Handling Dynamic Data Structures in Search Based Testing

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

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

Phil McMinn
University of Sheffield
Regent Court,
211 Portobello St, Sheffield
S1 4DP, UK
p.mcminn@dcs.shef.ac.uk

ABSTRACT

There has been little attention to search based test data generation in the presence of pointer inputs and dynamic data structures, an area in which recent concolic methods have excelled. This paper introduces a search based testing approach which is able to handle pointers and dynamic data structures. It combines an alternating variable hill climb with a set of constraint solving rules for pointer inputs. The result is a lightweight and efficient method, as shown in the results from a case study, which compares the method to CUTE, a concolic unit testing tool.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging
General Terms: Algorithms, Experimentation, Measurement, Performance.
Keywords: Automated test data generation, concolic testing, symbolic execution

1. INTRODUCTION

Search Based Testing (SBT) [15] uses search based optimization techniques [6] to formulate the test data generation problem as a search problem. The search space is the space of possible inputs to the program under test. The search is guided by a fitness function that captures the particular test adequacy criterion of interest. SBT has proved to be effective partly because it has a wealth of optimization techniques upon which to draw and because the generic nature of the approach allows it to be adapted to a wide range of test data generation problems; in principle, all that is required to adapt a search based technique to a different test adequacy criterion is a new fitness function. This paper is concerned with search based structural testing [21].

Previous work on search based test data generation has generally considered the input to the program under test to be a fixed–length vector of input values, making it a well–defined and fixed–size search space, or has been based on data-flow-analysis and backtracking [13]. However, such analysis is non–trivial in the presence of pointers and points–to analysis is computationally expensive. The approach presented in this paper incorporates elements from symbolic execution. Symbolic execution is a source code analysis technique in which program inputs are represented as symbols and program outputs are expressed as mathematical expressions involving these symbols [12]. Symbolic execution mainly suffers from two problems: 1. the path explosion problem; 2. limited ability to reason about certain code constructs such as system objects, external function calls, *etc.* The first of these obstacles can be partially overcome by a *lazy initialization* technique [11] in the presence of dynamic data structures. During lazy initialization, pointers or data structures are non–randomly assigned $NULL$, and only initialized when required during the symbolic execution. Concolic testing [19, 5, 2] can serve as a means to alleviate the second problem.

The search algorithm chosen in this paper is based on the alternating variable hill climb algorithm introduced by Korel [13]. Hill climbing has been shown to outperform other search algorithms for structural test adequacy criteria [7], in particular branch coverage, which is the criteria of interest in this paper. A hill climb algorithm requires the definition of a *neighbourhood* for any solution and the search can only be effective if this neighbourhood is kept relatively small. However, this is not guaranteed in the case of pointers and dynamic data structures, which can potentially cause an explosion in size.

This paper addresses this issue. It presents an algorithm that combines a lazy initialization technique adapted from symbolic execution with search based testing.

The primary contributions of the paper are as follows:

- The introduction of an algorithm that combines search based testing and symbolic execution.
- The adaption of a lazy initialization approach for dynamic data structures within search based testing, thereby extending the applicability of these techniques.
- An experimental study, the results of which demonstrate the effectiveness of the algorithm in terms of branch coverage achieved, both for synthetic examples and for several real world test data generation problems drawn from open source and industry.

The rest of the paper is organized as follows. Section 2 provides the background information for the work in this pa-

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

per. It contains a description of the hill climb algorithm on which our approach is based, the principles of symbolic execution and a brief section on concolic testing. The extended hill climb algorithm proposed is described in Section 3. An example and details of input representation and generation, as well as the symbolic execution performed are discussed. We validated a prototype implementation of our algorithm with a set of case studies. These are presented and discussed in Section 4. Section 5 describes related work, while Section 6 concludes and presents an outlook on future work.

2. BACKGROUND

2.1 Hill Climb

A hill climb is a metaheuristic search technique, also classed as local search [13]. An input vector to a function is constructed with random values. For each element in the vector a set of exploratory moves are made. If one of these moves results in an improved value for the objective function, the search continues in its current direction with ever increasing step sizes. When no further improvements can be found for an element, the search continues exploring the next element in the vector. Once the entire input vector has been exhausted, the search recommences with the first element if necessary. In case the search stagnates, *i.e.* no move leads to an improvement, the search restarts at another randomly chosen location in the search space. This is designed to overcome local optima and enable the hill climb to explore a wider region of the search space.

