Hybrid Concolic Testing*

Rupak Majumdar
CS Department, UC Los Angeles, USA
rupak@cs.ucla.edu

Koushik Sen
EECS Department, UC Berkeley, USA
ksen@cs.berkeley.edu

Abstract

We present hybrid concolic testing*, an algorithm that interleaves random testing with concolic execution to obtain both a deep and a wide exploration of program state space. Our algorithm generates test inputs automatically by interleaving random testing until saturation with bounded exhaustive symbolic exploration of program points. It thus combines the ability of random search to reach* deep *program states quickly together with the ability of concolic testing to explore states in a neighborhood exhaustively. We have implemented our algorithm on top of CUTE and applied it to obtain better branch coverage for an editor implementation (VIM 5.7, 150K lines of code) as well as a data structure implementation in C. Our experiments suggest that hybrid concolic testing can handle large programs and provide, for the same testing budget, almost 4× the branch coverage than random testing and almost 2× that of concolic testing.*

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and debugging.
General Terms: Verification, Reliability.
Keywords: directed random testing, concolic testing.

1 Introduction

Testing is the primary way to find bugs in software. Testing using manually generated test cases is the primary technique used in industry to improve reliability of software—in fact, manual testing accounts for 50–80% of the typical cost of software development. However, manual test input generation is expensive, error-prone, and usually not exhaustive.

With the increasing power of computers and advances in theorem proving and constraint solving technologies, there has been a renewed interest in automated testing. A simple and often effective technique for automated testing is *random testing* [3, 22, 12, 5, 7, 23]. Random testing generates a large number of inputs randomly. The program is then run on those inputs to check if programmer written assertions hold, or in the absence of specifications, if a wide range of program behaviors including corner cases are exercised. Random testing scales well in the sense that the time taken to run the program on an input does not incur additional overhead beyond program execution. However, random testing does not guarantee correctness, and more disturbingly, the range of behaviors covered for large programs is often vanishingly small in comparison to all the possible behaviors of the program. As a consequence, many bugs remain after random testing. Thus, while random testing can reach *deep* states of the program state space by executing a large number of very long program paths quickly, it fails to be *wide*, that is, to capture a large variety of program behaviors.

The inadequacy of random test input generation has led to several *symbolic* techniques that execute a program using symbolic values in place of concrete inputs [19, 6, 30, 28, 2, 32, 33]. Precisely, the program is supplied symbolic constants for inputs, and every assignment along an execution path updates the program state with symbolic expressions and every conditional along the path generates a constraint in terms of the symbolic inputs. The goal is then to generate concrete inputs that satisfy the constraints generated along a symbolic execution path: these inputs are guaranteed to execute along this path. Moreover, different symbolic executions generate different program behaviors, leading to better coverage.

Recently, *concolic testing* [14, 25, 4] has been proposed as a variant of symbolic execution where symbolic execution is run simultaneously with concrete executions, that is, the program is simultaneously executed on concrete and symbolic values, and symbolic constraints generated along the path are simplified using the corresponding concrete values. The symbolic constraints are then used to incrementally generate test inputs for better coverage by conjoining symbolic constraints for a prefix of the path with the negation of a conditional taken by the execution. The primary advantage of concolic execution over pure symbolic simu-

*This research was sponsored in part by the grants NSF-CCF-0427202 and NSF-CCF-0546170.

lation is the presence of concrete (data and address) values, which can be used both to reason precisely about complex data structures as well as to simplify constraints when they go beyond the capability of the underlying constraint solver.

In practice, however, both for symbolic and concolic execution, the possible number of paths that must be considered symbolically is so large that the methods end up exploring only small parts of the program state space, and those that can be reached by "short" runs from the initial state, in reasonable time. Furthermore, maintaining and solving symbolic constraints along execution paths becomes expensive as the length of the executions grow. Thus previous applications of these techniques have been limited to small units of code [25] and path lengths of at most about fifty thousand basic blocks [18]. That is, although *wide*, in that different program paths are explored exhaustively, symbolic and concolic techniques are inadequate in exploring the *deep* states reached only after long program executions.

This is unfortunate, since concolic techniques hold most promise for larger and complicated pieces of code for which generating test suites with good coverage of corner case behavior is most crucial. A natural question then is how to combine the strengths of random testing and concolic simulation to achieve both *a deep and a wide* exploration of the program state space.

We present *hybrid concolic testing*, a simple algorithm that interleaves the application of random tests with concolic testing to achieve deep and wide exploration of the program state space. From the initial program state, hybrid concolic testing starts by performing random testing to improve coverage. When random testing *saturates*, that is, does not produce any new coverage points after running some predetermined number of steps, the algorithm automatically switches to concolic execution *from the current program state* to perform an exhaustive bounded depth search for an uncovered coverage point. As soon as one is found, the algorithm reverts back to concrete mode. The interleaving of random testing and concolic execution thus uses both the capacity of random testing to inexpensively generate deep program states through long program executions and the capability of concolic testing to exhaustively and symbolically search for new paths with a limited lookahead.

