the interest of readability we will refer to AUSTIN's AVM+ simply as AUSTIN henceforth.

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.*

5.2. Experimental setup

The data for the experiments with the ETF on the functions listed in Table 3 had already been collected for the evaluation phase of the EvoTest project [6]. This sub-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 [28]. 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.

5.3. Evaluation

Effectiveness of AUSTIN. Fig. 8 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. This time we used a two-sided Wilcoxon test so that we are able to detect reductions as well as improvements in coverage. At a confidence level of 99% we obtained a p-value of 0.9236, indicating that we cannot reject the null hypothesis, and therefore conclude that AUSTIN and its AVM+ are as effective as the ETF in achieving branch coverage.

Visually this is confirmed by Fig. 8, where for all except one function AUSTIN is either better or equal to the ETF. Function 08 is interesting because it is the only function for which AUSTIN performs 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. As a result the fitness landscape in which AUSTIN's hill climber operates contains both, large plateaus offering no guidance to the search, and local optima. A global search algorithm, such as the GA used in the ETF is able to explore a wider region of the search space in parallel. It is therefore better suited to escape from plateaus (and local optima) than a hill climber.

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 increased fitness budget gives the search

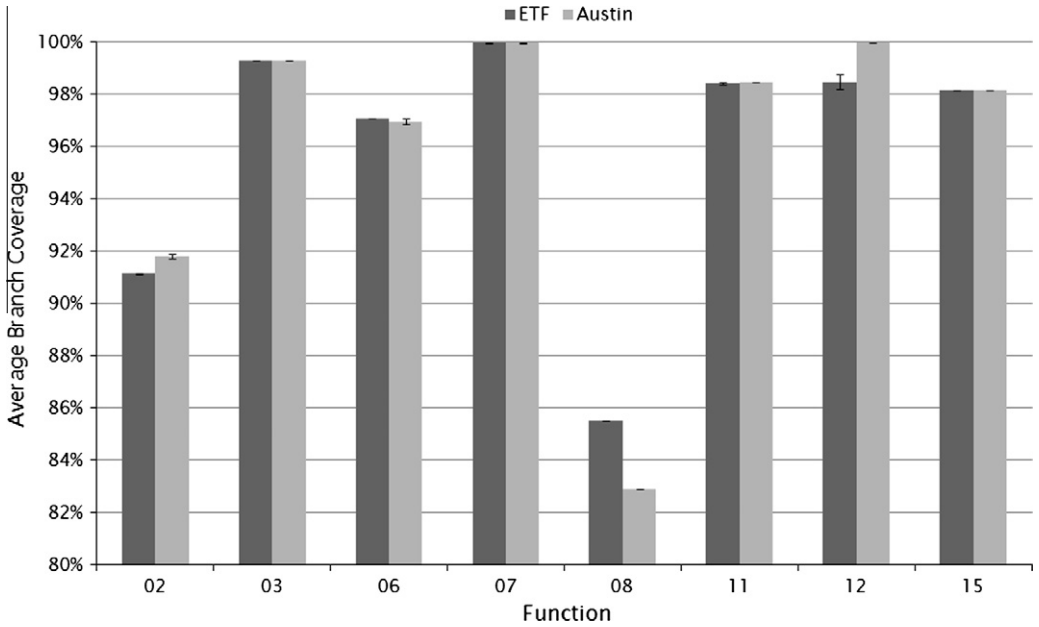

Fig. 8. Average branch coverage of the ETF versus AUSTIN. The vertical axis shows the coverage achieved by each tool in percent, for each of the functions shown on the horizontal axis. The error bars show the standard error of the mean.

more opportunity to explore a wider region of the search space in case it gets trapped on plateaus or local optima.