The speed with which the search navigates around the search space is determined by the step size taken to explore the neighbourhood of an input. For this paper, the following formula was used to calculate the size of the move: $m_i = s^{it} * dir * acc_i$, where $dir \in \{-1, 1\}$, m_i is the move for the i^{th} input variable, s the repeat base (2 by default) and it the repeat iteration of the current move, and acc_i the accuracy of the i^{th} input variable. Choosing a high accuracy for real types (*i.e.* `floats, doubles`) can significantly slow down the search, thus it is usually kept at 1 to 2 decimal places.

2.2 Symbolic Execution

Symbolic execution can be viewed as a mix between testing and formal methods; a testing technique that provides the ability to reason about its results, either formally or informally [12].

During symbolic execution, a program is executed statically using a set of symbols instead of dynamically with instantiations of input parameters. The execution can be based on forward or backward analysis. During backward analysis, the execution starts at the exit node of a Control Flow Graph (CFG), whereas forward analysis begins at the start node of a CFG. Both type of analyses produce the same execution tree, however forward analysis allows faster detection of infeasible paths, which is a common application of symbolic execution [10].

Symbolic execution uses a path condition (pc) to describe the interdependency of program variables and input parameters along a path through a program. It consists of a combination of algebraic expressions and conditional operators. Any instantiation of input parameters that satisfies the pc will thus follow the path described by the pc. In the absence of preconditions, pc is always initialized to $true$, *i.e.* no assumptions about the execution flow are made.

Two types of statements are key during symbolic execution: *assignment* and *branching* statements. Assignments to variables are expressed as polynomial expressions of the symbols used. During assignment statements, the symbolic execution needs to evaluate the semantics of computational operators such as $+, -, *, /$. Branching statements, also known as *forking* statements, cause symbolic execution to generate two path constraints: one where the condition c in a branch statement is assumed $true$ and one where it is assumed $false$, thus generating $pc : c$ and $pc : \neg c$. Note that if the pc implies that the condition (or negated condition respectively) in the branching statement is $true$, symbolic execution can proceed without forking into two paths. If a pc is unsolvable, the path described by pc can be considered *infeasible*.

However, theorem provers are also the Achilles heels of symbolic execution, because they either poorly scale or are limited to linear constraints to name but a few problems. Without theorem provers, able to handle any kind of path constraint symbolic execution is useless for testing. Sometimes the nature of a program inherently *prevents* reasoning about path constraints, *e.g.* when it contains library calls to system functions or the program performs certain kinds of machine dependent operations [4]. Indeed certain functions (like the hash function described in [4]) are inherently designed to prevent reasoning about them.

2.3 Concolic Testing

Some of these limitations are addressed in concolic testing. Concolic testing is a technique that combines concrete execution of a program with symbolic execution [19, 5, 2]. The idea of concolic testing originates from Godefroid *et al.*'s Directed Automated Random Testing approach, DART [5]. DART is aimed at unit testing. First, a unit is executed with a randomly generated input vector. Symbolic constraints are collected along the path taken by the concrete execution, in effect executing a program both, symbolically and concretely in parallel. Once the program *halts*, *e.g.* by reaching the return statement of a unit, the last condition in a symbolic path constraint is inverted. If this new constraint is satisfiable, solving it with concrete inputs provides the input vector for the next execution of the unit, driving the program down a different path.

One of the principle strengths of concolic testing is the way in which concrete values are used to overcome many of the problems associated with symbolic execution. When symbolic constraints become too complex to solve, symbolic expressions are simply replaced by concrete values. In the case of DART, the tool developed is able to use a light weight linear constraint solver [1]. However, the limitations of many constraint solvers, especially in the presence of floating point arithmetic, means that concolic testing as performed by DART or CUTE [19] often deteriorates to a random search. This is also true for programs involving state variables [14].

The literature has compared the effectiveness of random testing with a search based testing approach with respect to structural test adequacy criteria [7, 9, 18, 22]. The results from these studies suggest that a search based approach such as hill climb is able to outperform simple random testing. However, due to the lack of support for dynamic input parameters such as pointers, a lot of this work is sometimes overshadowed by other forms of test data generation, such as

