2014 Eighth International Conference on Software Security and Reliability

Automated Coverage-Driven Test Data Generation Using Dynamic Symbolic Execution

Ting Su, Geguang Pu, Bin Fang, Jifeng He
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
Email: tsu,ggpu,bfang,jfhe@sei.ecnu.edu.cn

Jun Yan
State Key Laboratory of Computer Science,
Chinese Academy of Sciences, Beijing, China
Email: yanjun@otcaix.iscas.ac.cn

Siyuan Jiang
Department of Computer Science and Engineering,
University of Notre Dame, USA
Email: sjiang1@nd.edu

Jianjun Zhao
School of Software,
Shanghai Jiao Tong University, Shanghai, China
Email: zhao-jj@cs.sjtu.edu.cn

***Abstract*—Recently code transformations or tailored fitness functions are adopted to achieve coverage (structural or logical criterion) driven testing to ensure software reliability. However, some internal threats like negative impacts on underlying search strategies or local maximum exist. So we propose a dynamic symbolic execution (DSE) based framework combined with a *path filtering* algorithm and a new heuristic path search strategy, i.e., *predictive path search*, to achieve faster coverage-driven testing with lower testing cost. The empirical experiments (three open source projects and two industrial projects) show that our approach is effective and efficient. For the open source projects w.r.t branch coverage, our approach in average reduces 25.5% generated test cases and 36.3% solved constraints than the traditional DSE-based approach without path filtering. And the presented heuristic strategy, on the same testing budget, improves the branch coverage by 26.4% and 35.4% than some novel search strategies adopted in KLEE and CREST.**

I. Introduction

Software testing is the most commonly adopted technique to ensure software reliability. Specially, *unit testing* is an important white-box technique to independently check functional correctness of a unit (a unit may contain a number of functions). The traditional manual testing and random testing [1], [2] are usually adopted to deliver test cases, although useful, could only distinguish small parts of all possible program behaviors [3], [4].

Thanks to the recently proposed *Dynamic Symbolic Execution* (DSE) technique [5], [6], the situation was alleviated. This technique traverses program paths as many as possible to automatically generate test cases. It collects the symbolic constraints along the execution path triggered by a concrete input. A variant of the conjunction of these symbolic constraints, i.e., the *path constraint*, is then solved by a constraint solver to output a new test case. This test case will be used as the next input to witness another new program path. In order to achieve high code coverage, this process will continuously iterate by enumerating all possible paths. This technique has been widely used to generate test cases w.r.t statement or branch coverage [5]–[9].

However, in software testing, different coverage (structural or logical [10]) criteria are used to measure test adequacy. For example, for safety critical software, it requires test cases conforming to branch coverage as well as MC/DC (Modified Condition/Decision Coverage) criterion [11]. In the context of DSE, Pandita et al. [12] proposed a trade-off approach to achieve a specified coverage criterion by source code transformations. As a result, the block coverage in the transformed program implies the MC/DC coverage in the original program. This approach is general and easy to be implemented on existing DSE-based tools which internally support block coverage. However, code transformations are language-dependent. It may change the original purposes of the program. In addition, the transformations have to be adapted to inherent search strategies of the DSE engine [13] or even sometimes complicate search strategies.

In this paper, we propose a general DSE-based framework to generate test cases under different coverage criteria. This framework takes a unit under test (UUT) as input and outputs test cases w.r.t. a target criterion. It unifies different criteria through a *unified* structure, i.e., *coverage structure* (CS). This structure is derived from the inter-procedural control flow graph (CFG) of the UUT associated with criterion-specific information. The *key conceptual idea* of CS is to provide the freedom to design efficient underlying search strategies in a DSE engine regardless of the intended coverage criterion.

On the other hand, test budgets are usually constrained in coverage-driven testing. Thus, at the high level, we intend to achieve faster coverage-driven testing with lower testing cost. We design a CS-based path filtering algorithm in our framework to prune path search space. The intuition is that a feasible path candidate may be *irrelevant* w.r.t. some coverage criterion, i.e., it once exercised may not improve code coverage. Thus, we can reduce testing cost by *safely* skipping these *irrelevant* path candidates. It can produce smaller test suites with lower testing cost than the traditional DSE-based approach in which it simply enumerates all paths. It can be dynamically achieved during DSE without static reachability

978-1-4799-4296-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SERE.2014.23

analysis [14].

In addition, we also propose a CS-based heuristic path exploration strategy in our framework, i.e., *predictive path search*. It favors those path candidates which have predictively higher contribution on code coverage. The insight is that when these candidates are exercised, they tend to cover more coverage goals at one blow. So it is promising to achieve reasonable testing budget allocation especially when the budgets (e.g. program iterations or execution time) are constrained. It can also further accelerate the coverage-driven testing. Our coverage-driven testing framework currently supports statement, branch and MC/DC criteria. The tool is available online[1].

This paper makes the following main contributions:

- A coverage-driven DSE-based testing framework is proposed. It supports different (structural or logical) coverage criteria through the unified *coverage structure*. It is easy to be implemented on existing DSE-based tools with different underlying path exploration strategies.
- In this framework, a path filtering algorithm and a new path exploration strategy are implemented to achieve faster coverage-driven testing with lower testing cost. They are effective especially when the testing budgets are constrained. We present details on how to realize them.
- We built this framework on our DSE engine [15]. Empirical experiments are carried out on three open source projects and two real industrial projects. For three open source projects w.r.t branch coverage, the CS-based path filtering algorithm in average reduced 25.5% generated test cases and 36.3% solved constraints than the traditional DSE-based approach without path filtering. And the new CS-based path exploration strategy, on the same testing budget, improves the branch coverage by 26.4% and 35.4% than some novel search strategies adopted in KLEE [7] and CREST [16].

The remainder of this paper is organized as follows. Section 2 gives an illustrative example of our approach. In section 3, we introduce some background and define *coverage structure* followed by the details of the path filtering algorithm and the predictive path search strategy in section 4. Section 5 presents the empirical evaluation results. We discuss the related work in Section 6 and conclude in the last section.