The interleaving of random and symbolic techniques is the crucial insight that distinguishes hybrid concolic testing from a naïve approach that simply runs random and concolic tests in parallel on a program. This is because many programs show behaviors where the program must reach a particular state s and then follow a precise sequence of input events σ in order to get to a required coverage point. It is often easy to reach s using random testing, but not then to generate the precise sequence of events σ. On the other hand, while it is usually easy for concolic testing to generate σ, concolic testing gets stuck in exploring a huge number of program paths before even reaching the state s. We give a few examples of this behavior. For example, in a web server, each connection maintains a state machine that moves the server between various states: disconnected, connected, reading, etc. Random testing can provide the inputs necessary to reach particular states of the machine, for example, when the server is processing a request, by generating inputs that exercise the "common case." However, from a particular state, the server can consider a specific sequence of events to account for application specific rules (for example, the server must disconnect if a user name that is not registered requests a special command) which are not found by randomly setting the inputs. Similarly, in a text editor, random inputs can get the system into a state where there is enough data in the editor's buffers so that certain commands (for example, delete lines or format paragraphs) are enabled.

As the examples indicate, hybrid concolic testing is most suitable for testing *reactive* programs that periodically get input from their environment. Examples of such programs include editors, network servers, simple GUI based programs, event based systems, embedded systems, and sensor networks. On the other hand, *transformational* programs, that get some fixed input initially, are not suitable for hybrid concolic testing, since the future behavior cannot be affected by symbolic execution after the initial input has been set.

In the end, hybrid concolic testing has the same limitations of symbolic execution based test generation: the discovery of uncovered points depends on the scalability and expressiveness of the constraint solver, and the exhaustive search for uncovered points is limited by the number of paths to be explored. Therefore, in general, hybrid concolic testing may not achieve 100% coverage, although it can improve random testing considerably. Further, the algorithm is not a panacea for all software quality issues. While we provide an automatic mechanism for test input generation, all the other effort required in testing, for example, test oracle generation, assertion based verification, and mock environment creation still have to be performed as with any other test input generation algorithm. Further, we look for code *coverage*, which may or may not be an indicator of code *reliability*.

We have implemented hybrid concolic testing on top of the CUTE tool for concolic testing [25] and applied it to achieve high branch coverage for C programs. In our preliminary experiments, we compare random, concolic, and hybrid concolic testing on the VIM text editor (150K lines of C code) and on an implementation of the red-black tree data structure. Our experiments indicate that for a fixed testing budget, hybrid concolic testing technique outperforms both random and concolic in terms of branch coverage, of-

```
void testme() {
  char * s;
  char c;
  int state = 0;

  while (1) {
      c = input();
      s = input();

     /* a simple state machine */
      if (c == '[' && state == 0) state = 1;
      if (c == '(' && state == 1) state = 2;
      if (c == '{' && state == 2) state = 3;
      if (c == '~' && state == 3) state = 4;
      if (c == 'a' && state == 4) state = 5;
      if (c == 'x' && state == 5) state = 6;
      if (c == '}' && state == 6) state = 7;
      if (c == ')' && state == 7) state = 8;
      if (c == ']' && state == 8) state = 9;

      if (s[0] == 'r' && s[1] == 'e'
           && s[2] == 's' && s[3] == 'e'
           && s[4] == 't' && s[5] == 0
           && state == 9) {
         ERROR;
      } } }
```

Figure 1. A simple function

ten getting almost $2\times$ the coverage achieved by either random or concolic testing alone. These results, together with the relative ease with which hybrid concolic testing can be implemented on top of existing random and concolic testers, demonstrate that hybrid concolic testing is a robust and scalable technique for automatic test case generation for large programs.

2 Motivating Example

We illustrate the benefits of hybrid concolic testing using the simple function `testme` shown in Figure 1. The function, which runs in an infinite loop, receives two inputs in each iteration. One input is a 8-bit character and the other input is a string. The function gets into an error state if the variable `state` is 9 and the input `s` is the string `''reset''`. Such functions are often generated by lexers. In the vim editor, we also found more complex forms of similar functions.

To test the function `testme`, if we generate random values for `c` and `s`, then after a few thousands of iterations the variable `state` will become 9 with high probability. However, the probability that `s` will be `''reset''` is extremely low. As such the probability that the `ERROR` statement will be hit after a large number of iterations is negligibly small. Therefore, for all practical purposes, random testing would not be able to reveal the `ERROR` in the `testme` function. This is true even if we bias random testing by restricting the values that a character can take to the set {`'['`, `']'`, `'{'`, `'}'`, `'('`, `')'`, `' '`, `'a'`, `'x'`, `'r'`, `'e'`, `'s'`, `'e'`, `'t'`, `0`}. This is because the probability of randomly generating the string `''reset''` is $1/15^6 \approx 10^{-7}$.

A better alternative that can reveal the error in `testme` is concolic testing, which will systematically explore all possible execution paths of the function `testme` by generating test inputs from symbolic constraints that force execution along particular program paths. Since the function `testme` runs in an infinite loop, the number of distinct feasible execution paths is infinite. Therefore, to perform concolic testing we need to bound the number of iterations of `testme` if we perform depth-first search of the execution paths, or we need to perform breadth-first search. The number of possible choices of values of `c` and `s` that concolic testing would consider in each iteration is 17. Moreover, at least 9 iterations are required to hit the `ERROR`. Therefore, concolic testing will explore approximately $17^9 \approx 10^{11}$ paths before it can hit the `ERROR`. Therefore, concolic testing is unlikely to reveal the `ERROR` in `testme` in a reasonable amount of time.