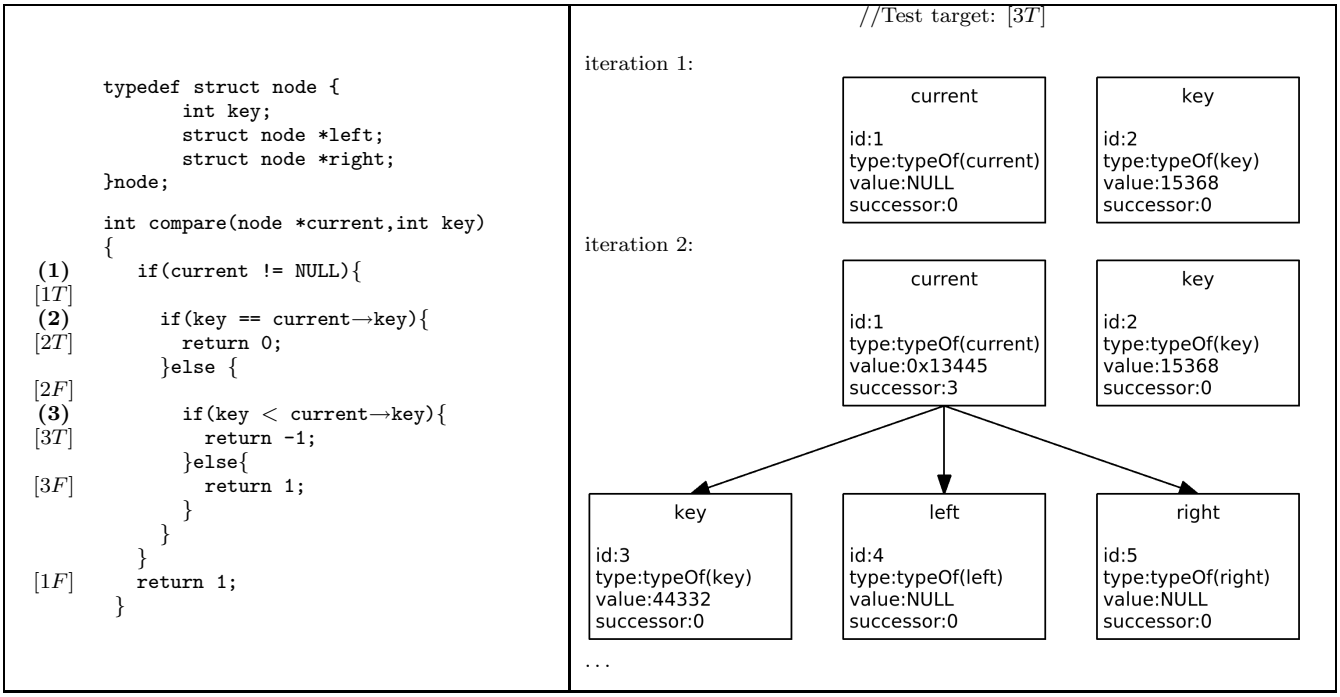

Figure 1: Example C code snippet. The numbers in the square brackets indicate the *true* and *false* branches of a predicate. The right column shows an example input map being constructed for the target branch $[3T]$.

concolic testing. While a comprehensive empirical investigation of the two different techniques is beyond the scope of this paper, the motivation was to take a step in this direction by advancing the capabilities of the current state-of-the-art search techniques, extending them so they can provide better support for dynamic data structures.

3. EXTENDED HILL CLIMB

The extended hill climb algorithm proposed in this paper augments the hill climb described in Section 2.1 with a set of simple constraint solving rules. We have implemented the proposed algorithm in a prototype tool (SYCON).

The proposed algorithm is best described with a small example before providing details of the different operations performed. Assume node $[3T]$ in Figure 1 is selected as the target.

SYCON first constructs a CFG of the program, with each branching node (*e.g.* `if` statement) being labelled. Further, each branch contains a list of critical branching nodes and their required outcome. Traversing an undesired edge of a critical node signals that the execution has taken a path which cannot lead to the target node.

The search starts by executing a randomly generated solution. Note that only primitive types are assigned random values, while pointer types are non–randomly set to $NULL$. The corresponding input vector is $< NULL, 15368 >$, with $current = NULL, key = 15368$. SYCON records the execution path taken by the solution ($< 1F >$). During evaluation, the objective function iterates over the execution trace. If the execution reached the target branch, the search stops. Else, SYCON retrieves the *id* of the critical branch where execution diverged away from the target, in this case node (1). SYCON performs a backward symbolic analysis from this node to first check that the variables appearing in the predicate can be expressed in terms of input symbols, and second, if the input symbols involved represent primitive or pointer types. Depending on the outcome of this analysis, the objective function either calculates a branch distance, or applies the constraint solving algorithm shown in Figure 4. The instrumentation used (described in Section 3.1) ensures that conditions only contain either pointer or primitive types but not both. Note that the constraint algorithm is *not* applied to pointers to primitive types. These pointers are dereferenced instead, so they can be used in the branch distance calculations.