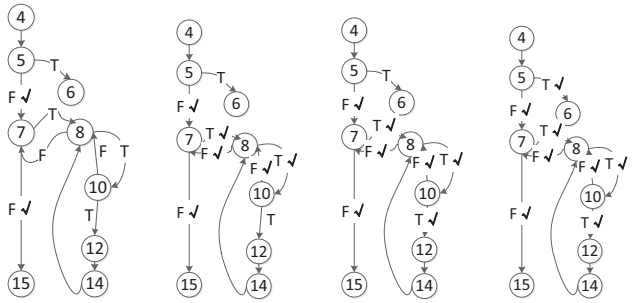
II. Example

In this section, we present an example to illustrate our idea. When testing a unit, our approach follows three main steps, i.e., coverage structure updating, path filtering, and path selection. Figure 1 shows the unit $bubble$ taken from the $flex$ open source project and Figure 2 shows its control flow graph (CFG). This unit realizes the bubble sorting by sorting the first n elements in a given array v and places them in an increasing order.

[1]CAUT: http://www.lab205.org/caut

```
1  #define MAXLEN 6
2  void bubble(int v[MAXLEN],int n)
3  {
4    int i,j,k;
5    if(n>=MAXLEN)
6     return;
7    for(i=n;i>1;--i)
8     for(j=1;j<i;++j)
9      /* compare */
10     if(v[j]>v[j+1])
11     {/* exchange */
12      k=v[j];
13      v[j]=v[j+1];
14      v[j+1]=k;}
15 }
```

Fig. 1. Example *bubble*



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

Fig. 2. Branch Coverage Testing on *bubble*

Here we denote a branch as the line number followed by T or F to respectively represent the true or false one. From the CFG, there are total 8 branches, i.e., $5F$, $5T$, $7F$, $7T$, $8F$, $8T$, $10F$ and $10T$. In the following we illustrate the process under branch coverage testing. Suppose the initial input values of v and n are randomly initialized as $[4, 0, 1, 8, 0, 2]$ and 0, respectively. After we test $bubble$ within our framework, we could get the following test cases and corresponding execution paths:

1) Test 1: (v=[4, 0, 1, 8, 0, 2], n=0) Path 1: (5F, 7F)
2) Test 2: (v=[4, 0, 1, 8, 0, 2], n=3) Path 2: (5F, 7T, 8T, 10F, 8T, 10F, 8F, 7T, 8T, 10F, 8F, 7F)
3) Test 3: (v=[4, 2, 1, 8, 0, 2], n=3) Path 3: (5F, 7T, 8T, 10T, 8T, 10F, 8F, 7T, 8T, 10F, 8F, 7F)
4) Test 4: (v=[4, 2, 1, 8, 0, 2], n=6) Path 4: (5T)

After the first execution on the initial input, we get Path 1 and the branches $5F$ and $7F$ are covered in Figure 2a (indicated by $\surd$). From Path 1, two path candidates are available, i.e., ($5T$) and ($5F$, $7T$), by flipping the branching nodes (the instance of the original branch) for the false branch of Line 5 and Line 7 (in Path 1) respectively to their true branches. We can see that these two path candidates are both relevant w.r.t branch coverage. Because they once exercised may hit the uncovered branches $5T$ or $7T$. Thus no path candidates are skipped. They are both put into the *path pool*, which stores all relevant path

candidates.

In next iteration, we favor the path candidate ($5F$, $7T$) because it is promising to cover more branches than ($5T$) (details on this judgement is explained in the predictive path search strategy (PPS) in section 4). As a result, we would get Path 2 with Test 2 (assume the constraint solver randomly assigns 3 to n to satisfy the path constraint $n > 1 \wedge n < 6$). The four branches $7T$, $8T$, $8F$, $10F$ are covered in Figure 2b at one blow. Next we re-check the remaining path candidates in the *path pool* to filter irrelevant ones. Because after a fruitful program execution, the coverage status of a UUT is changed. Remaining path candidates in the pool may turn relevant to irrelevant. From Figure 2b, it is obvious ($5T$) is still relevant and we keep it. On the other hand, ten new path candidates (no duplicate path candidates are generated) are generated in this iteration by flipping the branching nodes from their original sides to the unexplored sides in Path 2. Nine candidates among them are relevant and stored in the pool except the one ending with $7F$. Because from Figure 2b, only these nine path candidates are possible to reach the uncovered branch $10T$.

We next choose to explore the path candidate ($5F$, $7T$, $8T$, $10T$) generated by flipping the first branching node for the false branch of Line 10 (in Path 2) to the true branch (because it is more promising to cover the $10T$ branch with the shorter length than others), we will get Path 3 with Test 3 hitting $10T$. We re-check the pool and find out all path candidates but ($5T$) are irrelevant. Because according to Figure 2c, those remaining eight path candidates (as well as new candidates generated from Path 3) starting with ($5F$, $7T$) will not hit the last uncovered branch $5T$. So we filter them away and exercise the only remaining candidate ($5T$). At last, we get Path 4 and cover all 8 branches in Figure 2d.

As illustrated in the above example, we generate only 4 test cases with 4 fruitful program iterations in achieving 100% branch coverage. In this example, the breadth-first path selection [8] requires at least 5 program iterations (because it wastes time on exercising the path candidate ($5F$, $7T$, $8F$) which is eventually proved to be infeasible) and the random path selection [7] generates more redundant test cases (because it has no knowledge of path exercising priorities). In section 4, we will explain the predictive path search strategy in more detail to show how to achieve faster coverage-driven testing as illustrated in this example.

III. Coverage Structure

In this section, we give the background of MC/DC coverage criterion and the definition of *coverage structure*.

A. Background

1) Modified Condition/Decision Coverage: MC/DC coverage [11] is one of rigorous logical coverage criteria. It is usually adopted in safety-critical domains. For a program (*P*) under test, its branch predicates are called *Decisions* (*D*), which contains one or more *Conditions* (*C*), i.e., D_p=$C_1 \oplus C_2 ... C_n$ ($\oplus$ stands for $\wedge$ or $\vee$). The testing adequacy of MC/DC coverage criterion requires the following four points:

- Every point of entry and exit in *P* has been invoked at least once.
- For every D_p has taken all possible outcomes.
- For every $C_i \in D_p$ has taken *true* and *false* at least once.
- For every $C_i \in D_p$ has been shown to independently affect D_p's outcome. C_i is shown to independently affect D_p's outcome by varying just C_i while holding fixed all other possible conditions C_j. Namely, we fix the logical value of C_j(j $\neq$ i), and then require D_p(C_i = true) $\neq$ D_p(C_i = false).

B. Coverage Structure

Our framework is based on the unified *coverage structure* (CS) to support coverage-driven testing. It is derived from inter-procedural[2] control flow graph [17] and supports both structural and logical coverage criteria including statement, branch, condition, condition/decision, MC/DC and etc.