In hybrid concolic testing, we exploit the fact that random testing can take us in a computationally inexpensive way to a state in which `state=9` and then concolic testing can enable us to generate the string `''reset''` through exhaustive search. The random testing phase takes a couple of minutes to reach `state=9`. After that there will be no increase in the coverage and hybrid testing will start the concolic testing phase. In the concolic testing phase, concolic testing will generate the string `''reset''` in a single iteration after exploring 7 feasible execution paths. As a result hybrid concolic testing will usually hit `ERROR` in a couple of minutes.

We validated this fact by testing the function `testme` using all the three methods–pure random testing, pure concolic testing, and hybrid concolic testing. We found that both pure random testing and pure concolic testing was not able to hit the `ERROR` after one day of testing. However, hybrid concolic testing was able to hit the bug within two minutes on a 2GHz Pentium M laptop with 1GB RAM.

Figure 2 provides an informal comparison between concolic testing and hybrid concolic testing. The boxes represent the entire program state space, with particular coverage points shown using bold squares. The initial program state is the filled circle. Figure 2(a) shows concolic testing. After an initial random run (shown using the thin jagged lines), constraint solving tries to exhaustively search part of the state space. In this way, concolic testing does eventually hit the coverage points in the vicinity of the random execution, but the expense of exhaustive searching means that many other coverage points in the program state space can remain uncovered while concolic testing is stuck searching one part

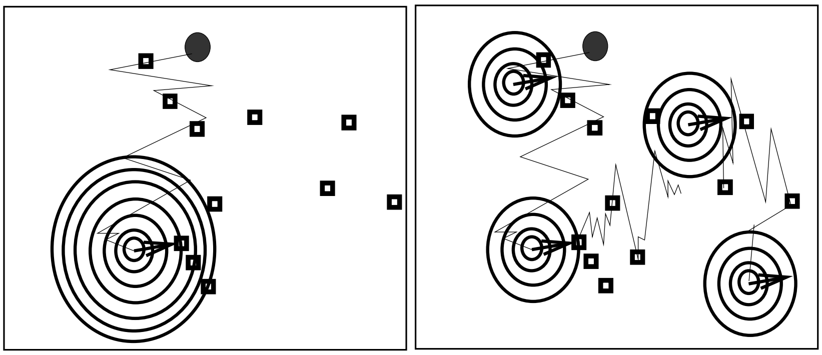

Figure 2. Comparison between (a) concolic and (b) hybrid concolic testing

of the state space exhaustively. In contrast, hybrid concolic testing (Figure 2(b)) switches to inexpensive random testing as soon as it identifies *some* uncovered point, relying on fast random testing to explore as much of the state space as possible. In this way, it avoids expensive constraint solving to perform exhaustive search in some part of the state space. Moreover, if random testing does not hit a new coverage point, it can take advantage of the locally exhaustive search provided by concolic testing to continue from a new coverage point.

3 Algorithm

We now present the algorithm for hybrid concolic testing preceded by a description of the programming model and a brief recapitulation of concolic testing.

3.1 Programs and Concrete Semantics

We illustrate the hybrid concolic testing algorithm on an imperative programming language. The operations of the programming language consist of labeled statements $\ell : s$. Labels correspond to instruction addresses. A statement is either (1) the halt statement halt denoting normal program termination, (2) an *input statement* $\ell : m := \mathsf{input}()$ that gets an external input into the lvalue m, (3) an assignment $m := e$ where m is an lvalue and e is a side-effect free expression, (4) a conditional statement $\mathsf{if}(e)\mathsf{goto}\ \ell$ where e is a side-effect free expression and ℓ is a program label, and (5) an abort statement signifying program error. Execution begins at the program label ℓ_0. For a labeled assignment statement $\ell : m := e$ or input statement $\ell : m := \mathsf{input}()$ we assume $\ell + 1$ is a valid label, and for a labeled conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ we assume both ℓ' and $\ell + 1$ are valid program labels.

The set of *data values* consists of program memory addresses and integer values. The semantics of the program is given using a *memory* consisting of a mapping from program addresses to values. Execution starts from the initial memory M_0 which maps all addresses to some default value in their domain. Given a memory M, we write $M[m \mapsto v]$ for the memory that maps the address m to the value v and maps all other addresses m' to $M(m')$.

