Exhaustive Branch Coverage with TreeFrog

Nicky Williams
Université Paris-Saclay, CEA, LIST,
F-91120 Palaiseau
France
nicky.williams@cea.fr

ABSTRACT

Concolic methods efficiently generate test inputs for exhaustive path coverage. However, exhaustive path coverage is not often required or even realistic whereas exhaustive branch coverage is at the heart of many verification tasks. We explain how, in TreeFrog, we have tried to find an efficient solution to this very different problem. We have kept the efficient aspects of depth-first concolic generation but combined it with multi-threading for breadth-first search, conflict learning lifted to branches and finally, some controlled path enumeration. First results show dramatic improvements over concolic methods[1].

CCS CONCEPTS

•**Software and its engineering~Software creation and management~Software verification and validation~Software defect analysis~Software testing and debugging**

KEYWORDS

Automatic Test Generation, Concolic Test Generation, Dynamic Symbolic Execution, Path Explosion, Conflict Learning

ACM Reference format:

N. Williams. 2023. In *Proceedings of ACM SAC Conference, Tallinn, Estonia, March 27- March 31, 2023 (SAC'23),* 8 pages. DOI: 10.1145/3555776.3577738

1 INTRODUCTION

Structural code coverage criteria in testing are based on the simple observation that if the part of the program containing a bug is not executed (a.k.a. *covered*) by any test case then the bug cannot be detected. Various structural coverage criteria defining *test objectives* in source or binary code have been proposed but those most widely used in industry, typically for embedded safety-critical software, seem to be line coverage, branch coverage and modified condition/decision coverage (*MC/DC*), and these are often imposed by certification norms. Moreover these norms often impose 100% coverage of coverable test objectives and, by extension, some justification of why certain test objectives cannot be covered. We call this *exhaustive* coverage and define it as the generation, for all test objectives, of either

a. input values for a test case which covers the objective or
b. a *justification* of the lack of a test case.

We insist on the difference between exhaustive coverage and *bug-finding*, i.e. the speedy coverage of many test objectives with no guarantee of completion and where the stopping criterion is often a time limit on test generation rather than complete coverage. Indeed, a bug-finding test generation tool may succeed in covering all test objectives in many cases but exhaustive coverage requires tools that do this in all possible cases and justify any failure to reach an objective. Exhaustive coverage of large code bases is unrealistic today, because of the size of the search space and also because calls to library functions may have to be stubbed and the coverage will be dependent on the quality of the stubs. However, even relatively small programs can be very difficult to cover exhaustively during unit testing, particularly if they contain unreachable objectives.

Automatic test input generation techniques for exhaustive coverage must keep track of which objectives have already been covered in order to know when to stop test generation and to avoid generating numerous test cases which cover the same objectives while failing to cover others. Moreover, in order to provide a justification of failure to cover a particular test objective, test generation should not stop until all possible attempts to cover the objective have been made and should then report on the result of these attempts. Let us suppose that the uncovered objective is a branch in the source code. If there seem to be several partial paths through the source code leading to the branch, then all these paths must be taken into account. If the infeasibility of all paths to a branch, or the unsatisfiability of its weakest precondition can be automatically demonstrated, then this is the justification of failure to cover the branch. If infeasibility of one or more paths cannot be demonstrated automatically, for example because a prover fails or a constraint solver times out, then the justification for these paths is the formula submitted to the prover or solver.

Exhaustive branch coverage is also an interesting subject because of its proximity to other problems. For the automatic generation of test input values, line coverage and MC/DC are quite similar to branch coverage, even if MC/DC involves additional book-keeping. The coverage of other test objectives which can be defined in the source code (e.g. assertion violations), or even as pseudo-branches (e.g. run-time errors) is also similar. Techniques for branch coverage can be extended to these test criteria and indeed already are in many test generation tools. Moreover, the detection of certain types of "dead code" is ensured by exhaustive branch coverage. In other problems, often framed in terms of reachability, unwanted program states are defined and the problem is to demonstrate whether or not they can occur. Often, if they can occur, the user would like a counter-example, i.e. test input values, to help with debugging. In some cases, such as reactive systems, the reachability of even a single state requires sophisticated techniques (e.g. [5]) which we do not address in this paper. Nonetheless, in many cases there are actually numerous unwanted states and these multi-reachability problems are similar to exhaustive branch coverage.

Note that static analysis based on abstract interpretation can be an efficient technique for detecting unreachable branches but it cannot generate test input values and because of the over-approximation which is inherent in this approach, it cannot guarantee detection of all unreachable branches. It can be used as a prelude to automatic test generation in order to reduce the number of test objectives.

In this paper, we consider test generation based on *symbolic execution* and describe an extension of the *concolic* method. Concolic test generation offers the promise of exhaustive coverage because it can be used to systematically explore the tree of execution paths. It is very efficient for path coverage but less so when used for exhaustive branch coverage. This is because path coverage requires all feasible paths to be covered in any case but we do not usually need to cover all feasible paths in order to cover all reachable branches (because most branches occur in several feasible paths). If, as is often the case, the tree of execution paths is very large, then exhaustive branch coverage must limit how much of the tree is explored, i.e. must avoid what is known as *path explosion*.