Proceeding with the example, SYCON allocates sufficient memory (via `malloc`) to hold a variable of type `node` structure. The corresponding input vector (flattened) looks like $< 44332, NULL, NULL, 15368 >$, with $current \rightarrow key = 44332$, $current \rightarrow left = NULL$, $current \rightarrow right = NULL$, $key = 15368$. This input will follow the path $< 1T, 2F, 3F >$. At this point the search will start with the exploratory moves described in Section 2.1, while leaving any pointer inputs fixed:
$< 44333, NULL, NULL, 15368 >$
$< 44331, NULL, NULL, 15368 >$
$< 44330, NULL, NULL, 15368 >$
...
$< 11565, NULL, NULL, 15368 >$

3.1 Instrumentation

CIL [17] is used to first simplify (using various CIL-options) and then instrument the program under test. Achieving branch coverage on the CIL transformed code is equivalent to achieving MC/DC (Modified Condition/Decision Coverage) on the original code.

```
/*inputs:
* v = concrete input variable
* m = &v
* id = unique identifier for v
* offset = pointer to successor index for pointers
* typeOf(v) returns the type of v
*/
generateInput(m,id,offset,typeOf(v)):
if id ≥ inputMap.size() then
  if typeOf(v) = pointerType then
    *m := NULL;
  else
    *m := random(LB_v,UB_v);
    updateHCVector(m,id);
  end if
  inputMap.push_back(new obj(m,id,offset,typeOf(v)));
else
  v := getObject(&inputMap,id,offset,typeOf(v));
  if v = NULL then
    v := updateInputMap(&inputMap,id,offset,typeOf(v));
  end if
  if typeOf(v) = pointerType then
    if v→pointToNull = true then
      *m := NULL;
    else
      if v→targetNodeId = 0 then
        n := sizeOf(typeOf(v));
        {m_1,...,m_n} := malloc(n);
        *m := m_1;
        for j:=1 to n
          generateInput(m_j,id,(j-1),typeOf(*m_j));
      else
        *m := getTargetNode(v→targetNodeId)→value;
      end if
  else
    *m := getValue(v);
  end if
end if
S→add(m,typeOf(v));
```

Figure 2: Algorithm for input initialization. $inputMap$ is a vector storing node objects representing all formal parameters to the function under test. LB_v,UB_v are the lower and upper bounds of the data type returned by $typeOf(v)$ respectively. S is a symbolic map (see Section 3.3 for details).

3.2 Input Representation and Generation

All formal parameters of a unit under test are stored as node objects in an input vector called $inputMap$. Each node object contains a unique node identifier (id), which is kept throughout consecutive iterations, the symbolic variable associated with the input parameter, its data type and a pointer to a vector of successor nodes. Successor nodes represent members of a data structure, union, or fields of an enum type respectively. The data type associated with a node object can either be a primitive type or a pointer. The function $typeOf(v)$ returns the given type for an input (the CIL-API is used to obtain the correct types). Primitive types have no successors, while pointers can have zero or more. The input vector also provides a mapping between logical addresses (*i.e.* position within the input vector) and concrete addresses. An illustration of an input vector can be seen in the right column of Figure 1, while the algorithm used to initialize inputs is shown in Figure 2.

The remainder of this Section describes in detail how inputs are initialized. This is done in two stages. First primitive types are discussed, before moving on to pointer inputs. Note that arrays are treated as pointers.

Primitive Inputs:
Whenever an input needs to be instantiated, SYCON first checks to see if $inputMap$ already contains the corresponding identifier of the variable (provided as part of the instrumentation process). Generally, during the first iteration of a search, or after a random restart, the input vector will be empty. In this case the input parameter is assigned a randomly generated value (within the bounds of the input type). A node object representing the parameter is added to $inputMap$. Further, a reference to the input parameter is added to a hill climb vector via the function call $updateHCVector$. The use of this vector is described in more detail in Section 5.

If $inputMap$ already contains a node object for the current parameter, the tool either retrieves the value stored from the previous iteration, or it returns the result of a hill climb move for this variable (via the $getValue$ function call).

Pointer Inputs:
Pointer inputs are always initialized to $NULL$. The decision to assign a memory location (either new or existing) to a pointer is governed by the outcome of symbolic execution and the constraint handling algorithm shown in Figure 4, and described in more detail in Sections 3.3 and 3.4. Memory is only ever allocated if the $pointToNull$ flag for a node object is set to $false$ by the constraint handling algorithm. The remainder of this sub–section discusses the scenario where this flag has been set to $false$.

The node object for pointer inputs contains a field to store the id of the node pointed at ($targetNodeId$). Whenever this field is updated during the course of a search (by the algorithm in Figure 4), the target node id's of each pointee are recursively traversed.