Definition 1 (Control Flow Graph): A Control Flow Graph[3] G of program P is a directed graph G=(N,E,s,e), where

- N is a set of nodes representing statements (including *instruction* statements and *conditional* statements) in P.
- E is a binary relation on N (a subset of $N \times N$), referred to as a set of edges which represent the flow of control in P.
- s and e are, respectively, entry and exit nodes, s, $e \in N$.

Definition 2 (Coverage Structure): Coverage Structure S is a CFG G associated with the coverage criterion C, i.e., S=(G, C, $Eval$, Cov), where

- G is a CFG in *Definition 1*.
- $Eval$ is a data structure which maps a CFG node to its all evaluation results. It maintains the evaluation results of all CFG nodes during program executions. For a conditional statement, the evaluation result is its logical evaluation (*true* or *false*). For an instruction statement, the evaluation result is *true* if it is executed or *false* if not executed.
- Cov is a function. It returns a set of new covered *coverage items*[4] in the original program w.r.t C according to the current $Eval$.

In fact, we are interested in specific CFG nodes in G w.r.t the criterion C (e.g., instruction statements w.r.t statement coverage, conditional statements w.r.t branch or MC/DC coverage). We call these interested nodes w.r.t C in the simplified

[2]Inter-procedural CFG is constructed by connecting the function call site statement in a caller to the entry statement in the corresponding callee and connecting the return statement in this callee to the statement immediately following that call site statement.

[3]We adopt a simplified form of CFG obtained by transforming composite decisions to atomic conditional statements. We internally maintain the **mapping** relations between these statements and their original decisions. Because we intend to measure the test adequacy on the original program instead of the simplified program. But we conduct symbolic execution on the simplified program.

[4]In the original program, we concern about instructions, branches or decisions w.r.t different coverage criteria. We call them *coverage item*. But note the CS is constructed from the simplified program.

program as *CS nodes* or *coverage goals*. Take $bubble$ as an example, there are four CS nodes (four conditional statements located at Line 5, 7, 8 and 10) w.r.t the branch criterion. When a conditional node has the *true* or *false* evaluation, Cov will regard its corresponding branch (in the original program) as covered. In the following code snippet, *if*$((A \wedge B) \vee C)$ {s1;} *else* {s2;}, three conditional nodes[5] (A, B, C) are concerned w.r.t the MC/DC criterion. Cov will decide whether the corresponding decision node satisfies the test adequacy by computing the available combinations[6] of these conditions' evaluations in $Eval$.

The coverage status of CS is computed from the coverage status of all CS nodes. It is updated during program executions. The test adequacy of the original program is measured by $Cov(C,Eval)$. If a path candidate fails to reach coverage items of the original program, it is regarded as *irrelevant*. We will discuss the details in the next section. Another possible application of CS is to generate test cases for interested code parts (e.g., in regression testing [18]). We can simply set irrelevant nodes as *covered*. This can prune enormous irrelevant path candidates which can not reach the interested code parts.

IV. Approach

The idea behind our approach is that a feasible path candidate may be *irrelevant* w.r.t. some coverage criterion. We can *safely* skip them to avoid wasting testing budgets. In addition, we can select the remaining *relevant* path candidates with different priorities to further accelerate coverage testing. In this section, we are going to present this coverage-driven testing framework in detail. We give the overview of our approach followed by the path filtering algorithm and the predictive path search strategy.

A. Overview of our approach

Algorithm 1 gives the outline of our approach. Initially, the path pool σ is empty (denoted by ϕ) and the input vector τ is randomly initialized as τ_0. After an execution path Π is explored under τ (Line 5). The coverage status of CS will be updated according to Π. The engine will collect all available path candidates (i.e., path prefixes [19]) (Line 7). CS will be used as a filter to eliminate irrelevant path candidates among them w.r.t. the target criterion. The remaining relevant paths will be added into σ. Then we adopt some path search strategy Ψ to select a candidate P to continue the coverage-driven testing until no candidates are available or the code coverage satisfies some level β. Here the framework is general and able to deal with different coverage criteria under different search strategies. Because the search strategy Ψ does not need to concern about the target criterion. CS is responsible for maintaining the criterion-specific information.

[5]In the simplified CFG, this decision corresponding to three atomic conditional statements.

[6]According to the MC/DC definition, at least four test cases are needed to satisfy the test adequacy on this composite decision. Refer to http://www.verifysoft.com/en_example_mcdc.html for details.

In the following, we describe the *CS Updating* procedure and the *Path Filtering* algorithm.

Algorithm 1 Coverage-driven Testing Framework

```
1: Input: σ: the path candidates pool, τ: the input vector, Ψ: the
   search strategy, CS: the coverage structure
2: Output: TEST: test cases, CS: the updated coverage structure
3: σ ← φ, τ ← τ0 /*initialize inputs before the main loop*/
4: repeat
5:    Π ← DSE_EXEC(τ) /*concrete and symbolic execution*/
6:    CS ← UpdateCS(CS,Π) /* update coverage structure */
7:    path_set ← CollectPathCandidates(Π) /*collect all available
      path candidates*/
8:    σ ← FilterPath(CS,σ,path_set) /*filter irrelevant path candi-
      dates*/
9:    P ← SelectNextCandidate(σ,Ψ) /*select a path candidate*/
10:   τ ← SolveConstraints(P) /*solve the path constraint of P, τ
      is a new generated test case*/
11:   TEST ← TEST + {τ}
12: until σ=φ ⋁ CoveragePercentage≥β
```

B. CS Updating and Path Filtering Algorithms

In CS, a conditional CS node is *covered* if it has both the *true* and *false* evaluations. An instruction CS node is covered if it has the *true* evaluation.

CS Updating. This procedure (at Line 6 in Algorithm 1) aims to update the coverage status of the CS w.r.t the target criterion. The updated CS will be used to filter irrelevant path candidates before path selection. It takes an explored path Π and the CS as input, and return the updated CS. Internally, $Eval$ will be updated during each program iteration by adding new evaluations from those CS nodes along Π. During this procedure, the code coverage of the original program under test is also updated according to $Cov(C,Eval)$.