This paper describes a method for test generation for exhaustive branch coverage which retains the advantages of the concolic method but increases efficiency by using conflict learning to prune the search space. The conflicts enable us to take account of the redundancy in the tree of execution paths and avoid repeating the same calculation. We start by recalling the classic concolic method based on symbolic execution and an existing strategy for branch coverage before exposing our new learning-based method, presenting the first results, comparing TreeFrog to the work in the literature and discussing possible improvements.

2 TEST GENERATION BASED ON SYMBOLIC EXECUTION

Test generation techniques based on symbolic execution explore the binary tree of execution paths (*EPtree*). For ease of explanation, we assume that the source code (in an imperative language such as C) has already been simplified in order to unroll loops, separate out side-effects, decompose complex conditions, inline functions, etc. In practice, system calls must also be stubbed and variables used when accessing array elements or dereferencing pointers give rise to additional alias constraints, but we can consider here, without loss of generality, that the EPtree represents simplified code composed of atomic branches and sequential blocks of assignments and that all variables are isolated (do not belong to a data structure). The EPtree is rooted in the entry point to the tested function and the leaves all represent exit from the tested function. A path from the root to a leaf represents an execution path of the source code. Each branch, b^t, in the tree represents an instance of a source code branch and each source code branch, b, has a different occurrence in the EPtree for each possible execution path prefix up to b. For exhaustive branch coverage, we must find at least one instance, b^t, of each source code branch, b, such that the execution path prefix of b^t is feasible and consistent with b^t. This means that when we append b^t to its prefix, the result is also feasible and a test can be generated to cover b (note that here and in the rest of the paper, we often shorten "execution path prefix" and "path prefix" to "prefix").

Symbolic execution traverses an execution path (or partial path), p, in the source code, starting from the function entry point and constructs a conjunction of constraints, $pred(p)$, on V, the symbolic values at input of the input variables. Each variable in the source code may be mutable, i.e. take successive values. The successive values are distinguished, by renaming mutable source code variables whenever they take a new value, as in *Single Static Assignment*. Each of the resulting single-assignment source code variables has only one value, which is represented by one symbolic constrained variable in the solver. We represent path p as a sequence of source code branches, $b_0,b_1,...$ Each branch, b_i, in p is translated by symbolic execution into a conjunction of constraints including the constraints, ca_i, due to the (possibly empty) block of assignments before b_i and an additional constraint, c_i, due to the branch condition of b_i. The resulting conjunction of constraints, $ca_0,c_0,ca_1,c_1,...$ is the path predicate, $pred(p)$, which defines the input values of all test cases which would cover p. To generate a new test case, the solver is called to find a solution to a new path predicate.

3 THE CONCOLIC METHOD

In the concolic method, the first test case is generated by calling the solver on an initial set of constraints over V representing the type declarations and a user-supplied precondition which encodes the test context. The solution will be an arbitrary set of legitimate concrete input values, i.e. the inputs of the initial test case. The tested code is instrumented and run on this test case to recover the path, p, covered by this initial case. For this and each subsequently generated test case, symbolic execution is used to translate the branches in p into $pred(p)$ as explained above. In order to generate new test inputs to cover a different path, one of the branch constraints, c_i, must be negated. This is often referred to as *flipping* the branch in the EPtree which represents the occurrence of b_i with the prefix b_0, b_1,... b_{i-1}. The result, $pred(flip(p,i))$, is the conjunction of the prefix of $pred(p)$ up to c_{i-1} and the negation, $\neg c_i$, of c_i. It is the predicate of the path prefix, $flip(p,i)$, formed by appending the opposite branch, b_i', of b_i to the prefix of b_i. $pred(flip(p,i))$ is submitted to a constraint solver. A solution to this formula gives the input values of a new test case which will cover one of the feasible paths with prefix $flip(p,i)$. We say in this case that the flip is *successful*.

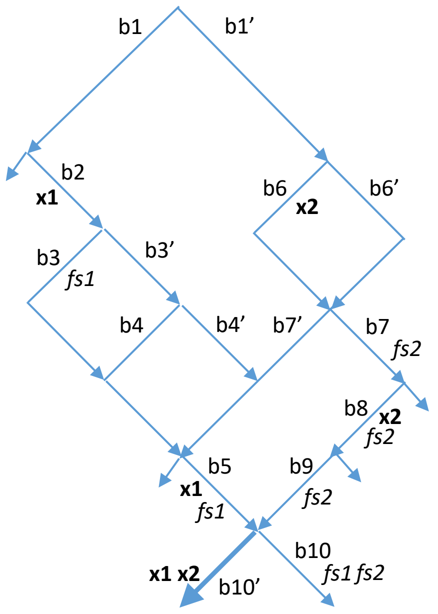

Figure 1 Example of a partial control-flow graph

Consider the example of Figure 1 where b10' is the only uncovered branch left and covered paths include

p1 = `b1,b2,b3,b5,b10,`...

p2 = `b1,b2,b3',b4,b5,b10,`...

p3 = `b1,b2,b3',b4',b5',`...

p4 = `b1',b6,b7,b8,b9,b10,`...

p5 = `b1',b6',b7',b5',`...