If a constraint requires a pointer to be non-$NULL$, SYCON first examines the pointer type. If $typeOf(v)$ is a pointer to a primitive type, then it simply allocates sufficient physical memory (via `malloc`) to hold an object of the appropriate type. The content of the newly allocated memory location will be initialized by the algorithm for handling primitive inputs as previously described. However, instead of being added as an element to $inputMap$, the function $updateInputMap$ is used to insert the node object into the list of successors for the parent pointer instead (see right column, iteration 2 in Figure 1 for an example layout of $inputMap$).

If $typeOf(v)$ is a pointer to another pointer, data structure, union, or enum type, SYCON has two choices. Either allocate a new block of memory large enough to hold an object of $typeOf(*v)$, or assign an existing memory location to the input parameter. The decision depends on the object's $targetNodeId$, which can either be 0 or an identifier of another node. If it is 0, a new physical block of memory is allocated. For enumerated or composite types, members are recursively initialized, either following the procedure for primitive or pointer inputs. Note that any new pointers introduced in the process of allocating memory are again initialized to $NULL$. In case the $targetNodeId$ points to an existing object, either in $inputMap$ or reachable from an element in $inputMap$, the value stored in the target object (in effect another memory location) is assigned instead.

3.3 Symbolic Execution

This section describes the symbolic execution performed by SYCON alongside concrete execution of an input vector. Symbols are represented via the physical address of each variable involved in the computation of a program. The code is instrumented such that at the start of each function the addresses of all local variables are added to a symbol map S. In order to allow interprocedural tracking of constraints, SYCON uses a symbolic stack. Whenever a function call is made to a function whose source is available during the instrumentation process, the symbols of the variables passed as parameters are added onto a call stack. At the start of each function, SYCON then pops each formal parameter from the stack and adds the corresponding symbol to S. The type of the variables represented by the symbols is also recorded, so SYCON can distinguish between pointer or primitive constraints later on.

Concretely, SYCON uses the call stack to add "dummy" assignments for each formal parameter of a function. Symbols are pushed onto the call stack in the order they appear in the list of formals of the function called. Then, after S has been updated with the list of formals for that function, SYCON pops as many items from the stack as appear in the function's list of formals. For each formal parameter, the "dummy" assignment assigns the last item popped to the corresponding symbol for that parameter (see Figure 3 for an example). Note that variable–argument list functions are currently not handled.

SYCON distinguishes between four types of symbols: *primitive*, *pointer*, *constant* and *unknown*. The first three types are self–explanatory. The last type is used in case a symbolic variable is defined via an external function or system call, or in the process of bitwise operations currently not handled by SYCON. Whenever SYCON encounters an unknown symbol during its backward analysis it stops and tries to compute a branch distance instead.

In addition to a stack for interprocedural tracking of symbolic constraints, SYCON maintains a stack describing the symbolic expressions along the path taken by an input, *i.e.* the path condition for that input. Note that SYCON delays evaluation of the symbolic expressions in the stack until requested to do so by the fitness function, described in Section 3.4. Hence, SYCON only needs to evaluate expressions describing a sub–path as opposed to a complete path.

3.4 Fitness Function

During the fitness evaluation of a solution, the fitness function iterates over the execution trace produced by the solution. At each branching node, the function first checks if the node is critical with respect to the current target. If the node appears in the critical path, the function queries the symbolic stack whether to obtain a branch distance, or whether to pass the node to the constraint solving algorithm.

Due to the simplifications performed before and during the instrumentation, all variables used in predicates are of primitive type, either representing a memory location, or a primitive. Therefore, it is insufficient to examine the type of these variables only. In order to determine if a predicate involves a pointer comparison (either $=$ or $\neq$), SYCON has to perform a backward analysis. During this analysis, SYCON performs a substitution of expressions. Because we are not interested in constraints involving primitives or pointers to primitive types, SYCON does not need to perform a full

```
void foo(int x){ /*function Id = 1*/
   /*implicit assignment X = X1 added by SYCON
    *X, X1 represent the symbolic values for x, x1*/
   pop_symbol(&x,typeOf(x),1);
}
void main(){
   int x1;
   addLocalSymbol(&x1,typeOf(x1));
   push_symbol(&x1);
   foo(x);
}
```

Figure 3: Illustration of symbolic stack

evaluation of all symbolic expressions, *e.g.* handle expressions involving $\times, \div$ *etc.* The only expressions of interest are simple assignment statements without any arithmetic operator, except $+$ or $-$. Whenever SYCON encounters addition of two symbolic expressions, it checks to see if the result is another memory location. It is able to do so because the concrete value of expressions appearing on the right hand side of an assignment is recorded during execution. This makes the analysis very lightweight.