Algorithm 2 FilterPath: Path Filtering Algorithm

```
1: Input: CS: the coverage structure, σ: the path candidates pool,
   path_set: the path candidates set
2: Output: σ: the updated path candidates pool
3: for all P ∈ path_set do
4:    /* P is a partial path not a complete execution path*/
5:    if isRelevant(P)=true then
6:       σ.push(P) /*add into the pool*/
7:    end if
8: end for
9: function isRelevant(path P)
10: /* sequentially check the CS nodes along the path P */
11: for all cs_node_i ∈ P do
12:    new_eval ← getEval(P, cs_node_i)) ∪ Eval
13:    if Cov(C,new_eval)!=φ then
14:       /* Cov returns new covered coverage items in the original
          program, otherwise it returns empty, i.e., φ */
15:       return true /* a must-relevant path */
16:    end if
17:    if isCovered(cs_node_i ∪ cs_node_i.Succs)=true then
18:       /* the CS node itself and its successors are all covered */
19:       return false /* an irrelevant path */
20:    end if
21: end for
22: return true/* a may-relevant path */
```

Path Filtering. Algorithm 2 shows the path filtering algorithm. Lines 5 checks whether a candidate is relevant w.r.t C by function *isRelevant*. A candidate is *relevant* if it will *probably* reach new coverage items of the original program. Otherwise, a candidate is *irrelevant* w.r.t C. At Line 12, we get the evaluation result of cs_node_i from P by function *getEval*, and the result from combining the new evaluation result of cs_node_i and the original $Eval$ in CS are stored in the variable *new_eval*. If new coverage items in the original program could be covered under *new_eval* (Line 13-16), *P* is a *must-relevant* path candidate. If there exists a CS node itself and its successors are all covered (Line 17-20), P will not reach any uncovered items by following it. In other words, it will not improve code coverage of the original program. So *P* is an *irrelevant* candidate. In the remaining circumstances (e.g. P *may* reach uncovered CS nodes), *P* is a *may-relevant*[7] candidate (Line 22).

In the $bubble$ example, w.r.t branch criterion, after Path 1 ($5F$, $7F$) is executed, we get two path candidates ($5T$) and ($5F$, $7T$). From Path 1, the CS node located at Line 5 get the $false$ evaluation. From the candidate ($5T$), we can see its new evaluation result is $true$. According to Cov, this coverage item (i.e., n>=MAXLEN) at Line 5 could be covered if the $true$ and $false$ evaluation are both available. So ($5T$) is *must-relevant*. Similarly, ($5F$, $7T$) is also *must-relevant*. After Path 2 is executed, the candidate ($5F$, $7T$, $8F$) is available. No new coverage items could be covered according to Cov. But the CS node at Line 10 (a successor of the CS node at Line 8) still remains uncovered. So the candidate is *may-relevant* (because it may reach the uncovered branch $10T$). After Path 3 is explored, the CS node at Line 7 and its successors are all covered, so all remaining candidates starting with the prefix ($5F$, $7T$) are *irrelevant* and filtered away.

In Algorithm 2, only *irrelevant* path candidates are filtered away. In addition, after one fruitful path exploration, all remaining candidates in *path pool* will be re-checked to remove those candidates turning must-or-may relevant to irrelevant.

Discussion. An early heuristic on pruning irrelevant paths was described in [20]. But our path filtering algorithm differs from it in several points: (1) The heuristic is implemented on static symbolic execution while ours is built within a dynamic symbolic execution based framework. (2) The heuristic focuses on branch coverage. It does not maintain any data structures (like $Eval$ in our approach) to record the program execution history. So it is not easy to prune irrelevant paths w.r.t other criteria (especially logical criteria). (3) In addition, our algorithm distinguishes must-relevant candidates from may-relevant ones in order to prioritize relevant ones (will be discussed in the next section) when testing budgets are constrained. To our knowledge, we are the first to apply path filtering to achieve faster DSE-based coverage-driven testing.

[7]It is not easy to rule out certain P because some branches on it may affect the reachability towards other branches later in the program. So we bias efficiency instead of precision under this circumstance.

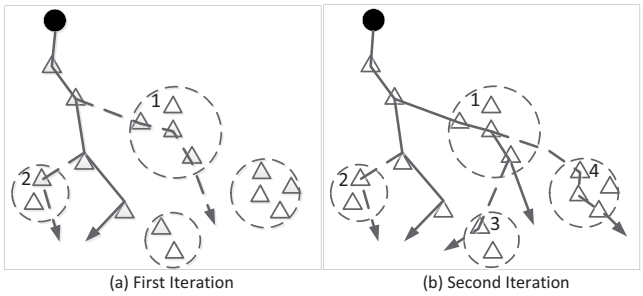


Fig. 3. Predictive Path Search

C. Predictive Path Search

Earlier DSE-based testing tools [5], [6] adopt the depth-first path exploration strategy. The DFS strategy always explores the new path by negating the last predicate of the previous path constraint. So it is likely to be trapped in a small code area (e.g. in the presence of infinite loop unfolding) during path exploration. Testing budgets (e.g., program iterations) are wasted without hitting new coverage goals.

To cope with this disadvantage, we propose the *predictive path search* (PPS) strategy. The intuition behind PPS is that if we give a higher priority to explore those path candidates which tend to cover more coverage goals *at one blow* (i.e., collateral coverage [21]), the program iterations as well as generated test cases will be considerably reduced. It is also promising to achieve reasonable testing budget allocation especially when the budgets are limited. In other words, PPS is intended to drive the path exploration to the code areas with more *dense* coverage goals. As illustrated in Figure 3, the filled circle represents the initial program state and the triangles stand for coverage goals in the program under test. After the first path (indicated by the solid line) is explored, the PPS strategy will predict the density of coverage goals in a *local* area (indicated by dashed circles) along the exploration path. For example, it favors area 1 over 2 in Figure 3a (dotted lines indicate *possible* execution paths after exercising the corresponding path candidates) because area 1 contains more new coverage goals than area 2. For the same reason, it chooses area 4 over 3 in the next iteration in Figure 3b.

In the following, we detail the PPS strategy in the context of branch coverage to simplify the explanation. It is also suitable for other criteria like MC/DC coverage. Note that the CS is constructed at inter-procedural level in PPS.

Path Candidates Evaluation. In PPS, we define $PotentialContr(P)$ to predict the number of potential coverage goals that may be covered by a relevant (must-relevant or may-relevant) path candidate P. It is statically evaluated from CS.

$$PotentialContr(P) = UncoveredGoals(\bar{p_l}, WalkDown(\bar{p_l}, s))$$

Here, $\bar{p_l}$ is the opposite branch of the last CS node on P. $WalkDown(\bar{p_l}, s)$ is the evaluation method by walking down s levels starting from $\bar{p_l}$ on CS at inter-procedural level.