Flipping branch `b3` of *p1* resulted in flipped prefix `b1,b2,b3'.` The flip was successful and gave a test-case which covered a path with the flipped prefix, such as *p2*.

If the attempted flip is unsuccessful because the constraint solver finds that *pred(flip(p,i))* is unsatisfiable then we have the demonstration that *flip(p,i)* is an infeasible path prefix. If the flip is unsuccessful because the constraint solver times out, then *pred(flip(p,i))* can be used, if necessary, in a justification of non-coverage of either b_i' or some branch which could, in theory, be reached from b_i'.

The concolic method is a loop which interleaves generation of new test cases (each time a branch is flipped) and exploration of the paths covered by the previously generated test cases, until coverage is complete. Covered paths are feasible by definition, as are all their prefixes, so by flipping a single branch in a feasible prefix, the concolic method limits solver calls by ensuring detection of the shortest prefixes of all the infeasible paths. This is how concolic generation effectively prunes the subtrees rooted in infeasible prefixes from the space to be explored. This is the first reason for the efficiency of concolic generation.

Moreover, because concolic generation always follows a covered path, at each node in the feasible EPtree the solver is called at most once, if and when the branch is flipped. This is the second reason for the efficiency of concolic generation.

Finally, if the concolic method is implemented using an incremental solver and backtracking, then the predicates can be constructed incrementally, calling the solver to add new constraints to an existing constraint store at each node. Indeed for each node, n_i, in the EPtree there is a unique path prefix from the root to n_i and its predicate, which is shared by all paths in the subtree rooted in n_i, can be stored on a stack of constraint stores and recovered by backtracking. This sharing of the predicates of path prefixes is the final reason for the efficiency of the concolic method.

The unique feature of the concolic method is that the constraint solver returns one arbitrary solution from the set of solutions and so (unless the set is a singleton) we do not know which path will be covered by the test case corresponding to the chosen solution. In the case of path coverage, this does not matter because all feasible paths must be covered in any case. When using concolic generation for branch coverage, the choice of which branch to flip next is very important and yet made difficult by our ignorance of the suffix that will be covered in the case of a successful flip. Indeed, if the opposite branch at a particular node is an instance of source code branch, b_i', which is not covered yet (such as `b10'` in our example) then a successful flip of b_i will certainly increase branch coverage because b_i' will be covered and the suffix may also include other uncovered branches. However, if b_i' is already covered, a successful flip of b_i may result in a case which covers no new branches. In our example, flipping `b7` in *p4* might cover a path going through `b5'` or through `b5,b10` or `b5,b10'`. We call a flip in the case where the opposite branch is already covered a *hopeful flip*. A hopeful flip with no new coverage can be useful if it reaches a new part of the tree and enables a subsequent flip which increases coverage (e.g. if a new path through `b5,b10` is covered by the hopeful flip above and then `b10` is eagerly flipped with this prefix) but this is uncertain and we want to limit the number of test cases with no new coverage. One reason is that limiting the number of generated test cases limits the time to treat each case and so total test generation time. Another consideration is that constraint solving is NP-hard and may run until timeout. In order to limit test generation time, we should therefore also try to limit the number of solver calls and certainly avoid calling the solver several times on the same unsatisfiable problem. In fact, in real-life code most constraints are trivial and easily solved or found unsatisfiable, even by syntactic checks. Nonetheless, it is hard to ensure that pathological constraint solving problems will not be encountered at some point, endangering exhaustive coverage, especially if they are treated repeatedly.

4 DECIDING WHICH BRANCH TO FLIP NEXT: THE MTELSE STRATEGY

The considerations above led us to design a concolic generation strategy called *MTelse*, to ensure that hopeful flipping was only performed when all branches with uncovered opposites had already been flipped and thereby cut down on calls to the solver. In [11] we compared this strategy to several others, including classic depth-first concolic generation. The results show a correlation between the number of hopeful flips and solver calls.

Moreover, MTelse ensures the lowest number of hopeful flips of all compared strategies on most of the examples, and fewer hopeful flips than depth-first on all examples.

Here we describe the concolic MTelse strategy and how it evolved from the starting point of the classic depth-first strategy with coverage of all the reachable branches as a stopping criterion. Remember that each time a new test case is generated from the predicate of a flipped prefix *flip(p,i)* (or, in the special case of the first test case of all, from the predicate encoding the precondition), then we recover the predicate *pred(flip(p,i))* and the suffix, *s*, of *flip(p,i)* covered by the new case and, if *s* is non-empty, traverse the branches b_j,... in *s*. At each branch, b_n, of *s*, if we choose to flip, then we add $\neg c_n$ to the predicate and then try to resolve it. If we choose not to flip then we add c_n and proceed to the next branch, b_{n+1}.

Let us now consider the test generation strategy as an exploration of the EPtree, in which each branch is an instance of a source code branch. In a depth-first strategy, the generator starts by traversing the path fragment in the EPtree which represents *s* from the start towards the end (i.e. a leaf) without flipping any branches. The first branch in *s* to be flipped is the final branch before the leaf. If this causes a new test case to be generated then the new suffix, *s*', is then treated, as well as all suffixes generated from *s*'. After that, the generator backtracks up the branches in *s*, from the last towards the first, flipping each branch in the same way. Test generation stops when either all source code branches have been covered or no EPtree branches are left to flip in any suffix. At this point, any source code branches which are still uncovered must be unreachable, unless the solver has timed out at some point, see [12].