Statements update the memory. The concrete semantics of the program is given in the usual way as a relation from program location and memory to an updated program location (corresponding to the next instruction to be executed) and updated memory [21]. For an assignment statement $\ell : m := e$, this relation calculates, possibly involving address arithmetic, the address m of the left-hand side, where the result is to be stored. The expression e is evaluated to a concrete value v in the context of the current memory M, the memory is updated to $M[m \mapsto v]$, and the new program location is $\ell + 1$. For an input statement $\ell : m := \mathsf{input}()$, the transition relation updates the memory M to the memory $M[m \mapsto v]$ where v is a nondeterministically chosen value from the range of data values, and the new location is $\ell + 1$. For a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, the expression e is evaluated in the current memory M, and if the evaluated value is zero, the new program location is ℓ' while if the value is non-zero, the new location is $\ell + 1$. In either case, the new memory is identical to the old one. Execution terminates normally if the current statement is halt, abnormally if the current statement is abort.

The nondeterminism introduced by input statements is resolved by using an *input map*. An input map IMap is a function that specifies values for inputs based on the execution history of the program. The *random input map* Random generates a value at random every time a concrete input is requested and returns this random value. We assume that the concrete semantics of the program is implemented as a function $\mathsf{Concrete}$ that takes a program location, a memory, and an input map, and returns a new program location and a new memory or terminates the program.

3.2 Concolic Testing

We now recapitulate the concolic testing algorithm from [14, 25]. Concolic testing performs symbolic execution of the program together with its concrete execution. It maintains a *symbolic memory map* μ and a *symbolic constraint* ξ in addition to the memory. These are filled in during the course of execution. The symbolic memory map is a mapping from concrete memory addresses to symbolic expressions, and the symbolic constraint is a first order formula over symbolic terms. The details of the construction of the symbolic memory and constraints is standard [28, 14, 25]. That is, at every statement $\ell : m := \mathsf{input}()$, the symbolic memory map μ introduces a mapping $m \mapsto \alpha_m$ from the address m to a fresh symbolic value α_m, and at every assignment $\ell : m := e$, the symbolic memory map updates the mapping of m to $\mu(e)$, the symbolic expression obtained by evaluating e in the current symbolic memory. The concrete values of the variables (available from the memory map M) are used to simplify $\mu(e)$ by substituting concrete values for symbolic ones whenever the symbolic expressions go beyond the theory that can be handled by the symbolic decision procedures.

The symbolic constraint ξ is initially `true`. At every conditional statement $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, if the execution takes the then branch, the symbolic constraint ξ is updated to $\xi \wedge (\mu(e) \neq 0)$ and if the execution takes the else branch, the symbolic constraint ξ is updated to $\xi \wedge (\mu(e) = 0)$. Thus, ξ denotes a logical formula over the symbolic input values that the concrete inputs are required to satisfy to execute the path executed so far.

Given a concolic program execution, concolic testing generates a new test in the following way. It selects a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ along the path that was executed such that (1) the current execution took the "then" (respectively, "else") branch of the conditional, and (2) the "else" (respectively, "then") branch of this conditional is uncovered. Let ξ_ℓ be the symbolic constraint just before executing this instruction and ξ_e be the constraint generated by the execution of this instruction. Using a decision procedure, concolic testing finds a satisfying assignment for the constraint $\xi_\ell \wedge \neg\xi_e$. The property of a satisfying assignment is that if these inputs are provided at each input statement, then the new execution will follow the old execution up to the location ℓ, but then take the conditional branch opposite to the one taken by the old execution, thus ensuring that the other branch gets covered. The satisfying assignment is used to define a new input map for the next run of the program. Suppose that there are k symbolic variables in the symbolic constraint, arranged in chronological order (that is, the symbolic input α_i was introduced for the ith input statement along the execution). Then, the next time the program is executed, the ith execution of an input statement for $i \leq k$ will return the value of variable α_i from the satisfying assignment, and for $i > k$ will return a random value.

We assume that concolic testing is implemented as a function $\mathsf{Concolic}$ that takes as input a program location and an initial memory map and returns a new input map. Such a function is easily obtained by wrapping existing implementations [14, 25].

3.3 Hybrid Concolic Testing: Schema

In *hybrid concolic testing*, random or biased random testing phases (that explore deep states of the program) are interleaved with concolic testing (that ensure complete coverage for a shallow neighborhood). Algorithm 1 shows a non-deterministic version of the hybrid concolic testing algorithm, where we have abstracted out certain implementation-dependent heuristics. The algorithms takes a program and a set of coverage goals (for example, branch coverage), and performs coverage-driven test input generation. The main loop of the algorithm (lines 1–15) runs while there are unsatisfied coverage goals (or, in practice, until resources run out or coverage goals are met). Each iteration of the loop starts with the initial location of the program, the initial memory map M_0 and the random input map (line 2) and runs the program until the program halts or hits abort. Each step of the execution is chosen according to some heuristic to be either a concrete execution (line 9), when the previous symbolic states are discarded and only the concrete semantics is followed, or a concolic execution starting with the current symbolic state (lines 11–13). The concolic execution first checkpoints the current concrete execution state (line 11), and starts running a concolic testing algorithm from the current state with the aim of hitting some unsatisfied coverage goals. When the concolic execution returns (either because it finds a new input to an uncovered coverage goal or because some resource budget is exhausted), the program state is restored but the input map is updated to be the new input that is guaranteed to hit a new coverage point (or, if resources were exhausted, generates random inputs). This has the effect of putting the execution back at the concrete state while setting (using the concolic execution) the future values of symbolic inputs to ensure that a new uncovered coverage goal is reached.