Here, a level corresponds to the level where a CS node (it could be an *instruction* or a *conditional* statement) locates at. It counts the uncovered goals between the start level ($\bar{p_l}$'s level) and the target level ($\bar{p_l}$'s level plus s). The final result is $PotentialContr(P)$'s value. Here we restrict the prediction in a *local* area instead of simply counting the number of all uncovered goals which are transitively control-dependent [22] on $\bar{p_l}$. Because local prediction is intuitively more precise and cheaper than global prediction and we usually have no idea about the actual explored paths after exercising P from $\bar{p_l}$. Global prediction can be regarded as a special case when $s \to \infty$ in PPS.

Path Candidates Selection. We favor the path candidate with bigger $PotentialContr(P)$ value. Because it is likely to hit more coverage goals at one execution. If two path candidates have the same $PotentialContr(P)$ value, the shorter one is preferred. After each iteration, $PotentialContr(P)$ will be re-computed for all remaining path candidates. In particular, during the path selection, we always choose must-relevant candidates before may-relevant ones to eagerly climb towards fruitful direction. Although P may turn must-relevant to may-relevant when new goals are covered. The must-relevant candidate queue always has a higher priority than the may-relevant one. It could avoid possible *local maxima* in the PPS strategy.

In the *bubble* example, from Path 1 we get two candidates ($5T$) and ($5F$, $7T$). Under the PPS strategy, ($5F$, $7T$) has a higher priority than ($5T$) because the former can potentially hit at least 3 uncovered branches ($7T$, $8T$ and $8F$) while the latter at most 1 uncovered branch ($5F$) (assume the walk down level s is 2).

V. Evaluation and Analysis

A. Evaluation Design

In order to assess the benefit of our coverage-driven testing framework, we conduct evaluations on both the path filtering algorithm and the PPS strategy. We choose two novel path search strategies from CREST [16] and KLEE [7] to compare against the PPS strategy. The CFG-Directed Search strategy in CREST achieves the highest branch coverage level against other heuristics in its evaluation. The default RP-MD2U Search strategy in KLEE combines both random path selection (RP) and coverage optimized search (MD2U). It aims to attack path explosion by advantages from both RP and MD2U.

- CFG-Directed Search: The CFG-Directed search strategy attempts to drive the execution down the branch with the minimal distance towards uncovered goals measured by static CFG paths. It also adopts a branch heuristic to backtrack or restart the current search under some failing circumstances. It is essentially a local optimal search, since it focuses on each recently-covered branch and never explicitly revisits previous paths.
- RP-MD2U Search: It interleaves the Random Path strategy with Min-Dist-to-Uncovered heuristic. The Random Path strategy is actually a probabilistic version of breadth-first search, which weighs a path candidate of length l by 2^{-l} and randomly chooses candidates with the same length. The Min-Dist-to-Uncovered heuristic prefers the path candidates with minimal distance to uncovered goals in CFG.

We built the framework on our DSE-based engine [15] and carried out the empirical experiments on three open source projects from SIR [8] and two industrial safety-critical projects from our research partners. Three path exploration strategies, i.e., the traditional depth-first search (DFS), the CFG-Directed search from CREST (CAUT-CREST) and the RP-MD2U search from KLEE (CAUT-KLEE) are implemented. These three strategies are representatives among the-state-of-art symbolic executors [5]–[8], [16]. Choosing these three typical strategies helps us to assess our framework on a fair basis. All evaluations were run on a 2.67 GHz Intel Xeon(R) X5650 server with 2GB of RAM and running Ubuntu GNU/Linux 12.04.

In our evaluation, we intend to address the following research questions:

- **RQ1**: Based on the traditional DSE-based approach (DFS-based search strategy), how much does the efficiency increase with the help of our coverage-driven testing framework (with path filtering)? We give the evaluation results w.r.t both the branch and MC/DC criteria.
- **RQ2**: Within the *same* constrained testing budgets, to what extend does the CS-based predictive path search strategy (CAUT-PPS) improve the code coverage compared to other symbolic execution based search strategies, i.e., CAUT-CREST and CAUT-KLEE in our coverage-driven testing framework (without path filtering)?

B. Implementation

CAUT [15] is a unit-testing prototype tool on C program. The program under test is instrumented by CIL [9], which is a C analysis and transformation infrastructure. During the execution, the symbolic engine dynamically maintains a logical memory map and an execution tree. The map records the mapping between symbolic and concrete values of variables. The execution tree stores each execution path. CAUT uses lp_solve[10], an open source library for linear programming, to solve path constraints. The tool supports constraint reasoning on both scalar types and derived types, i.e., structures, arrays and pointers (equality/inequality constraints). It currently provides statement, branch and MC/DC coverage-driven testing.

C. Subject Programs

We choose the units from three open source projects and two industrial projects developed by our research partners to evaluate our approach. We adopt the cyclomatic complexity [23], which is a metric to measure code complexity, to choose units or modules with *enough* code complexity. We use the tool *cyclo*[11] to calculate the complexity value of a

[8]SIR: http://sir.unl.edu/php/previewfiles.php
[9]CIL: http://kerneis.github.io/cil/
[10]lp_solve: http://sourceforge.net/projects/lpsolve/
[11]cyclo: http://bima.astro.umd.edu/nemo/man_html/cyclo.l.html

TABLE I
BRANCH COVERAGE TESTING

Project	#BR	#ITER (dfs/cs-dfs)	#CON (dfs/cs-dfs)	#COV (dfs/cs-dfs)
bash	312	907/643	6981/4663	83.3%/83.3%
flex	248	837/596	7367/4712	85.1%/**86.7**%
make	436	1043/849	9518/5734	83.6%/**85.8**%
osek_os	276	737/534	6458/4409	80.4%/80.4%
space_control	424	1112/737	9692/6964	82.3%/**84.4**%

TABLE II
MC/DC COVERAGE TESTING

Project	#DC	#ITER (dfs/cs-dfs)	#CON (dfs/cs-dfs)	#COV (dfs/cs-dfs)
bash	156	1231/984	9981/7663	80.2%/80.2%
flex	124	1029/733	8460/5973	70.3%/**76.6**%
make	218	1235/994	12518/8734	76.9%/**77.9**%
osek_os	138	978/654	9827/6397	76.6%/76.6%
space_control	212	1373/943	11739/8436	80.4%/**83.1**%

unit. For three open source projects from SIR (*bash*, *make*, and *flex*), we choose 69 units whose cyclomatic complexities all exceed 8[12] with total size 14K LOC. One industrial project is a commercial automotive operating system (*osek_os*) conforming to the OSEK/VDX [24] standard. It contains 51 units (with complexity value 7.5 in average) with about 10k LOC. It runs on electronic control units with static priority schedule mechanism in automobiles. The other one is a control system software (*space_control*) of some satellites from China Academy of Space Technology (CAST). It contains 70 units (with complexity value 9.4 in average) with roughly 15K LOC. These two industrial projects feature complicate execution logic. We choose benchmarks from different software projects, as we intend to make our evaluation results as convincing as possible.