The MTelse strategy differs from depth-first in several ways:

- If the opposite source code branch, *b*', is not covered yet, then its representative in the EPtree is *eagerly* flipped on the first traversal down towards the leaf. Hopeful flipping is not performed on traversal in this direction.
- During backtrack up *s*, we perform hopeful flipping of the remaining (i.e. whose opposite is already covered) branches on one condition: we first check whether there is any chance of covering an uncovered branch, i.e. whether any EPtree branches in the subtree rooted at the flipped branch represent an occurrence of an uncovered source code branch.
- To achieve true breadth-first search, and avoid performing hopeful flipping of branches in newly-covered suffixes before eager flipping of branches in previously covered suffixes, we use multi-threading (MT). Each thread treats one (non-empty) covered suffix and threads are classified as high-priority while the treatment advances down *s* and then become low-priority when the treatment backtracks up *s* for hopeful flipping. Low-priority threads are only activated while no threads are still high-priority. The information on which branches are covered is shared between threads.

MTelse reduces the number of hopeful flips in most cases, but cannot avoid another problem posed by the concolic method, and which applies to both eager and hopeful flipping: the danger that because of redundancy in the tree of execution paths, constraint solving repeatedly (and maybe very slowly) fails on the same set of unsatisfiable constraints.

5 CONFLICT LEARNING

To counter this problem, we take inspiration from SAT-solving and previous work in the literature and analyse and memorise the reasons for the unsatisfiability of infeasible path prefixes. We lift the resulting conflict sets from the level of the constraints to that of source code branches. *Conflicts* are therefore ordered sets of source code branches. This enables us to check, before calling the solver to find a test covering a particular path prefix, whether the prefix contains a learnt conflict which makes it infeasible. Our implementation of conflict learning involves several steps, all based on the same principle, which is to avoid repeating the same calculations. We now describe how we proceed.

5.1 Finding unsatisfiable cores

Each time a predicate, *pred(p)*, where *p* is some partial path $b_0,...,b_n$, is found to be unsatisfiable by the solver, we incrementally construct a new conjunction of constraints in order to find one or more *unsatisfiable cores*, i.e. minimal unsatisfiable subsets of the set of constraints, which are sufficient to cause the unsatisfiability of the predicate.

The calculation of the unsatisfiable cores reveals information which can be reused later, as described below. This is why we do not rely on the solver to find the unsatisfiable cores but find them ourselves as follows:

1. First we find the *minimal infeasible sequence*, *mis(p)*, of *p* by operating *back-substitution* over the elements of *p*, from b_n towards b_0. Back-substitution projects the branch constraints backwards over the previous values of the source code variables and builds, for successive branches b_i, a conjunction of constraints which is the *path-based weakest precondition*, of the path fragment $b_i,...,b_n$ and which we call $wp(b_i,...,b_n)$. Each $wp(b_i,...,b_n)$ is associated with one mapping between the constraints it contains and the branches $b_i,...,b_n$ and another mapping between the symbolic constrained variables and the corresponding source code variable values. Mutable source code variables are renamed whenever they take a new value, as in Section 2.
   a) For each branch, b_i, the branch condition is translated into a constraint over the values (at the EPtree node with prefix $b_0,...,b_{i-1}$) of the relevant source code variable values. The resulting constraint is added to $wp(b_i,...,b_n)$.
   b) For each assignment, *lhs=rhs;*, if $wp(b_i,...,b_n)$ contains constraints over the symbolic variable lhs_s representing the value of *lhs* at this point then all occurrences of lhs_s in $wp(b_i,...,b_n)$ are replaced by a new symbolic variable representing the value of *rhs* at this point. Otherwise, the assignment is ignored.
2. At each branch, b_i, (except b_n, the first to be treated), we call the solver to check the satisfiability of $wp(b_i,...,b_n)$.

3. If *wp(b_i,...,b_n)* is satisfiable, then we memorise it, along with the corresponding path fragment b_i,...,b_n, which we call a *feasible sequence*.
4. If *wp(b_i,...,b_n)* is unsatisfiable, then we have found *mis(p)*, which is just b_i,...,b_n, i.e. the unsatisfiability of *p* can be due to just the constraints from b_i,...,b_n. We now stop back-substitution and try to find one or more unsatisfiable cores in *wp(b_i,...,b_n)*. We do this by removing constraints one by one from *wp(b_i,...,b_n)* and re-checking satisfiability. We obtain one or more unsatisfiable cores: i.e. ordered subsets of *wp(b_i,...,b_n)* containing the minimal number of constraints for unsatisfiability. For each unsatisfiable core, we map the constraints to the corresponding subset *core(mis(p))* of branches from *mis(p)*. The branches of each *core(mis(p))* are in the same order as which they appear in *mis(p)* but they may not be contiguous in *mis(p)*.