The substitution continues until either the corresponding expression involving input symbols only is found, or no more substitutions can be made. In the latter case the algorithm indicates to use a branch distance. If both symbols appearing in the predicate can be expressed in terms of input symbols, the predicate is passed to the constraint handling algorithm shown in Figure 4. This algorithm returns either 0 or 1 depending on if the constraint was satisfied or not respectively. These values correspond to the *ideal* and *worst* branch distance values. The branch distance is combined with the approximation level to form the overall fitness of a solution [21].

A special case are conditionals checking for $NULL$, or non-$NULL$. If these predicates can be expressed in terms of input symbols, one of the symbols for the expression will be $NULL$, to indicate the special variable $NULL$. Note that whenever a pointer input is assigned a value by the constraint handling algorithm, a flag ($fixed$) is set in the corresponding node object for the input.

3.5 The Algorithm

This section describes the top level algorithm for the extended hill climb (shown in Figure 5). Recall that the search starts with a (semi) randomly generated input vector. The termination criteria for a search is a solution whose fitness is 0, or, the budget of fitness evaluations has been exhausted. Fitness values are assigned to solutions at the end of the *evaluateSolution* function. This function performs the fitness calculations described in Section 3.4. If the constraint algorithm from Figure 4 is used to obtain a fitness value, the global flag *solveConstraint* is set. This indicates that the algorithm needs to check if the constraint was successfully satisfied in the next iteration. In case it was not, the search will repeatedly attempt to solve the constraint until it is either satisfied, or the number of fitness evaluations has exceeded the allowed maximum.

Recall that all primitive inputs and primitive types reachable from a pointer to a primitive type are also added to a separate vector. The exploratory moves made by the function *exploreNeighbourhood* (described in Section 2.1) are

```
/*op is a relational operator, with op ∈ {=, ≠}
 *mem1 is the address stored in the pointer to the left of op
 *mem2 is the address stored in the pointer to the right of op
 *c is a predicate of the form mem1 op mem2
*/
if c is satisfied then
  return 0;
end if
if mem ⋈ NULL then //mem ∈ {mem1,mem2}, ⋈∈ {=, ≠}
  set mem ⋈ NULL;
else
  if op = "=" then
    if mem1 is free then
      set mem1 to mem2;
    else if mem2 is free then
      set mem2 to mem1;
    else
      set mem1 to mem2;
      set mem2 to mem1;
    end if
  else if op = "!" then
    if mem1 is free then
      set mem1 to NULL or new memory location; //via malloc
    else if mem2 is free then
      set mem2 to NULL or new memory location; //via malloc
    else
      toss = random()
      if(toss = 0)
        set mem1 to NULL or new memory location; //via malloc
      else
        set mem2 to NULL or new memory location; //via malloc
      end if
    end if
  end if
return 1;
```

Figure 4: Pseudo code describing the set of rules for handling pointer constraints. $mem1$ and $mem2$ represent the memory addresses stored in the pointer variables appearing in a predicate. Whenever an input pointer is assigned via the constraint algorithm, a flag is set to indicate that the input is no longer 'free', and the algorithm will try and set free pointer variables first, before reassigning values to previously handled input pointers. The value returned by the rules is equal to the ideal and worst branch distance for a predicate, 0 and 1 respectively.

performed on this vector. During these moves, the search moves to the first neighbour that achieves a better fitness value. Once such a move is made, the algorithm tries to speed up the search by performing a series of consecutive moves with ever increasing step size, until the search either reaches the target, or overshoots it. In the latter case, the search needs to backtrack, leading to a homing–in effect on the target.

4. EXPERIMENTAL EVALUATION

We conducted a set of case studies to gain insight into how effective SYCON is compared to CUTE in achieving branch coverage. The experiments were conducted on a total of 116 branches drawn from code used in industry (`f2`), open source (`space,cpplib`) as well as a synthetically constructed program. Details of the test objects and the results can be seen in Table 1. The industrial code example was provided by DaimlerChrysler, while the other examples were specifically chosen to accommodate the constraint solver used by CUTE, which cannot handle non–linear constraints and only has a very limited ability to deal with floats and doubles.

```
improvedMove := true;
*solveConstraint := false;
*solution := random();
while !terminate() do
  improvedMove := evaluateSolution(solution,solveConstraint);
  while improvedMove && !terminate() do
    *solution := exploreNeighbourhood();
    improvedMove := evaluateSolution(solution,solveConstraint);
    if solveConstraint then
      while improvedMove && !terminate() do
        *solution = handleConstraint();
        improvedMove := evaluateSolution(solution,solveConstraint);
      end while
    else if improvedMove then
      makeMove();
      consequtiveImprovement := true;
      while consequtiveImprovement && !terminate() do
        consequtiveImprovement := tryConsecutiveMoves(solution);
      end while
    end if
  end while
  solution := random();
end while
```

Figure 5: Extended hill climb algorithm

In order to facilitate comparison between SYCON and CUTE we defined the following framework. SYCON was allowed a maximum of 1000 fitness evaluations per branch. The maximum number of iterations for CUTE was set to $1000 \times branches$ per program. CUTE can be run in two modes. During one, all primitive inputs are non–randomly initialized to 0, while in the second, primitive types are randomly initialized. We adapted our tool to also run in these two modes. Pointers are always initialized to $NULL$ in both SYCON and CUTE. For CUTE we set the bounded depth–first search parameter to infinite.