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 Fig. 8. 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. Fig. 9 shows the average number of fitness evaluations used by both ETF and AUSTIN when trying to achieve coverage of each function. Since the difference in achieved coverage between the two tools was generally very small, it was neglected when comparing their efficiency. In order to test the second hypothesis we also used a two-tailed Wilcoxon test. The test reports a p-value of 0.218 (at a confidence level of 99%), indicating that we cannot reject the null hypothesis.

Visual inspection of the results confirm that overall AUSTIN is as efficient as the state-of-the-art ETF. AUSTIN only requires more fitness evaluations for function 08, while it is more efficient for six of the eight functions.

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 with the ETF, we used the same input domain reduction as previously described. For each branch the random search was allowed at most

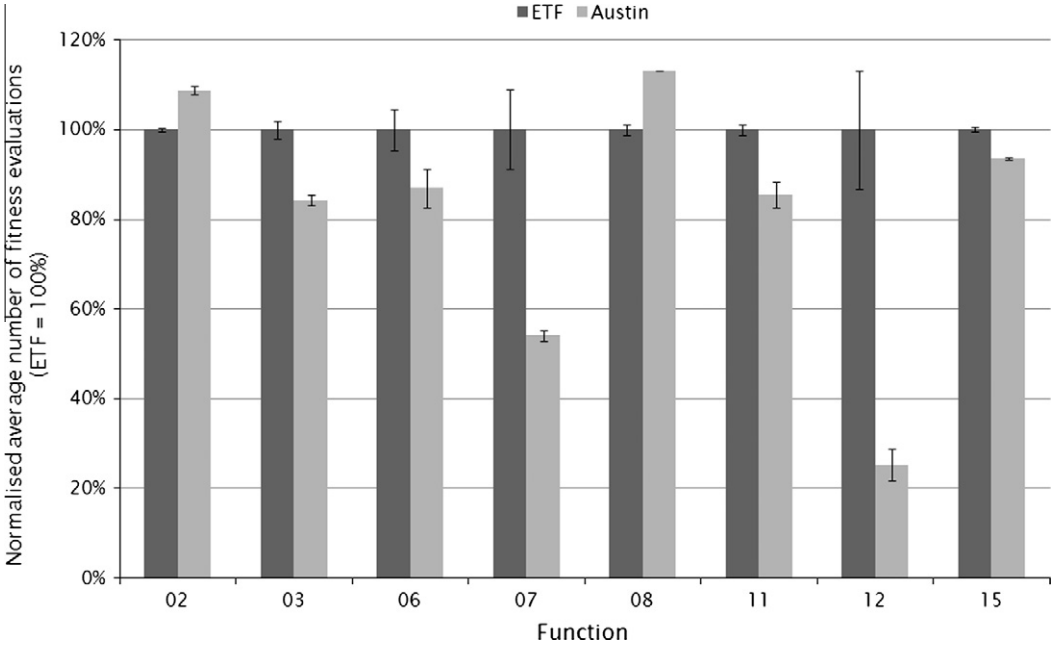

Fig. 9. Average number of fitness evaluations (normalised) for ETF versus AUSTIN. The vertical axis shows the normalised average number of fitness evaluations for each tool relative to the ETF (shown as 100%) for each of the functions shown on the horizontal axis. The error bars show the standard error of the mean.