5.2 Transforming Unsat Cores into Conflicts

We now insert extra branches into each *core(mis(p))*, as necessary in order to protect the *def-use links*, see [4]. Indeed, the condition of a particular branch, *b*, may depend in *mis(p)* on the value of source code variable *v* set by a previous assignment *a*, i.e. there is a def-use link between *a* and *b*. Note that if *a* is *dominated* by another branch, b_a, then b_a will already be included in *core(mis(p))*. Now, another path fragment, *f*, in the EPtree may cover the same assignment *a* and the same branch *b* but, in between the two, contain an additional assignment *a1*, dominated by additional source code branch b_{a1} (in this case there is always a dominating branch because *a1* is not present in *mis(p)*), which subsumes *a* and gives *v* a different value. The danger is that *f* contains *core(mis(p))* but is not infeasible, because it has broken the def-use link between *a* and *b*. To avoid this problem, we find all branches such as b_{a1} which dominate possible re-assignments of variables in def-use links between elements of *core(mis(p))*. The opposites, b_{a1}' of all such branches b_{a1} must have been in *mis(p)* and we add them to *core(mis(p))* (respecting the order between branches in *mis(p)*). After insertion of these supplementary branches, we obtain an ordered superset of *core(mis(p))* which is still an ordered subset of *mis(p)* but also protects the def-use links. This is what we call a conflict. Any other path fragment containing this conflict will also be infeasible.

We can now check, before flipping any branch, whether the flipped prefix contains a conflict. Moreover, before hopefully flipping a branch in an attempt to cover another, uncovered, branch, *u*, we can check whether the result of appending *u* to the flipped prefix contains any conflicts. We should only perform hopeful flipping if there may be a path from *b*' to some *u* which is not *in a conflict with* the flipped prefix (i.e. which does not contain a conflict when appended to the flipped prefix).

5.3 Propagating Conflicts

We also use information from the code structure to propagate and combine conflicts in order to create new ones, which we call *lemmas*. A simple example is the propagation of all conflicts ending in a particular branch, *b*, to all branches dominated by *b*. Another example is the combination of the pair of conflicts b_k,b_j and b_j',b_i: if all paths from b_k go through either b_j or b_j' then we can imply the new conflict b_k,b_i. If the result of combining conflicts is a conflict containing just one branch, *u*, then *u* is unreachable.

6 REUSING FEASIBLE SEQUENCES

Feasible sequences are a by-product of conflict learning as described in Section 5.1 and they can be reused.

Firstly, when finding a new unsatisfiable core as described in Section 5.1, we may analyse a path fragment, *s*, which ends in a known feasible sequence, *fs*. There is no need to re-perform back-substitution on this part of *s*, we can just start from *fs* and its known path-based weakest precondition.

Secondly, when deciding whether to flip a branch *b*, if a feasible sequence, *fs*, starts at *b*' and ends with an uncovered branch, *u*, then we can add the path-based weakest precondition of *fs* to the predicate of the prefix up to *b*' (after mapping the symbolic variables in the constraints of *fs* to the corresponding symbolic variables in the predicate) and call the solver. If the (satisfiable) constraints from the path-based weakest precondition of *fs* are consistent with the (satisfiable) constraints of the predicate of the path prefix to *b*, then a test will be generated which immediately covers *u*. If not, we can continue back-substitution from *fs* to learn the reason for the infeasibility of this new partial path to *u*.

7 LEARNING IN OUR EXAMPLE

Let us return to the example in Figure 1. The flipped prefix of *p1* contains no known conflicts so we eagerly try to flip `b10` but fail and learn the conflict **`x1`**`=b2,b5,b10'` and the feasible sequence *`fs1`*`=b3,b5,b10'`. We consider eagerly flipping `b10` in *p2* but do not try because the flipped fragment would contain **`x1`**. Next, we eagerly try to flip `b10` in *p4* but fail and learn the conflict **`x2`**`=b6,b8,b10'` and the feasible sequence *`fs2`*`=b7,b8,b9,b10'`. There are no more covered paths including `b10` so when all eager flips have been tried we consider hopefully flipping `b5'` in *p3*: `b5` has already been covered but it could lead to `b10'` but the flipped prefix includes **`x1`** so we don't try this hopeful flip. We then consider hopefully flipping `b7` in *p4*: the flipped prefix contains no known conflicts and we can append the feasible subsequence `b5,b10'` of `fs1` to `b1',b6,b7'` in order to cover `b10'`, also without including any known conflicts. We recover the stored path-based weakest precondition of `b5,b10'`, and add it to the constraints for the flipped prefix in order to generate a solution covering `b10'`.

8 ENUMERATING SUFFIXES TO UNCOVERED BRANCHES

We found that the previous measures reduced hopeful flipping but not enough to have a real impact on path explosion. This is why we took the radical decision to eliminate hopeful flipping altogether and enumerate individual path suffixes from previously covered opposite branches to uncovered targets. Path enumeration is clearly combinatorial, it is exactly what is at the root of path explosion, and it can only pay off if conflicts are found fast enough to substantially prune the search space. We now describe how we manage path enumeration to try to ensure that this is the case.

8.1 Building a Conflict-free Suffix

Before starting enumeration of the numerous path suffixes from a flipped prefix towards some uncovered branch, we ensure that at least one suffix exists that is *conflict-free*, i.e. that does not contain internal conflicts or conflicts with the flipped prefix.