D. Result and Analysis

RQ1: Efficiency and Effectiveness. In this evaluation, we use the number of program iterations and solved constraints as measurements on the performance of coverage testing. Note one program iteration indicates one feasible path candidate is exercised and one test case is generated. For a DSE-based approach, program execution and constraint solving count for most testing costs. So, it is reasonable to measure the performance by these two factors. We limit the maximum testing budget (time cost) as 20 minutes for each project.

Tables I and II show the detailed statistics of the traditional DSE-based approach (DFS-based search strategy) without/with the help of the coverage-driven testing framework (with path filtering). They, respectively, show the evaluation results w.r.t branch coverage and MC/DC coverage testing on these five projects. The second columns list the corresponding total branches (BR) or decisions (DC). The third and fourth column represent the total number of program iterations (ITER) and solved constraints (CON). The last column gives the average coverage percentage (COV). In the last three columns, we give the result without/with the help of the coverage-driven testing framework (dfs/cs-dfs).

From these two tables, we can see our approach greatly reduced the iterations of program execution (as well as generated test cases). The solved constraints are also greatly reduced. In Table I, take the project *flex* as an example, compared with the traditional DSE-based approach, our approach reduced roughly 28% program iterations and 36% solved constraints w.r.t branch coverage testing. For MC/DC coverage, the situation is similar. From Table II, nearly 31% iterations and 28% constraints are reduced for *space_control*. And in two tables the coverage levels on some projects are a little higher. The reason is that we encountered 10 units (nearly 5% in total) featuring complicate loop structure. They ran without termination within the maximum allowed time (so it failed to cover some branches or decisions). The DFS strategy alone is easy to be trapped in infinite loop unfolding and wastes time in exploring unfruitful path candidates. But with the benefit from path filtering, many irrelevant paths generated by loop unfolding are filtered away.

Although we only compare the evaluation results between the classic DFS strategy without/with path filtering, this path filtering algorithm is also effective and adaptable to other path exploration strategies, like CAUT-KLEE and CAUT-CREST. Because these search strategies may also exercise irrelevant path candidates although they are less likely to be trapped during execution.

RQ2: Heuristic Strategies To evaluate the effectiveness of the PPS strategy, we run the coverage-driven testing under three path exploration strategies, i.e., CAUT-PPS, CAUT-CREST and CAUT-KLEE on the constrained testing budget in terms of program iterations. We set the maximum iterations of one UUT as 100 in case it does not terminate. All UUTs are tested in turn (start testing the next unit after finishing one) under three different strategies. In addition, because three heuristics inherently contain randomness, we repeat the testing process 30 times for each heuristic and take the average value as the final result. In the PPS strategy, we observed different walk down levels have different impacts in preliminary evaluations. The overall optimal performance could be achieved when s equals 3. So we set the walk down level as 3 in this evaluation (We omit the experiment on this observation because of space limitations). Some readers may wonder it will be more convincing to compare the performance between the proposed design with and without the PPS strategy. But in a DSE framework, we have to adopt one path search strategy to select paths. So it is impossible to conduct testing without the PPS strategy.

Figure 4 shows the results of three heuristic strategies on branch coverage while Figure 5 shows the results on MC/DC coverage. For example, the left two sub-figures (a) and (b) of Figure 4 shows the results on two open source projects ($bash$ and $flex$) while the right two sub-figures (c) and (d) shows the results on two industrial projects ($osek\_os$ and $space\_control$). The horizontal axis denotes the number of

[12]In software practice, when the cyclomatic complexity value of a unit or module exceed 10, this module is regarded as too complex and should be split. So we choose units with complexity value above 8.

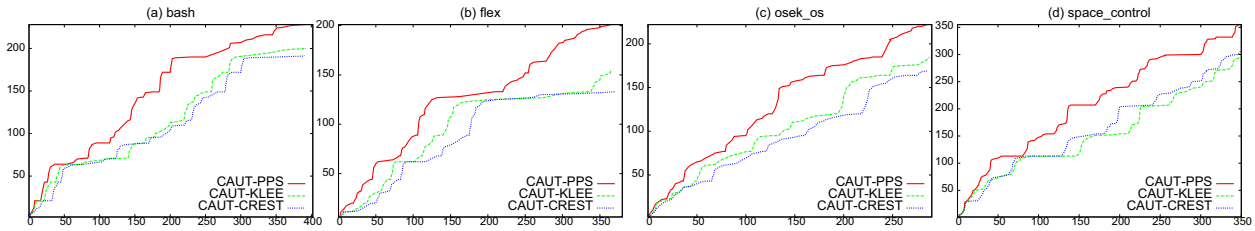


Fig. 4. Branch Coverage Testing, (a) **bash** (b) **flex** (c) **osek_os** (d) **space_control**, X-axis: iterations, Y-axis: covered branches

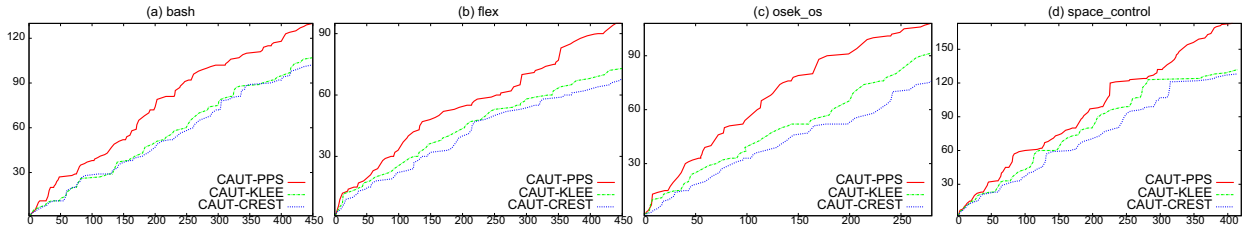


Fig. 5. MC/DC Coverage Testing, (a) **bash** (b) **flex** (c) **osek_os** (d) **space_control**, X-axis: iterations, Y-axis: covered decisions

program iterations and the vertical axis represents the number of covered branches.