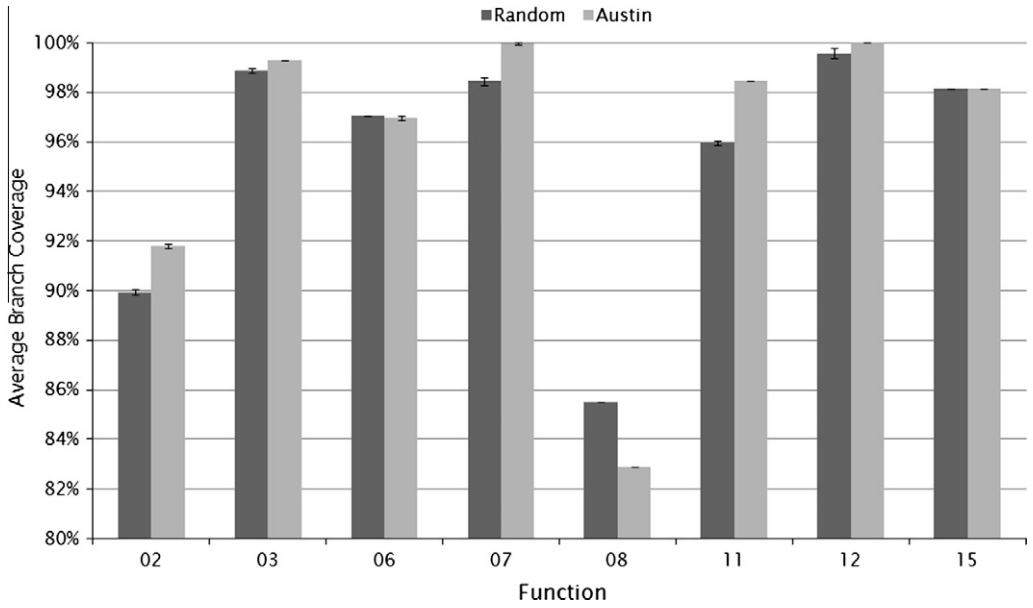

Fig. 10. Average branch coverage of random search versus AUSTIN. The vertical axis shows the coverage achieved by each tool in percent, for each of the functions shown on the horizontal axis. The error bars show the standard error of the mean.

10,000 evaluations. Any branches covered serendipitously by the random search during the testing process were counted as covered and removed from the pool of target branches.

Since we expect AUSTIN to outperform a random search, we used a one-sided Wilcoxon test. The test returned a p-value of 2.427×10^{-06} indicating that we can reject the null hypothesis that AUSTIN (AVM+) is only equally effective as a random search. Again this is visually confirmed by Fig. 10. Only function 08 stands out where AUSTIN achieves less coverage than random. However, we already explained why function 08 is difficult to test for a hill climb algorithm. As with the GA, a random search is better at exploring the search space, though it is not good at *exploiting* it once it is in near an optimum. Because the hill climb search gets trapped too often for function 08, even the random search achieves a higher branch coverage. Recall though, that given a larger fitness budget as mentioned previously, AUSTIN is able to improve its coverage of function 08 to an equal level with the ETF and random search.

Comparing AUSTIN's efficiency with that of a random search using a one-sided Wilcoxon test we obtain a p-value of 2.2×10^{-16}. This value also indicates that we can reject the null hypothesis stating AUSTIN and the random search are equally efficient. Visually this is confirmed by Fig. 11.