The test continues until the program terminates or a bug is found. At that point, if there are further uncovered coverage goals, the outer while loop restarts a new hybrid concolic execution.

3.4 Hybrid Concolic Testing: Algorithm

Algorithm 2 shows a deterministic version of Algorithm 1 where we instantiate the nondeterministic choices of Algorithm 1 with particular heuristics. Instead of choos-

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

Algorithm 1 Algorithm HCT (nondeterministic)

Input: program P, set of coverage goals Goals.

1: **while** Goals $\neq \emptyset$ **do**
2: $\quad \ell = \ell_0$, $M = M_0$, IMap $=$ Random
3: $\quad$ **while** nondet **do**
4: $\quad\quad$ **if** stmt_at$(\ell) =$ halt **then**
5: $\quad\quad\quad$ **break**
6: $\quad\quad$ **if** stmt_at$(\ell) =$ bug **then**
7: $\quad\quad\quad$ **return** bug
8: $\quad\quad$ **if** nondet **then**
9: $\quad\quad\quad (\ell, M) =$ Concrete$(\ell, M,$ IMap$)$
10: $\quad\quad\quad$ remove covered goals from Goals
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ snapshot(M)
13: $\quad\quad\quad$ IMap $=$ Concolic(ℓ, M)
14: $\quad\quad\quad M =$ restore$()$
15: $\quad$ **endwhile**
16: **endwhile**

Algorithm 2 Algorithm HCT

Input: program P, set of coverage goals Goals.

1: **while** Goals $\neq \emptyset$ **do**
2: $\quad \ell = \ell_0$, $M = M_0$, IMap $=$ Random
3: $\quad$ iter $= 0$
4: $\quad$ **while** iter $< \theta_2$ **or** stmt_at(ℓ) is not $x :=$ input$()$ **do**
5: $\quad\quad$ **if** stmt_at$(\ell) =$ halt **then**
6: $\quad\quad\quad$ **break**
7: $\quad\quad$ **if** stmt_at$(\ell) =$ bug **then**
8: $\quad\quad\quad$ **return** bug
9: $\quad\quad (\ell, M) =$ Concrete$(\ell, M,$ IMap$)$
10: $\quad\quad$ remove covered goals from Goals
11: $\quad\quad$ **if** coverage has increased **then**
12: $\quad\quad\quad$ iter $= 0$
13: $\quad\quad$ **else**
14: $\quad\quad\quad$ **if** stmt_at(ℓ) is $x :=$ input$()$ for some x **then**
15: $\quad\quad\quad\quad$ iter $=$ iter $+ 1$
16: $\quad$ **endwhile**
17: $\quad$ **if** iter $= \theta_2$ **then**
18: $\quad\quad$ snapshot(M)
19: $\quad\quad$ IMap $=$ Concolic(ℓ, M)
20: $\quad\quad M =$ restore$()$
21: $\quad\quad$ **goto** 3
22: **endwhile**

ing a random step or a concolic step at each iteration, the algorithm maintains a counter iter and runs the random steps until convergence, that is, until no new coverage goal has been discharged in the last θ_2 input instructions executed in the random testing. The condition in the **while** loop on line 4 ensures that we switch to concolic mode only at an input statement after θ_2 input statements have gone by without seeing a new coverage goal. At this point, the algorithm switches to the concolic mode, by first taking a snapshot of the current state and then running concolic execution from the current node, looking for a new uncovered goal. Once a new uncovered goal is found, the input map is updated and the program state is restored. The counter is reset and the loop starts executing the random mode again. Notice however that in this mode, the first inputs returned by the input map have been carefully selected by the concolic engine to hit an uncovered coverage point. Again, the algorithm continues running till a bug is found or at least some θ_1 fraction of coverage goals are met (or resource bounds are exhausted).

Snapshot and restore. The only remaining technical issue is the implementation of checkpointing and restoring states through the functions snapshot and restore. We use process creation through the system call `fork` to achieve checkpointing. Precisely, in our implementation, at the point we need to snapshot the current state, we fork off a child process. The child process starts with an exact copy of the parent's state and performs the concolic execution from the current location ℓ. At the end of the concolic execution, the child transmits the new logical input map back to the parent and dies. Meanwhile, the parent blocks waiting for the new logical input map. When it receives the new map, the parent continues executing the rest of the testing loop. The net effect is that the parent maintains the program state, gets an updated logical input map through the concolic testing, and can continue executing from the current state using this input map.

4 Experiments

We have implemented hybrid concolic testing on top of CUTE, a concolic unit testing engine for C [25]. In this section, we report the results of our experiments with two programs– an implementation of the red-black tree data structure, and the popular text editor VIM.