Given a branch whose opposite, *b*', is already covered, we traverse the control flow graph from *b*' towards the exit of the tested function until we encounter an uncovered branch, *u*. This traversal results in a skeleton path suffix, *sk*, from *b*' to *u*. *sk* contains just the necessary branches to get from *b*' to *u* while avoiding conflicts. As each new branch is added to *sk*, we check for conflicts between the flipped prefix and the branches in *sk*. We also check for conflicts involving branches on potential paths joining the branches in *sk*, and add branches to *sk* as necessary to avoid these conflicts. If we cannot build a skeleton from *b*' to *u* which has no conflicts, then we try to build a skeleton to the next uncovered branch that we may be able to reach from *b*', and so on. If we succeed in building *sk* for some *u* then it will be contained in any full conflict-free suffix from *b*' to *u*.

Next, we start enumerating all the conflict-free suffixes from *b*' to *u* by fleshing out *sk*. The full suffix contains additional branches, between those of *sk*, and we select these arbitrarily, checking that each one does not induce some conflict which was discovered since *sk* was built.

8.2 Back-substitution of the Suffix

When we find a conflict-free suffix, *s*, from *b*' to *u*, we then apply back-substitution as in Section 5.1 to check its internal consistency. If part of *s* is found to be unsatisfiable then we learn the new conflict and enumerate the next possible suffix from *b*' to *u*. If not, we add the path-based weakest-precondition which is the result of back-substitution of *s* to the predicate of the flipped prefix and try to generate a new test to immediately cover *u*.

8.3 Over-approximate Conflicts

If all suffixes from *b*' to *u* are found to be infeasible, then the flipped prefix must contain a set of branches which makes *u* unreachable from *b*'. We store and reuse this information as follows. We construct the union of *b*' and *u* and all the branches in the flipped prefix which contributed to the different conflicts found during enumeration. This is a conflict which is not necessarily minimal, but which we cannot refine any further, so we call it an *over-approximate conflict*. Before enumerating paths between *b*' and *u* from some new flipped prefix, we check whether it contains an over-approximate conflict with *u*.

9 DISCARDING CASES

A final advantage of our backtracking concolic search strategy is that we can reduce the size of the set of test cases which we propose in order to cover the reachable branches. Note that finding the smallest possible set of cases which satisfy a given criterion is a hard problem which we do not claim to solve. We memorise the branches covered by each case and if a case, *e*, generated early on turns out not to cover any branches that are not also covered by a set, *l*, of later cases and if, because of backtracking and low thread priority, the coverage of the cases in *l* is known before treatment of the suffix of *e* ends, then we can exclude *e* from the final set of cases.

10 RESULTS ON TWO REAL-LIFE EXAMPLES

We have implemented TreeFrog as a proof-of-concept prototype extension of PathCrawler [12] and so far we have only implemented conflict learning on a subset of code containing no arrays, pointers, function calls or loops. This is why we first used the Tcas example from [11] to compare TreeFrog to the depth-first and MTelse strategies. Tcas is a control logic function written in C with 80 branches, one of which is non-trivially unreachable.

Table 1 Results of different strategies on the Tcas example

strategy	cases	discarded	solver calls	eager flips
depth-1st	404	381	420	18
MTelse	395	374	411	18
TreeFrog	19	1	20	16

Table 1 compares, for exhaustive coverage, the total number of cases generated, the number which were discarded from the final set, the number of branches flipped eagerly and the total number of solver calls to flip branches. We averaged the results over 10 runs for the depth-first strategy (because of the non-determinism in the solver) and 100 runs for MTelse and Tree Frog (because of the additional non-determinism due to multi-threading).

401 hopeful flips were performed on Tcas with the depth-first strategy and 392 with MtElse whereas TreeFrog performed 533 satisfiability checks (not counted in the solver calls) on the results of back-substitution in order to detect 52 conflicts and construct 4 new feasible paths to uncovered branches.

Table 2 Results on the Complex example

strategy	cases	discarded	solver calls	eager flips
MTelse	424839	424799	424838	40
TreeFrog	90	50	248	76

We then applied TreeFrog to Complex, another example of a C function containing 270 branches of which 2 are non-trivially unreachable, see Table 2. The depth-first strategy timed out long before running to completion but the MTelse strategy took about half an hour (on an Intel Corei9 machine with 64GB RAM) to perform exhaustive branch coverage and TreeFrog took around 3 minutes on the same machine. At the cost of 16022 satisfiability checks on the results of back-substitution, TreeFrog found 246 conflicts and constructed 170 new feasible paths to uncovered branches. On this example, one of the test cases generated after construction of a new feasible path to an uncovered branch enabled eager flipping of another uncovered branch, i.e. TreeFrog effectively interleaved concolic generation with enumeration.

11 RELATED WORK

There has been much previous work on combatting path explosion in test generation based on symbolic execution but some approaches, such as [13], are more suited to bug-finding than exhaustive coverage because they aim to achieve a high level of coverage as fast as possible, but do not try to account for uncovered test objectives and may even repeat previous calculations.

Other approaches, such as [10], try to partition the search space, e.g. by using function summaries, in order to avoid combinatorial explosion. These approaches are orthogonal to ours and they could probably be combined.