5.4. 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

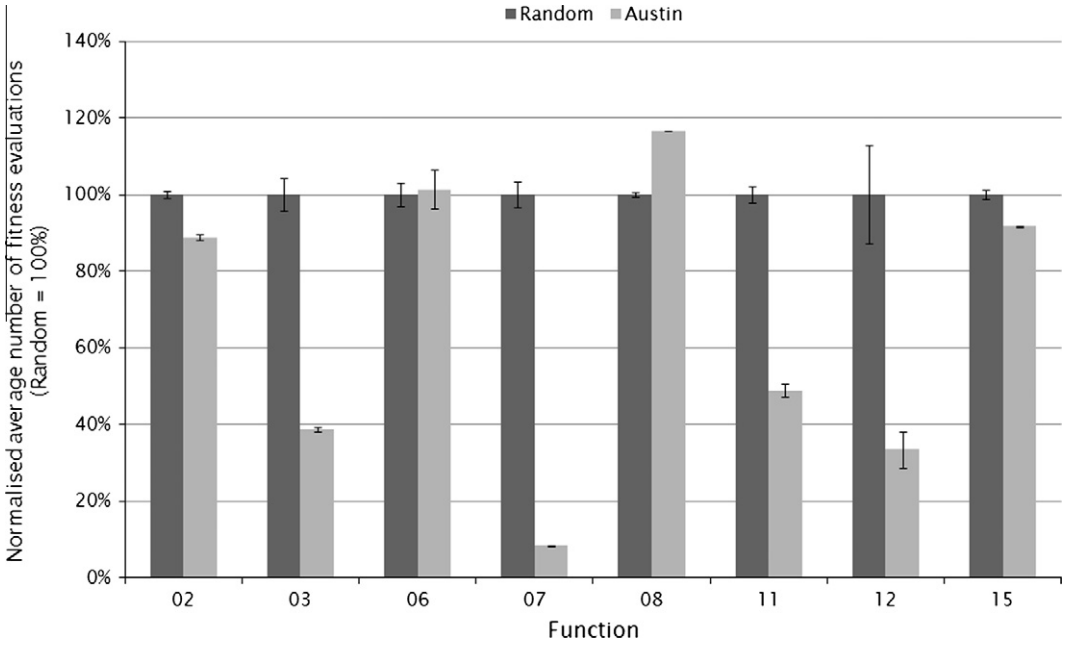

Fig. 11. Average number of fitness evaluations (normalised) for random versus AUSTIN. The vertical axis shows the normalised average number of fitness evaluations for each tool relative to the random search (shown as 100%) for each of the functions shown on the horizontal axis. The error bars show the standard error of the mean.

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 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.

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. This was a subjective choice made by the company and therefore caution is required before making any claims as to whether these results would be observed on other functions.

6. Conclusion

This paper has introduced and evaluated the open source, search-based testing tool named AUSTIN. It supports a random search, a hill climber in the form of the Alternating Variable Method, and a hill climber augmented with symbolic execution to handle constraints over pointer inputs to a function. Test data is generated by AUSTIN to achieve branch coverage for C functions.

We first evaluated a decision procedure for dealing with pointer inputs, introduced in this paper, on a set of functions taken from open source programs. We hypothesised that this decision procedure is more effective and efficient than randomly generating memory graphs. A Wilcoxon test revealed that we cannot reject the null hypothesis that both are equally effective and efficient. This result is discussed in detail in the paper.

We also present a larger empirical study that compares the hill climbers in AUSTIN with a closed source, state-of-the-art Evolutionary Testing Framework. Here we hypothesised that AUSTIN is at least as effective and more efficient than the ETF in generating branch adequate test data. In both cases a Wilcoxon test was used to asses our hypotheses. Results indicate that AUSTIN is equally effective and efficient as the ETF.