For each program, we describe the experimental setup and the results of comparing hybrid concolic testing to random testing[1] and (pure) concolic testing. We conducted the experiments on a 2GHz Pentium M laptop running Windows XP with 1 GB RAM. In our experiments, we report the relative branch coverage (which we will simply call coverage), i.e., the ratio of the total number of branches exercised during testing and the total number of branches present in the functions touched during testing. Relative coverage is important for the testing of red black tree because the implementation of red black tree is part of a large data structure library. Since we do not invoke the functions of other data structures during testing of the red black tree,

[1] We only consider uniform random testing where the input space is sampled uniformly at random.

```
typedef struct rbtree {
   int i;
   struct rbtree *left = NULL;
   struct rbtree *right = NULL;
   char color;
} rbtree;

void testme() {
   int toss;
   rbtree *elem, *tmp, *root = NULL;

   while(1) {
      CUTE_input(toss);
      if(toss<0) toss = -toss;
      toss = toss % 5;
      switch(toss) {
         case 1:
            rbtree_len(root);
            break;
         case 2:
            elem = (rbtree *)malloc(sizeof(rbtree));
            CUTE_input(elem->i);
            rbtree_add_if_not_member(&root,elem,&tmp);
            break;
         case 3:
            elem = (rbtree *)malloc(sizeof(rbtree));
            CUTE_input(elem->i);
            rbtree_delete_if_member(&root,elem,&tmp);
            break;
         case 4:
            elem = (rbtree *)malloc(sizeof(rbtree));
            CUTE_input(elem->i);
            rbtree_find_member(root,elem);
            break;
         default:
            elem = (rbtree *)malloc(sizeof(rbtree));
            CUTE_input(elem->i);
            rbtree_add(&root,elem);
            break;
      } } }
```

Figure 3. Driver for testing red-black tree

the absolute branch coverage will be very low and the number will not reflect the true branch coverage within the red black tree only. In the case of the VIM editor, we do not symbolically track all potential inputs, such as reading from a file. Therefore, we will not able to exercise many functions whose behaviors depend on such non-tracked inputs. By using relative branch coverage, we ignore the branches of such unreachable functions.

4.1 Red Black Tree

In our first experiment, we considered a widely-used implementation of the red-black tree data structure having around 500 lines of C code. We adopted the unit testing methodology to test this implementation. In particular, we adopted the approach of generating data structures using a sequence of function calls [28, 33]. This approach is based on the following observation: a data structure implements functions for several basic operations such as creating an empty structure, adding an element to the structure, removing an element from the structure, and checking if an element is in the structure. A sequence of these interface operations can be used to exhaustively test the implementation.

	Branch Coverage in Percentage		
Seed	Random Testing	Concolic Testing	Hybrid Concolic Testing
523	32.27	52.48	66.67
7487	32.27	52.48	67.02
6726	32.27	52.48	66.67
5439	32.27	52.48	67.73
4494	32.27	52.48	69.86
Average	32.27	52.48	67.59

Table 1. Results of Testing Red-Black Tree

Experimental Setup. To generate legal sequences of function calls of the red-black tree we used the manually written test driver shown in Figure 3. The test driver runs in a loop and calls a public function of the red-black tree in each iteration. The function to be called in each iteration is determined by an input variable `toss`. We biased the random testing so that each function call has an equal probability of being called in an iteration. We compared pure random testing, pure concolic testing, and hybrid concolic testing on the test driver using five different seeds. We allotted a time of 30 minutes for each testing experiment.

Results. Table 1 shows the results of testing the red-black tree implementation. The first column gives the initial seed for the random number generator used by each of the testing methods. The next three columns give the percentage of branch coverage for each of the testing methods. The last row gives the average branch coverage for each of the methods.

The table shows that the average branch coverage attained by pure random testing is low compared to both pure concolic testing and hybrid concolic testing. Moreover the branch coverage for random testing saturated at 32.27% for each of the five seeds. Random testing failed to attain high branch coverage because the probability of generating two random numbers having the same value is very small. As such random testing was not able to generate random numbers that are already in the tree. Therefore, the functions `rbtree_delete_if_member` and `rbtree_add_if_not_member` were not explored completely.

In concolic testing we bounded the number of inputs along each path by 10. This was required because the test driver has an infinite loop. Note that with the increase in the number of inputs along each path, the number of distinct feasible execution paths increases exponentially. Therefore, to be able to complete the exhaustive search of all the paths in a reasonable amount of time using concolic testing, we bounded the number of inputs along each path by 10. Then concolic testing gave us an average branch coverage of 52.48%. Although this number is better than that of random testing, we didn't manage to get better coverage. This is because to attain better coverage we need longer sequences of function call. This was also observed by D'Amorim et al. [9]. However, longer sequences cannot be completely tested by concolic testing due to the exponential blow-up in the number of paths.

To address this problem, hybrid concolic testing proved ideal. This is because the random testing mode of hybrid concolic testing generated long function call sequences. This resulted in the creation of large random red-black trees. After that the concolic testing mode was able to explore more execution paths. As a result hybrid concolic testing attained an average branch coverage of 67.59%, which was the highest of all the testing modes. Note that the branch coverage is still less than 100%. After investigating the reason for this, we found that the code contains a number of assert statements that were never violated and a number of predicates that are redundant and can be removed from the conditionals. Nevertheless, the experiment supports the claim that hybrid concolic testing, which combines the best of both worlds, can attain better branch coverage than pure random testing and pure concolic testing.