In one of the first attempts to combat path explosion by pruning the search space, the RWset tool [2] eliminates duplicated subtrees st_b rooted in a branch b in the case where after b there are no live variables, ie. all the variables used in st_b are defined after b. In order to do this, RWset performs a depth-first analysis of the source code to find the live variables at each point. In our approach, if there are no live variables after a particular branch b, then the first exploration of st_b will either cover the uncovered branches or else find the internal conflicts within the subtree which prevent their coverage. When a different prefix to b is treated, the exploration of st_b will be performed again, unlike in RWset, but will be much faster because all the internal conflicts in the subtree are already known. Moreover, in our approach, even if there are live variables after some branch, b, and so st_b would not be pruned by RWset, conflicts learnt in the first exploration of st_b can still be used in subsequent explorations of the same subtree.

In [4], which inspired the work described here, the reasons for infeasibility are analysed in order to construct *explanations*, which are the same as our conflicts, but back-substitution is not used and feasible sequences are not saved and reused. Rather than using the conflicts to decide whether to flip a branch as we do, they use them to generate an automaton capable of generating, or recognising, all paths which will be infeasible for the same reason.

In [9], as in TreeFrog, previously covered branches are only flipped if they may lead to uncovered branches and sets of conflicting branches are learnt and compared to flipped prefixes. However, the unsat core is recovered from the solver and the tested programs are stateless, which means that it is not necessary to add extra branches to the conflicting set in order to protect the def-use links. In [8], branch conflicts obtained in the same way from stateless programs are treated but as whole paths are treated instead of the path prefixes analysed in concolic generation, a conflict may be in the middle of a path. Instead of analysing path prefixes and suffixes as we do, a data-dependency analysis is performed and the code is dynamically partitioned in order to decide the order in which paths are treated.

The Kite tool [5] computes the same conflicts as we do but Kite's aim is to cover assertion violations rather than branches. The conflicts are derived from the conflict clause returned by the solver. Kite is based on *dynamic symbolic execution* instead of a concolic generator. The connections between branches and sequential blocks in the source code are encoded as constraints so that solutions to these constraints represent execution paths to be explored. This encoding also ensures the preservation of def-use chains in conflicts. Mimicking conflict-driven clause learning, the conflicts are encoded as constraints on combinations of branches. The negation of the constraints encoding the conflicts are added to the constraints encoding paths through the CFG so that only new paths containing no conflicts will be accepted. This elegantly achieves the same result as our enumeration of conflict-free suffixes but we invoke enumeration as a last choice, after concolic generation, whereas in Kite it is the sole mechanism used for test generation. Kite cannot reuse feasible sequences as we do, nor construct lemmas or overapproximate conflicts.

The BiTe [1] test generation tool for branch coverage does not learn conflicts but does combine concolic generation with reachability analysis, i.e. the construction of weakest preconditions which are more general than our path-based weakest preconditions because they encompass all possible path prefixes. BiTe starts with a concolic test generation phase and then calculates the weakest precondition of the remaining uncovered branches. It maintains a graph of program states annotated with the information calculated by symbolic execution, including infeasible prefixes, and reachability analysis and uses this to guide further symbolic execution in order to combine a weakest precondition with a path prefix predicate, rather as we reuse a feasible sequence, in order to try and cover a new branch.

Another response to path explosion in concolic testing has been the use of *interpolants*. Interpolants can be used to abstract the subtree rooted at a node in the EPtree. Unlike our branch conflicts, they are expressed directly in terms of the underlying constraints. This means that, unlike in our work, interpolants can be used when different branch combinations give rise to the same constraints. Interpolants are a potentially powerful tool but it is difficult to find the best way to combine them with test generation based on symbolic execution, especially for branch coverage. Indeed, an interpolant must be calculated at some point in test generation and express a property of the subtree which is useful at a later point. Moreover, interpolants can only been calculated once the entire subtree has been explored. As a result, [7] proposes using interpolants in concolic test generation, but for depth-first path coverage and not branch coverage and by eagerly exploring subtrees in parallel with concolic generation. TracerX [6] extends this work with more sophisticated interpolants and is implemented as an extension of dynamic symbolic execution instead of a concolic test generator. In these papers, the interpolant expresses conditions for feasible coverage of at least one assertion violation or runtime error in the subtree. Note that we check just the conflicts which apply to some currently uncovered branch but interpolants are specific to subtrees and not to test objectives so an interpolant may concern several test objectives, some of which may already have been covered. The TracerX tool was applied to different problems: proof of the reachability of a single test objective (where it was compared to the CMBC model-checker) and coverage of basic blocks, thereby demonstrating the link between branch coverage and other verification problems that we mention in Section 1.

12 CONCLUSIONS

We describe the TreeFrog test generation method which tries to retain the advantages of concolic test generation while using conflict learning to prune the search space and achieve exhaustive branch coverage. In TreeFrog, we lift conflicts, as well as the calculation of unsatisfiable cores, to the level of atomic branches in the simplified source code. We also replace hopeful flipping by reuse of learnt feasible branch sequences or controlled enumeration of paths to uncovered branches. Path enumeration causes a combinatorial explosion so can only be deployed if conflict learning enables substantial pruning of the space over which enumeration takes place. This is the case in the examples we have tried so far, in which the results are dramatically improved compared to concolic generation without conflict learning.