Acknowledgments

We would like to thank Bill Langdon for his helpful comments and Arthur Baars for his advice on the Evolutionary Testing Framework. Zhang kindly provided the two figures on growth trends in publications on SBST used in this paper. Kiran Lakhotia is funded through the European Union Project FITTEST (ICT-2009.1.2 no 257574). 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] K. Lakhotia, M. Harman, H. Gross, Austin: a tool for search based software testing for the C language and its evaluation on deployed automotive systems, in: International Symposium on Search Based Software Engineering, 2010, pp. 101–110.
[2] W. Miller, D.L. Spooner, Automatic generation of floating-point test data, IEEE Transactions on Software Engineering 2 (3) (1976) 223–226.
[3] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, 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.
[4] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[5] K. Lakhotia, P. McMinn, 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.
[6] H. Gross, P.M. Kruse, J. Wegener, T. Vos, Evolutionary white-box software test with the evotest framework: a progress report, in: ICSTW '09, Washington, DC, USA, 2009, pp. 111–120.
[7] Radio Technical Commission for Aeronautics, RTCA DO178-B software considerations in airborne systems and equipment certification, 1992.
[8] M. Harman, J. Clark, Metrics are fitness functions too, in: 10th International Software Metrics Symposium (METRICS 2004), IEEE Computer Society Press, Los Alamitos, California, USA, 2004, pp. 58–69.
[9] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[10] E. Alba, F. Chicano, Observations in using parallel and sequential evolutionary algorithms for automatic software testing, Computers & Operations Research 35 (10) (2008) 3161–3183.
[11] E. Alba, F. Chicano, Software testing with evolutionary strategies, Proceedings of the 2nd Workshop on Rapid Integration of Software Engineering Techniques, vol. 3943, Springer, Heraklion, Crete, Greece, 2005, pp. 50–65.
[12] R. Sagarna, A. Arcuri, X. Yao, Estimation of distribution algorithms for testing object oriented software, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07), IEEE, Singapore, 2007, pp. 438–444.
[13] R. Blanco, J. Tuya, E. Daz, B.A. Daz, A scatter search approach for automated branch coverage in software testing, International Journal of Engineering Intelligent Systems (EIS) 15 (3) (2007) 135–142.
[14] 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.
[15] R. Lefticaru, F. Ipate, Functional search-based testing from state machines, in: Proceedings of the First International Conference on Software Testing, Verfication and Validation (ICST 2008), IEEE Computer Society, Lillehammer, Norway, 2008, pp. 525–528.
[16] A. Windisch, S. Wappler, J. Wegener, Applying particle swarm optimization to software testing, in: Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07), ACM, London, England, 2007, pp. 1121–1128.
[17] E. Díaz, J. Tuya, R. Blanco, J.J. Dolado, A Tabu search algorithm for structural software testing, Computers & Operations Research 35 (10) (2008) 3052–3072.
[18] S. Ali, L.C. Briand, H. Hemmati, R.K. Panesar-Walawege, A systematic review of the application and empirical investigation of search-based test-case generation, IEEE Transactions on Software Engineering, in press.
[19] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, ACM SIGPLAN Notices 40 (6) (2005) 213–223.
[20] C. Cadar, D.R. Engler, Execution generated test cases: how to make systems code crash itself, Proceedings of the 12th International SPIN Workshop on Model Checking Software, San Francisco, CA, USA, August 22–24, 2005, vol. 3639, Springer, 2005, pp. 2–23.
[21] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: ESEC/SIGSOFT FSE, ACM, 2005.
[22] N. Tillmann, J. de Halleux, Pex white box test generation for .NET, in: B. Beckert, R. Hähnle (Eds.), TAP, vol. 4966, Springer, 2008, pp. 134–153.
[23] J.C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.
[24] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.
[25] A. Arcuri, It does matter how you normalise the branch distance in search based software testing, in: ICST, 2010, pp. 205–214.
[26] Free Software Foundation, Gcc, the gnu compiler collection, 2009. <http://gcc.gnu.org/>.
[27] G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, CIL: intermediate language and tools for analysis and transformation of C programs, Lecture Notes in Computer Science 2304 (2002) 213–228.

[28] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, J. Wegener, The impact of input domain reduction on search-based test data generation, in: I. Crnkovic, A. Bertolino (Eds.), ESEC/SIGSOFT FSE, ACM, 2007, pp. 155–164.
[29] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: local, global, and hybrid search, IEEE Transactions on Software Engineering 36 (2) (2010) 226–247.
[30] M. Vittek, P. Borovansky, P.-E. Moreau, A simple generic library for c, in: Proceedings of 9th International Conference on Software Reuse, Tur in Reuse of Off-the-Shelf Components, Italy, Springer, 2006, pp. 423–426.
[31] D.A. Wheeler, More than a gigabuck: Estimating GNU/Linux's size, June 2001. <http://www.dwheeler.com/sloc/>.
[32] A. Arcuri, Full theoretical runtime analysis of alternating variable method on the triangle classification problem, in: International Symposium on Search Based Software Engineering, 2009, pp. 113–121.
[33] R Development Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2011, ISBN 3-900051-07-0. <http://www.R-project.org>.
[34] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: Automated Software Engineering (ASE 2008), IEEE, 2008, pp. 443–446.
[35] 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.