From Figures 4 and 5, we can see that CAUT-PPS achieves higher coverage level than CAUT-CREST and CAUT-KLEE within the same constrained program iterations, which means CAUT-PPS performs more faster coverage-driven testing. We can notice the curves of CAUT-PPS are more *steep* than those of the other two strategies. The reason is that among all available relevant candidates, CAUT-PPS always eagerly explores the path candidates with highest predictive contribution on code coverage. But CAUT-KLEE or CAUT-CREST decides the exploration direction only on path length or distance metric. In detail, for open source projects and industrial projects, CAUT-PPS respectively improves about 26.4% and 23.2% in average than CAUT-KLEE w.r.t branch coverage. Compared with CAUT-CREST, it respectively improves about 35.4% and 22.9%. The trend is similar on MC/DC coverage.

In addition, in the most sub-figures of Figure 4, we can observe that there exist such situations (some horizontal parts of curves) where three strategies all *spin* at the same horizontal level, i.e., they continue to iterate without reaching new goals. But the spin time of CAUT-PPS is much shorter than CAUT-KLEE and CAUT-CREST. The reason behind this phenomenon is that when CAUT-PPS is exercising may-relevant candidates (they are eventually unfruitful without hitting new coverage goals), once must-relevant candidates emerge, it will immediately jump to exercise must-relevant ones (they are probably fruitful). In other words, CAUT-PPS is aware of the coverage information at runtime maintained by CS and capable of dynamically adjusting its exploration direction. However, CAUT-KLEE and CAUT-CREST are not directly driven by this information and may waste more time on continuing exploring eventually unfruitful path candidates.

However, we find CAUT-PPS is not always effective under some situations. Careful readers may notice another situation (for example, in the sub-graph (d) of Figure 4, for *space_control*, when the number of covered branches reach 300, observing from the same horizontal level, CAUT-PPS spins a longer time than CAUT-KLEE or/and CAUT-CREST). In other words, CAUT-PPS requires more iterations than the either one to reach a new goal. This phenomenon happens when the CS-based contribution prediction sometimes is imprecise. The following simplified code fragment from *space_control* demonstrates this situation.

```
1 void loop(int v[6],int x){
2   int i;
3   if(x==2){
4     for(i=0;i<6;i++)
5       if(v[i]==3) x++;
6     if(x==6) {...}
7   }
8 }
```

Under branch coverage testing, it is easy for CAUT-PPS to take three iterations to cover all branches except the $6T$ branch in the above code (assume all elements in array v are initially not equal to 3 and x is initially not equal to 2). Based on the feedback from CS, CAUT-PPS will next eagerly to exercise the candidate generated by flipping the branching node for the false branch of Line 6 to the true branch. Because it is a must-relevant one (the $6T$ branch is uncovered). But it is actually infeasible (In the third iteration, we flip one branching node for the false of Line 5 to the true branch in order to cover the branch $5T$. So only one element in v is equal to 3 and x can not be equal to 6). So CAUT-PPS degrades to exercise the remaining may-relevant candidates spawned by flipping the branching node $5T$ or $5F$ to its opposite (such

candidates continuously grow because of the loop unfolding) . CAUT-PPS treats them equally (their $PotentialContr$ values are all equal to 1) and randomly chooses them. However, through code inspection, we can see that only when at least four elements in v are equal to 3 will the $6T$ branch be covered. Eventually attempting many times, it hits the $6T$ branch by chance. But for CAUT-KLEE, it interleaves the RPS and MD2U in which the RPS is more focused than CAUT-PPS to exercise the shortest candidates by exercising those *critical* unexplored branching nodes (like $5T$) to satisfy the underlying constraint. In this example, it takes only 39% iterations than that of CAUT-PPS to hit $6T$ in CAUT.

This situation can be alleviated by combining the fitness-guided path exploration technique [25]. This technique attacks the problem, i.e., only by satisfying some indirect relationship between the test inputs and some program condition (like *x==6* in the above example) can we hit some goals. Thus the PPS and the fitness-guided strategy can be regarded as *orthogonal* methods. The former attempts to hit coverage goals as much as possible at one blow while the latter tries to cover one hard-to-hit goal at one time. We have integrated the fitness-guided strategy into our tool CAUT. By interleaving the PPS and the fitness strategy, we found the performance could be further improved by 5%-8% in average in terms of iterations. And in the above example, this interleaved strategy only takes 30% iterations than that of CAUT-PPS.

E. Discussion and Threat to validity

In our evaluation, we have not directly built our framework on KLEE or CREST. The main reasons are following: (1) CREST does not support real numbers, composite structures (*struct* or *union*) and symbolic pointer reasoning. But these features are required in testing real world programs. (2) KLEE focuses on line coverage and it measures branch coverage based on LLVM[13] bitcode instead of original source code. It it is not easy to measure code coverage w.r.t logical criterion on LLVM bitcode. Although it is possible to replay test cases from KLEE on CAUT, it is still difficult to profile execution performance among different symbolic executors. So we implemented all search strategies on top of CAUT.

Some threats exists in the validity of our evaluation. First, we implemented the search strategies on our CIL-based tool CAUT. The original RP-MD2U strategy in KLEE uses the number of LLVM instructions to measure the minimal distance between one coverage goal to another while CAUT uses CIL statements or instructions as its distance metric. KLEE is a static symbolic executor while CAUT is a dynamic symbolic executor. These differences may affect the performance of the RP-MD2U strategy on our benchmarks. Second, we re-implement the two search strategies from CREST and KLEE on our tool. The implementation may differ from their original versions. But we carefully inspected their source codes and technical reports to ensure our implementation correctness. Third, our benchmarks are much smaller than that of KLEE. The strategies from CREST and KLEE work on the whole program while we focus on unit (program moudles) testing according to our research motivation. But we have chosen units with enough code complexity from different projects as benchmarks to ensure the evaluation result as convincing as possible.

[13]LLVM: http://llvm.org/

VI. Related Work

Coverage-driven Testing. A lot of research works have been conducted in the field of automating test data generation to achieve high coverage testing [26]. Dynamic symbolic execution [27] is one of promising techniques to automate this testing process. A number of state-of-art DSE-based tools [5]–[7] achieve statement or branch coverage by path exploration. Random testing [28] has also been adopted to achieve MC/DC criterion. However, based on the unified *coverage structure*, our approach is able to directly support both structural and logical coverage criteria without code transformations [12]. Furthermore, this coverage-driven testing framework is easy to be implemented on existing DSE-based tools regardless of their inherent path exploration strategies. In the field of search-based techniques [29], researchers have adopted different evolutionary algorithms [30], [31] to identify test data by exploring the input space of the program. This technique is very effective in enforcing high branch coverage. In order to support other criteria like MC/DC, they tailored fitness functions in the search process to generate specific test cases. However, this approach may incur local maximum [12], i.e., the input value is usually optimal within a neighboring set of solutions. The formal-verification based technique [32] has also been applied to generate high coverage test data combined with formal specification languages. This proof-based approach statically identifies test data by reasoning about all possible runs of a program. In contrast, our DSE-based framework can precisely handle heap operations and pointer aliases. Augmented DSE [13] is also proposed to achieve criteria such as boundary and logical criteria from the regression testing perspective. But it is realized by augmenting path conditions with additional conditions which is different from our approach.