All experiments were carried out on a laptop running Linux, with an AMD Turion64 dual core processor. GCC version 4.1.220070925 (Red Hat 4.1.2 27) was used by CUTE and SYCON to compile the code under test. Due to the stochastic nature of our algorithm, we executed each test object 30 times, averaging the coverage achieved in each run and the respective number of iterations taken.

The `f2` program provided by DaimlerChrysler is part of an engine control system used in the cars of their S-class series. The code is machine generated from design models. As a result the code is not in human–readable form, placing extra burden on a tester, and making automated test data generation all the more desirable.

The examples from the `cpplib` library and from the European Space Agency program `space` were chosen because they take pointers to data structures as input. In the case of `space` these are often very complex, with members cross referencing other data structures. This makes the task of generating test data again more.

Finally the synthetic example illustrates how search based approaches are inherently interprocedural and thus not inhibited by internal function calls, providing the fitness function is able to compute a smooth fitness landscape for predicates dependent on the outcome of such a function call. The example contains a predicate which compares a value returned from a function call with an input parameter. The

Test Object	CUTE primitives init to 0		SYCON primitives init to 0	
	avg. coverage	avg. iterations	avg. coverage	avg. fitness evaluations
`f2`	69.56%	32	72.72%	12124
`cpplib:`				
`reverse_token_list`	100%	8000	100%	114
`compare_token_lists`	75%	2000	87.5%	1120
`space:`				
`addscan`	62.10%	58000	85.71%	8430
`synthetic`	100%	5	100%	117
	primitives init to random		primitives init to random	
`f2`	69.92%	32.8	68.18%	15179.6
`cpplib:`				
`reverse_token_list`	100%	8000	100%	114
`compare_token_lists`	33.33%	1.83	75%	115
`addscan`	N/A (infinite loop)		53.57%	27014.8
`synthetic`	100%	5	100%	308

Table 1: Results from an experimental study comparing SYCON and CUTE.

example also contains a predicate that checks for a self referencing pointer. SYCON is able to find instantiations of the input parameters satisfying both these conditions; the first by calculating a branch distance based on the return value of the function, and the second by using the constraint algorithm from Figure 4.

On average, the results suggest that SYCON is better at achieving branch coverage for the example programs from industry and open source. Interestingly the search based algorithm performed better when primitive inputs were initialized to 0 instead of being assigned random values. We believe this can be explained by the fact that the solution space for the open source and industry programs is very small, while the search space is very large. Especially in `f2` a lot of branches depend on small constant values. Hence the search is inherently closer to the optimal solution when primitives are initialized to 0.

SYCON was able to raise two internal errors when testing the `addscan` function from the `space` program. These were signalled by `interror(‘‘addscan() -1- ’’)` and `interror(‘‘sgrrot - 1 ’’)`. CUTE was unable to cover any of the branches raising these errors. When applied to `space`, the level of coverage achieved by SYCON dropped dramatically without the restrictions on initializing primitive inputs. Most of the branches in the `addscan` function are hard to cover. Having input parameters similar to each other (*e.g.* when initializing primitives to 0) can potentially make the search much more efficient and effective because it saves unnecessary moves. Without these restrictions, the search is likely to waste a significant proportion of its fitness allowance before making real progress.

CUTE also struggled with the `addscan` function in "random mode" and got stuck in an infinite loop at every attempt, forcing the search to be abandoned. Note, for programs where CUTE can in theory traverse infinitely many paths during its symbolic execution, the search will run until the maximum number of allowed iterations has been reached, even if CUTE is unable to cover any more branches. This is often the case when testing tree–like structures with an unbounded depth–first search, and explains the large number of iterations for `addscan`, `reverse_token_list` and `compare_token_lists`.

5. RELATED WORK

One of the first authors to address the problem of branch coverage using a search based technique was Korel [13]. Korel's approach used a local search procedure; the alternating hill climb method also adopted in this paper. Xanthakis *et al.* [22] were the first to apply a global search algorithm to the problem of test data generation. They used evolutionary algorithms, in particular genetic algorithms. Since then, many authors have applied evolutionary algorithms to unit testing and branch coverage, for many different testing criteria [9, 16, 18, 21].

Independent of the work on dynamic test data generation, symbolic execution and constraint solving have also been developed as techniques for automated software test data generation [12, 3] since the 1970's.

More recently these two strands of research have been combined in what is known as concolic testing. Concolic testing overcomes many problems associated with both symbolic execution and random search. Cadar and Engler [2] adopt a similar approach to that of CUTE. Their approach starts at the opposite end to CUTE. Initially a program is executed symbolically. When the execution encounters constraints it cannot handle (*e.g.* at a point $P(c)$), the path constraint is solved to generate a concrete input vector leading up to $P(c)$. From $P(c)$ onwards, they proceed with both, symbolic as well as concrete values. Once a path has fully been traversed, test cases are generated by concrete instantiations of the symbolic constraints.

The majority of literature on concolic testing has augmented static analysis techniques, primarily symbolic execution, with dynamic test data generation techniques, specifically random testing. Inkumsah and Xie [8] are the first authors to propose a framework (EVACON) combining evolutionary testing with concolic testing. Their framework targets test data generation for object oriented code written in JAVA. They use two existing tools, eToc [20], an evolutionary test data generation tool for JAVA, and jCUTE,

an explicit path model checker [19]. The first is used to construct method sequences to test a JAVA method, while the second is used to optimize the coverage for the method under test. eToc uses a genetic algorithm to evolve method sequences along with their parameters as a means to achieve unit testing. The parameters used in the sequence of method calls are randomly initialized. One drawback of eToc is that new parameter values are only introduced into the population via the mutation operator, which randomly changes a parameter value within given bounds. This may make the search inefficient and even ineffective for certain branches because genetic operators need to share their effort between evolving sequences and evolving their parameters. However, eToc provides an interface allowing customized input parameter generation. The EVACON framework uses this feature to combine jCUTE with eToc. Unlike eToc, jCUTE provides a systematic approach to achieving branch coverage based on a combination of symbolic execution and random testing.

First, method call sequences (test cases) are constructed by eToc. These test cases are then passed to jCUTE, which will try and generate parameter values to cover all feasible branches for a particular sequence call construct. The sequences with their modified parameter values are then returned to eToc, to further evolve the order and number of method calls in a sequence.

The approach introduced in the present paper is only targeted at procedural programming languages and aims to extend an existing search based testing technique rather than provide a framework integrating existing tools. The EVACON framework essentially relies on jCUTE's ability to achieve branch coverage. It is our belief that search based testing is applicable to a wider range of programs, and therefore the motivation was to extend its applicability by filling the gaps with respect to dynamic data structures.

6. CONCLUSION AND FUTURE WORK

This paper introduced a new approach to handling dynamic data structures in search based test data generation, inspired by work on directed random testing (also known as concolic testing). It combines a lazy initialization approach for pointer variables with an alternating variable hill climb method, similar to that introduced by Korel. The paper thus improves the effectiveness of a commonly used local search method in search based testing and extends its applicability to include dynamic data structures. The paper reports the results of experiments on real world as well as a synthetic program, comparing a fully automatic prototype implementation of the proposed algorithm with CUTE, a concolic unit test engine.

While the results of the experimental case study look promising, there remains scope for future work. This will include examining the use of a constraint solver for linear constraints in order to improve the efficiency of the search by saving hill climb moves. The intention is to use a hill climb algorithm only in the presence of floating point calculations, non–linear expression, or generally in cases which are too complex to be handled by a lightweight constraint solver.

7. ACKNOWLEDGMENTS

Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London. Kiran Lakhotia is funded by EU grant IST-33472.

8. REFERENCES

[1] lp_solve. web page: http://tech.groups.yahoo.com/group/lp_solve/.

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

[3] Richard A. Demillo and A. Jefferson Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

[4] Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 47–54. ACM, 2007.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[6] Mark Harman. The current state and future of search based software engineering. In *IEEE Computer Society Press*, Future of Software Engineering 2007, Los Alamitos, California, USA, 2007, 2007.

[7] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In David S. Rosenblum and Sebastian G. Elbaum, editors, *ISSTA*, pages 73–83. ACM, 2007.

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

[9] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.

[10] D. Kebbal. Automatic flow analysis using symbolic execution and path enumeration. In *ICPP Workshops*, pages 397–404. IEEE Computer Society, 2006.

[11] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

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

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

[14] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426. IEEE Computer Society, 2007.

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

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

[17] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–??, 2002.

[18] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[19] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[20] Paolo Tonella. Evolutionary testing of classes. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 119–128. ACM, 2004.

[21] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[22] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In 5^{th} *International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.