4.2 The VIM Editor

We next illustrate the use of hybrid concolic testing on VIM, a popular text editor [27]. The VIM editor has 150K lines of C code. We want to generate test inputs for VIM for maximal branch coverage. Unlike the unit testing approaches adopted by CUTE or DART, we targeted to test VIM as a whole system. This made the testing task challenging as the number of possible distinct execution paths that can be exhibited by VIM as a whole system is astronomically large.

VIM is a *modal* editor, that is, it has one mode for entering text and a separate mode for entering commands. It starts in the command mode, where the user can enter editor commands to move cursors, delete words or lines. When certain keys are pressed ("a" or "i"), the editor enters into *insert* mode, where the user can enter text. From the insert mode, the user goes back to command mode by pressing the ESC key. Further, in command mode, by pressing ":" the editor goes to a *command line* mode, where the next sequence of characters pressed by the user has special command significance to the editor (for example, ":" followed by "w" writes the current buffer back to disk). Similarly, pressing "/" in command mode takes the editor to a search mode, where the next sequence of characters typed by the user (up to a newline) is interpreted as a literal string to be searched for in the text buffer. There is also an `ex` mode for more complex command lines. There are many other modes VIM, and many other commands. For our purposes of exposition, we note that VIM has the characteristics of the example program in Figure 1: in order to hit certain branches, one has to take the program to a certain state, and then provide a precise sequence of inputs (which makes sense as a mode transfer followed by a command to the editor). For example, if we start VIM with an empty buffer, then the command `dd` (to delete a line) is not enabled. The command `dd` gets enabled after we have switched to the insert mode through the command `i`, entered some text into the buffer, and then switched to the command mode by pressing ESC. The random testing phase of hybrid concolic testing can enter garbage text into the buffer easily thus enabling the line deletion command. The concolic testing phase can then generate the sequence ESC `dd` during exhaustive search.

	Branch Coverage in Percentage		
Seed	Random Testing	Concolic Testing	Hybrid Concolic Testing
877443	8.01	21.43	41.93
67532	8.16	21.43	40.39
98732	8.72	21.43	33.67
32761	7.80	21.43	35.45
28683	9.75	21.43	40.53
Average	8.17	21.43	37.86

Table 2. Results of Testing the VIM Test Editor

Experimental Setup. To set up the testing experiment, we first identified the function in the VIM code that returns a 16-bit unsigned integer whenever the user presses a key. This function, namely `safe_vgetc`, provides inputs to VIM in the normal mode and the insert mode. In the VIM source code, we replaced `safe_vgetc` by the CUTE input function `CUTE_input()`. `CUTE_input()` provides random values to VIM in the random testing mode and provides values computed through constraint solving in the concolic testing mode. We observed that `safe_vgetc` does not provide input to VIM in the Ex mode and we failed to identify the exact low-level function that provides input to VIM in the Ex mode. As such in our testing experiments we were restricted to the exploration of behaviors of VIM in the insert mode and the normal mode only. This in turn affected the branch coverage that we obtained in the experiments.

We compared pure random testing, pure concolic testing, and hybrid concolic testing on the VIM source code using five different random seeds. In each experiment we restricted the total testing time to 60 minutes.

Results. Table 2 shows the results of testing the VIM text editor. As in the previous example, the first column gives the initial seed for the random number generator used by each of the testing methods. The next three columns give the percentage of branch coverage for each of the testing methods. The last row gives the average branch coverage for each of the methods.

For all the seeds for the random number generator, hybrid concolic testing gave better branch coverage than concolic testing and far better branch coverage than random testing. After analyzing the trace for one hybrid concolic testing experiment, we found that the random testing phases took VIM to deep states which cannot be otherwise be led by concolic testing. In the deep states, we found a lot of garbage text in the buffer. The concolic testing phases widely explored the state space near these deep states. This resulted in comparatively larger exploration of the state space of VIM.

Note that branch coverage obtained by hybrid concolic testing is still much lower than 100%. This is because we only tested the insert and the command modes of the VIM editor and did not touch the code for the other modes. The VIM experiment illustrates two caveats of our technique. First, the user has to identify the appropriate boundary at which to receive inputs for testing. This is not always easy to do without some knowledge of the implementation. We could only identify one input source (`safe_vgetc`). However, once a suitable input point is chosen, no further knowledge of the implementation is required. Second, even with some suitable input source, new coverage points may only be hit after a long sequence of correlated input. For example, certain configuration options may be read from a file. In our experience, concolic testing working at the individual character level does not scale very well in these cases. However, even with these caveats, hybrid testing performed around $4\times$ better than random testing because VIM requires a relatively short sequence of characters as input for a large number of commands; nevertheless, these sequences are long enough that they are not likely to be generated by random testing alone.

4.3 Discussion

In hybrid concolic testing, the concolic testing phase starts whenever random testing saturates, that is, does not find new coverage points even after running a predetermined number of steps. The presence of interleaved random testing phases in hybrid concolic testing thus guarantees that hybrid concolic testing is as good as random testing. Concolic testing generates meaningful sequences of inputs which cannot be otherwise generated by random testing. This boosted the branch coverage of the whole testing method considerably. In contrast, concolic testing was able to explore states only near the initial state rather than the deep states. As such in our experiments, pure concolic testing performed worse than hybrid concolic testing.