Search Space Reduction. The DSE-based approach inherently suffers from the problem of path search space explosion. The RWset [33] and path subsumption methods [34] reduce search space when the current state is considered similar to a previously visited state. This approach prunes path exploration from the execution history version. But our path filtering algorithm judges the relevance of path candidates from its future reachability.

Path Search Strategies. Earlier DSE-based tools [5], [6] adopt the depth-first search. SAGE [9] uses a *generational* algorithm for path selection. PEX [8] is designed for C# with a bundle of heuristic search strategies with a bias towards shorter path candidates. These search strategies usually target one coverage goal at one time by different measurements. In contrast, the PPS strategy favors the candidate with the ability to hit coverage goals as much as possible at one iteration.

The CFG-Directed search strategy [16] is also based on static measurement from CFG. But it attempts to drive the execution down the branch with the minimal distance towards uncovered goals while PPS drives the execution towards areas with more uncovered goals by a local prediction. PPS prioritizes path candidates by their potential contributions on the code coverage w.r.t. some criterion. Fraser et al. [35] adopted a genetic algorithm to consider multiple coverage goals simultaneously instead of one after another with the expectation to reduce the test suite as small as possible. This approach starts from an initial population of test cases. But the PPS strategy considers multiple coverage goals by predicting a candidate's coverage contribution without initial test suites.

VII. Conclusions and Further Work

In this paper a general DSE-based coverage-driven testing framework is proposed. It supports both structural and logical coverage criteria through the unified *coverage structure*. It is easy to be implemented on existing DSE-based tools. A CS-based path filtering algorithm and a CS-based path exploration strategy are proposed to achieve faster coverage-driven testing with lower testing cost. The empirical experiment on C programs shows they are effective. There are also some interesting avenues for future work. First, we would like to investigate the performance of the path filtering algorithm under different path search strategies. And we will conduct more experiments with different walk down levels in the PPS strategy. Second, we would like to investigate how to interleave different search strategies by combining their respective advantages more effectively but without affecting the original code coverage.

Acknowledgment

Ting Su is partly supported by ECNU Project for Funding of Overseas Short-term Studies, Domestic Academic Visit and International Conference and NSFC Project No. 91118007. Geguang Pu is partially supported by Shanghai Knowledge Service Platform No. ZF1213 and NSFC Project No. 61361136002. Fang Bin is partially supported by SHEITC Project 130407. Jifeng He is partially supported by NSFC Project No. 61021004.

References

[1] D. L. Bird and C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[2] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence (Program Testing)," *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, Dec. 1990.

[3] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, Jan. 1996.

[4] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *ICSE*, 2007, pp. 416–426.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI*. New York, NY, USA: ACM, 2005, pp. 213–223.

[6] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *ESEC/SIGSOFT FSE*. New York, NY, USA: ACM, 2005, pp. 263–272.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems," in *OSDI*, 2008, pp. 209–224.

[8] N. Tillmann and J. de Halleux, "Pex-White Box Test Generation for .NET," in *TAP*, 2008, pp. 134–153.

[9] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[10] P. Ammann, A. J. Offutt, and H. Huang, "Coverage Criteria for Logical Expressions," in *ISSRE*, 2003, pp. 99–107.

[11] R. Inc, "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," *Requirements and Technical Concepts for Aviation*, December 1992.

[12] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided Test Generation for Coverage Criteria," in *ICSM*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[13] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux, "Augmented Dynamic Symbolic Execution," in *ASE*, 2012, pp. 254–257.

[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[15] Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding, and J. Hu, "Test Data Generation for Derived Types in C Program," in *TASE*, 2009, pp. 155–162.

[16] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *ASE*, 2008, pp. 443–446.

[17] B. Korel, "Automated Software Test Data Generation," *TSE*, vol. 16, no. 8, pp. 870–879, 1990.

[18] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided Path Exploration for Regression Test Generation," in *ISSTA*. New York, NY, USA: ACM, 2011, pp. 1–11.

[19] R. E. Prather and J. P. M. Jr., "The Path Prefix Software Testing Strategy," *TSE*, vol. 13, no. 7, pp. 761–766, 1987.

[20] S. Bardin and P. Herrmann, "Pruning the Search Space in Path-Based Test Generation," in *ICST*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 240–249.

[21] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem," in *ICST Workshops*, 2010, pp. 182–191.

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[23] T. J. McCabe, "A Complexity Measure," in *ICSE*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–.

[24] "OSEK/VDX, operating system specification 2.2.3, 17th ed. the osek group, compiled by: Jochem spohr mbtech, february 2005."

[25] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-Guided Path Exploration in Dynamic Symbolic Execution." in *DSN*. IEEE, 2009, pp. 359–368.

[26] K. Lakhotia, P. McMinn, and M. Harman, "Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?" in *Testing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 95–104.

[27] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in PracticePreliminary Assessment," in *ICSE*. New York, NY, USA: ACM, 2011, pp. 1066–1071.

[28] M. Staats, G. Gay, M. W. Whalen, and M. P. E. Heimdahl, "On the Danger of Coverage Directed Test Case Generation," in *FASE*, 2012, pp. 409–424.

[29] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[30] Z. Awedikian, K. Ayari, and G. Antoniol, "MC/DC Automatic Test Input Data Generation," in *GECC*, 2009, pp. 1657–1664.

[31] K. Ghani and J. A. Clark, "Automatic Test Data Generation for Multiple Condition and MCDC Coverage," in *ICSEA*, 2009, pp. 152–157.

[32] W. Ahrendt, W. Mostowski, and G. Paganelli, "Real-time Java API Specifications for High Coverage Test Generation," in *JTRES*. New York, NY, USA: ACM, 2012, pp. 145–154.

[33] P. Boonstoppel, C. Cadar, and D. R. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *TACAS*, 2008, pp. 351–366.

[34] S. Anand, C. S. Pasareanu, and W. Visser, "Symbolic Execution with Abstract Subsumption Checking," in *SPIN*, 2006, pp. 163–181.

[35] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *TSE*, vol. 39, no. 2, pp. 276–291, 2013.