TreeFrog makes extensive use of incremental constraint solving and backtracking for increased efficiency, as described above. Moreover, we use multi-threading to ensure that instances of branches which are already covered are treated last.

TreeFrog can currently only learn and apply conflicts in code fragments with no loops, function calls, pointers or arrays. Treating pointers and arrays complicates back-substitution but is a well-known problem. Loops and function calls cause the same branch to have several locations in the simplified source code so that branches can no longer be identified solely by their location. Moreover, the treatment of loops (and recursive functions) poses the problem of how and when to terminate loop unrolling (or recursion) during path enumeration. We must now implement the treatment of these constructions in order to be able to apply TreeFrog to more examples for further evaluation.

Note that TreeFrog reduces the number of calls to the solver in order to flip branches (i.e. resolve path predicates) but at the expense of a large number of satisfiability checks during back-substitution. We assume that these checks are performed on short conjunctions of constraints over small numbers of variables and so are relatively inexpensive. In future work, we will try to assess whether this assumption is justified.

One shortcoming of our procedure, described in Section 5.1, for finding unsatisfiable cores is that it is based on the shortest infeasible fragment, *mis(p)*. This means that we do neglect the possibility that a longer infeasible fragment of *p* contains a different mutually inconsistent set of constraints and so we do risk missing other conflicts in *p*, which can only be revealed by another subsequent infeasible prefix. We need to evaluate in future work whether it would be more efficient to ensure we detect all possible unsatisfiable cores in *p*.

Another concern which we would like to address in future work, is that TreeFrog cannot learn a conflict which might cause the solver to timeout on a predicate rather than declare it unsatisfiable.

Finally, learning branch conflicts is a time-consuming way to learn the same unsatisfiable combination of constraints when it is is induced by several different sets of branches. This is why it might be worthwhile to extend our method to learn and avoid some unsatisfiable subsets of constraints as well as branch conflicts.

ACKNOWLEDGMENTS

This work was partially supported by ANR grant ANR-18-CE25-0015-01.

REFERENCES

[1] M. Baluda, G. Denaro, M. Pezzè. Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Software Eng., 42(5):403–426, 2016, doi:10.1109/TSE.2015.2490067.

[2] P. Boonstoppel, C. Cadar, D.R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In Proc. TACAS 2008, Lecture Notes in Computer Science vol. 4963, 351–366. Springer, 2008. doi:10.1007/978-3-540-78800-3_27.

[3] A.R. Bradley. Sat-based model checking without unrolling. In Proc. VMCAI 2011, Lecture Notes in Computer Science, vol. 6538, 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

[4] M. Delahaye, B. Botella, A. Gotlieb. Explanation-based generalization of infeasible path. In Proc. 3rd Intl. Conf. on Software Testing, Verification and Validation, ICST 2010, 215–224. IEEE, 2010. doi:10.1109/ICST.2010.13.

[5] C. Gomes do Val. Conflict-driven symbolic execution: How to learn to get better. Master′s thesis, University of British Columbia, Vancouver, 2014. doi:10.14288/1.0165906.

[6] J. Jaffar, R. Maghareh, S. Godboley, X. Ha. Tracerx: Dynamic symbolic execution with interpolation. In Proc. FASE 2020, Lecture Notes in Computer Science, vol 12076, 530–534. Springer, 2020. doi:10.1007/978-3-030-45234-6_28.

[7] J. Jaffar, V. Murali, J.A. Navas. Boosting concolic testing via interpolation. In Proc. Joint Meeting of the European Software Engineering Conf. and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE′13), ACM, 2013, 48-58, doi:10.1145/2491411.2491425.

[8] B.A. Marcellino, M.S. Hsiao. Dynamic partitioning strategy to enhance symbolic execution. In Proc. 2016 Design, Automation & Test in Europe Conf. (DATE 2016), IEEE, 2016, 774–779.

[9] S. Prabhu et al. An efficient 2-phase strategy to achieve high branch coverage. In Proc. 20th IEEE Asian Test Symposium (ATS 2011), IEEE, 167 –174. doi:10.1109/ATS.2011.83.

[10] R. Qiu, G. Yang, C.S. Pasareanu, S. Khurshid. Compositional symbolic execution with memoized replay. In Proc. 37th IEEE/ACM Intl. Conf. on Software Engineering, ICSE 2015, IEEE, Vol. 1, 632–642. doi:10.1109/ICSE.2015.79.

[11] N. Williams. Towards exhaustive branch coverage with PathCrawler. In Proc. 2021 IEEE/ACM Intl. Conf. on Automation of Software Test (AST 2021), IEEE/ACM, 117–120.

[12] N. Williams, B. Marre, P. Mouy. On-the-fly generation of k-path tests for C functions. In Proc. 19th IEEE Intl. Conf. on Automated Software Engineering (ASE 2004), IEEE, 290–293. doi:10.1109/ASE. 2004.10020.

[13] T. Xie, N. Tillmann, J. de Halleux, W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In Proc. 2009 IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN 2009), IEEE, 359–368, doi:10.1109/DSN.2009.5270315