5 Related Work

In order to improve *test coverage*, several techniques have been proposed to automatically generate values for the inputs during testing. The simplest, and yet often very effective, techniques use random generation of (concrete) test inputs [3, 22, 12, 5, 7, 23, 20]. Although it has been quite successful in finding bugs, the problem with such random testing is twofold: first, many sets of values may lead to the same observable behavior and are thus *redundant*, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [22].

One approach which addresses the problem of redundant executions and increases test coverage is *symbolic execution* [19, 6]. Tools based on symbolic execution use a variety of approaches—including abstraction-based model checking [1, 2], parameterized unit testing [26], explicit-state model checking of the implementation [28, 29] or of a model [16, 31, 10], symbolic-sequence exploration [33, 24], and static analysis [8]—to automatically generate non-redundant test inputs. Several other approaches, such as the chaining method[11] and the iterative relaxation method [15], for test case generation do not use random execution or symbolic execution. Concolic testing [14, 25, 4] is a variation of symbolic execution where the symbolic execution is performed concurrently with random simulation. However, our experience with concolic testing using the CUTE and jCUTE tools have been that all these techniques ultimately run up against *path explosion*: programs have so many paths that must be symbolically explored that within a reasonable amount of time concolic testing can only explore only a small fraction of branches, those that can be reached using "short" executions from the initial state of the program. This was the initial motivation for our work: we wanted to augment concolic testing so that "deep" program states could be explored.

Our work on hybrid concolic testing is inspired by similar work in VLSI design validation [13, 17] where a combination of formal (symbolic execution or BDD based reachability) and random simulation engines are combined to improve design coverage for large scale industrial designs. Our contribution is to scale the orchestration of random and concolic testing to large software implementations. In comparison to model based testing using model checking [16, 10], we use random testing to seed the test, and use concolic testing rather than abstract model checking to explore the vicinity of a state. Using concolic testing circumvents aliasing issues that arise in abstraction based software model checking which makes abstract reachability imprecise for programs that manipulate heap data struc-

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

tures [2], and also alleviate the capacity problem for software model checkers. On the other hand, abstraction based model checking can prove branches definitely unreachable whereas our incomplete technique can only prove reachability.

References

[1] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *ICSE 04: International Conference on Software Engineering*, pages 326–335. IEEE, 2004.

[3] D. Bird and C. Muñoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[4] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN 05: Software Model Checking*, LNCS, pages 2–23. Springer, 2005.

[5] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *ICFP 00*, pages 268–279. ACM, 2000.

[6] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2:215–222, 1976.

[7] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE 05: International Conference on Software Engineering*, pages 422–431. IEEE, 2005.

[9] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 06: Automated Software Engineering*, pages 59–68. IEEE, 2006.

[10] G. Devaraj, M. Heimdahl, and D. Liang. Coverage-directed test generation with model checkers: Challenges and opportunities. In *COMPSAC (1)*, pages 455–462. IEEE, 2005.

[11] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[12] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[13] M. Ganai, A. Aziz, and A. Kuehlman. Enhancing simulation with BDDs and ATPG. In *DAC 99: Design Automation Conference*, pages 385–390. ACM, 1999.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI 05: Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[15] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *FSE 98: Foundations of Software Engineering*, pages 231–244. ACM, 1998.

[16] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *SEFM 04: Software Engineering and Formal Methods*, pages 261–270. IEEE Press, 2004.

[17] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulaton using collaborative formal and simulation engines. In *ICCAD 00: International Conference on Computer-Aided Design*, pages 120–126. ACM, 2000.

[18] R. Jhala and R. Majumdar. Path slicing. In *PLDI 05: Programming Language Design and Implementation*, pages 38–47. ACM, 2005.

[19] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[20] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE 05*, pages 267–276, 2005.

[21] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[22] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *ISSTA 96*, pages 195–200. ACM, 1996.

[23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 05: European Conference Object-Oriented Programming*, LNCS 3586, pages 504–527. Springer, 2005.

[24] Parasoft. Jtest manuals version 6.0. Online manual, February 2005. http://www.parasoft.com/.

[25] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE 05: Foundations of Software Engineering*. ACM, 2005.

[26] N. Tillmann and W. Schulte. Parameterized unit tests. In *FSE 05: Foundations of Software Engineering*, pages 253–262. ACM, 2005.

[27] Vim. http://www.vim.org/.

[28] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA 04: International Symposium on Software Testing and Analysis*, pages 97–107. ACM, 2004.

[29] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *ASE 05: Automated Software Engineering*, pages 414–417. IEEE, 2005.

[30] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *ASE 02: Automated Software Engineering*, pages 149–162. IEEE, 2002.

[31] S. Xia, B. D. Vito, and C. Muñoz. Automated test generation for engineering applications. In *ASE 05: Automated Software Engineering*, pages 283–286. IEEE, 2005.

[32] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE 04: Automated Software Engineering*, pages 196–205. IEEE, 2004.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3440, pages 365–381. Springer, 2005.

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE