Towards Efficient Data-Flow Test Data Generation

Ting Su[1(✉)], Chengyu Zhang[2], Yichen Yan[1], Lingling Fan[3], Yang Liu[4], Zhoulai Fu[5], and Zhendong Su[2]

[1] East China Normal University, Shanghai, China
tsu@sei.ecnu.edu.cn
[2] ETH Zurich, Zürich, Switzerland
chengyu.zhang@inf.ethz.ch , zhendong.su@inf.ethz.ch
[3] Nankai University, Tianjin, China
linglingfan@nankai.edu.cn
[4] Nanyang Technological University, Singapore, Singapore
yangliu@ntu.edu.sg
[5] State University of New York, Incheon, Korea
zhoulai.fu@sunykorea.ac.kr

Abstract. Data-flow testing (DFT) aims to detect potential data interaction anomalies by focusing on the points at which variables receive values and the points at which these values are used. Such test objectives are referred as *def-use pairs*. However, the complexity of DFT still overwhelms the testers in practice. To tackle this problem, we introduce a hybrid testing framework for data-flow based test generation: (1) The core of our framework is symbolic execution (SE), enhanced by a novel guided path exploration strategy to improve testing performance; and (2) we systematically cast DFT as reachability checking in software model checking (SMC) to complement SE, yielding practical DFT that combines the two techniques' strengths. We implemented our framework for C programs on top of the state-of-the-art symbolic execution engine KLEE and instantiated with three different software model checkers. Our evaluation on the 28,354 def-use pairs collected from 33 open-source and industrial program subjects shows that (1) our SE-based approach can improve DFT performance by 15–48% in terms of testing time, compared with existing search strategies; and (2) our combined approach can further reduce testing time by 20.1–93.6%, and improve data-flow coverage by 27.8–45.2% by eliminating infeasible test objectives. This combined approach also enables the cross-checking of each component for reliable and robust testing results.

Keywords: Data-flow Testing · Symbolic Execution · Model Checking

1 Introduction

It is widely recognized that white-box testing, usually applied at unit testing level, is one of the most important activities to ensure software quality [4]. In

This paper was completed on Jifeng He's 80th birthday, May 2023.

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 257–293, 2023.
https://doi.org/10.1007/978-3-031-40436-8_10

this process, the testers design inputs to exercise program paths in the code, and validate the outputs with specifications [50]. Code coverage criteria are popular metrics to guide such test selection. For example, control-flow based criteria (*e.g.*, statement, branch coverage) require to cover the specified program elements, *e.g.*, statements, branches and conditions, at least once [100]. In contrast, data-flow based criteria [27,46,76] focus on the flow of data, and aim to detect potential data interaction anomalies. It validates the correctness of variable definitions by observing the values at the corresponding uses.

However, several challenges exist in generating data-flow based test cases: (1) *Few data-flow coverage tools exist.* To our knowledge, `ATAC` [52,53] is the only publicly available tool, developed two decades ago, to measure data-flow coverage for `C` programs. However, there are plenty of tools for control-flow criteria [96]. (2) *The complexity of identifying data flow-based test data overwhelms testers.* Test objectives *w.r.t.* data-flow testing are much more than those of control-flow criteria; more effort is required to satisfy a def-use pair than just covering a statement or branch, since the test case needs to reach a variable definition first and then the corresponding use. (3) *Infeasible test objectives* (*i.e.*, the paths from the variable definition to the use are infeasible) and *variable aliases* make data-flow testing more difficult.

To aid data-flow testing, many testing techniques have been proposed in the past few decades. For example, search-based approach [29,39,41,89] uses genetic algorithms to guide test generation to cover the target def-use pairs. It generates an initial population of test cases, and iteratively applies mutation and crossover operations on them to optimize the designated fitness function. Random testing [41,42] generates random test inputs or random paths to cover def-use pairs. Some work uses the idea of collateral coverage [62,77], *i.e.*, the relation between data-flow criteria and the other criteria (*e.g.*, branch coverage), to infer data-flow based test cases. However, these approaches are either *inefficient* (*e.g.*, random testing may generate a large number of redundant test cases) or *imprecise* (*e.g.*, genetic algorithms and collateral coverage-based approach may not be able to identify infeasible test objectives).

The preceding situations underline the importance of an automated, effective data-flow testing technique, which can efficiently generate test cases for target def-use pairs and detect infeasible ones therein. To this end, we introduce *a combined approach* to automatically generate data-flow based test data, which synergistically combines two techniques: dynamic symbolic execution and counterexample-guided abstraction refinement-based model checking. It takes as input the program under test, and (1) *outputs test cases for feasible test objectives*, and (2) *eliminates infeasible test objectives—without any false positives.*

Dynamic symbolic execution [20] is a widely-accepted and effective approach for automatic test case generation. It intertwines classic symbolic execution [26,58] and concrete execution, and explores as many program paths as possible to generate test cases by solving path constraints. As for counterexample-guided abstraction refinement-based (CEGAR) model checking [5,22,49], given the program source code and a temporal safety specification, it either statically

proves that the program satisfies the specification, or returns a counterexample path to demonstrate its violation. This technique has been used to automatically verify safety properties of device drives [5,13,14], as well as test generation *w.r.t.* statement or branch coverage [12] from counterexample paths.

Although symbolic execution has been applied to enforce various coverage criteria (*e.g.*, statement, branch, logical, boundary value and mutation testing) [55,60,73,85,99], little effort exists to adapt symbolic execution to data-flow testing. To counter the path explosion problem, we designed a *cut-point guided path exploration strategy* to cover target def-use pairs as quickly as possible. The key intuition is to find a set of critical program locations that must be traversed through in order to cover the pair. By following these points during the exploration, we can narrow the path search space. In addition, with the help of path-based exploration, we can also more easily and precisely detect definitions due to variable aliasing. Moreover, we introduce a simple, powerful encoding of data flow testing using CEGAR-based model checking to complement our SE-based approach: (1) We show how to encode any data-flow test objective in the program under test and systematically evaluate the technique's practicality; and (2) we describe a combined approach that combines the relative strengths of the SE and CEGAR-based approaches. An interesting by-product of this combination is to let the two independent approaches cross-check each other's results for correctness and consistency.

In all, this paper makes the following contributions:

- We design a symbolic execution-based testing framework, and enhance it with an efficient guided path search strategy, to quickly achieve data-flow testing.
- We describe a simple, effective reduction of data-flow testing into reachability checking in software model checking to complement our SE-based approach.
- We implement the SE-based data-flow testing approach, and conduct empirical evaluation on both benchmark and industrial C programs. Our results show that the SE-based approach is both efficient and effective.
- We also demonstrate that the CEGAR-based approach can effectively complement the SE-based approach by reducing testing time and identifying infeasible test objectives. In addition, these two approaches can cross-check each other to validate the correctness and effectiveness of both techniques.

The initial idea of this hybrid data-flow testing approach was described in [84], and in this paper we have improved this idea in several aspects: (1) We optimized our original cut-point guided search with several exploration strategies (*e.g.*, backtrack), and made substantial efforts to implement our approach on the state-of-the-art symbolic execution engine KLEE [18] (previously implemented on our own concolic testing tool CAUT [84,85], which was capable of evaluating only 6 subjects), and further compared our approach with various existing testing strategies on KLEE. Due to the differences in the design and architecture between KLEE and CAUT, the implementation is not straightforward. But this effort brings several benefits: first, it provides a uniform and fair platform to investigate the effectiveness of our testing strategy with many existing state-of-the-art ones; second, it provides a robust platform to enable exten-

sive evaluation of real-world subjects and better integration with model checkers; third, this extension of KLEE could benefit industrial practitioners and also academic researchers to apply or investigate data-flow testing. (2) We implemented and extended the model checking-based approach on two different techniques, *i.e.*, Counter-Example Guided Abstraction Refinement (CEGAR) [5,21,49] and Bounded Model Checking (BMC) [25], and comprehensively compared their effectiveness and performance for data-flow testing; (3) We rigorously setup a benchmark repository for data-flow testing, and extensively evaluated on 30 real-world programs with various data-flow usage scenarios, including seven non-trivial subjects from previous DFT research work [32,35,39,48,54,66,67], seven subjects from SIR [82], 16 subjects from SV-COMP [33] so as to gain a overall understanding of our hybrid testing framework. (4) We cross-checked each component to provide reliable testing results, investigated the reasons of inconsistent cases, and gave detailed discussions.

The paper is organized as follows. Section 2 surveys the related work in data-flow testing. Section 3 gives more background and Sect. 4 gives an overview of our testing framework with an illustrative example. Section 5 details our approach. Section 6 explains the design and implementation. Section 7 presents the evaluation results. Section 8 concludes the paper.

2 Related Work

This section discusses the closely related work: (1) data-flow based test generation, (2) directed symbolic execution, and (3) infeasible test objective detection.

2.1 Data-Flow Based Test Generation

Data-flow testing has been investigated in the past four decades [34–36,54,93]. Existing work can be categorized into five main categories according to the testing techniques. We only discuss typical literature work here. Readers can refer to a recent survey [86] for details.

The most widely used approach to is *search-based testing*, which utilizes meta-heuristic search techniques to identify test inputs for target def-use pairs. Girgis [41] first uses Genetic Algorithms (GA) for Fortran programs, and Ghiduk *et al.* [39] use GA for C++ programs. Later, Vivanti *et al.* [89] and Denaro *et al.* [29] apply GA to `Java` programs by the tool EvoSuite. Some optimization-based search algorithms [38,69,80,81] are also used, but they have only evaluated on small programs without available tools. *Random testing* is a baseline approach for data-flow testing [3,29,39,41,42]. Some researchers use *collateral coverage-based testing* [45], which exploits the observation that the test case that satisfies one target test objective can also "accidentally" cover the others. Malevris *et al.* [62] use branch coverage to achieve data-flow coverage. Merlo *et al.* [68] exploit the coverage implication between data-flow coverage and statement coverage to achieve intra-procedural data-flow testing. Other efforts include [65,66,77,78]. Some researchers use *traditional symbolic execution*. For

example, Girgis [40] develops a simple symbolic execution system for DFT, which statically generates program paths *w.r.t.* a certain control-flow criterion (*e.g.*, branch coverage), and then selects those executable ones that can cover the def-use pairs of interest. Buy *et al.* [17] adopts three techniques, *i.e.*, data-flow analysis, symbolic execution and automated deduction to perform data-flow testing. However, they have provided little evidence of practice. Hong *et al.* [51] adopt *classic CTL-based model checking* to generate data-flow test data. Specifically, the program is modeled as a Kripke structure and the requirements of data-flow coverage are characterized as a set of CTL property formulas. However, this approach requires manual intervention, and its scalability is also unclear.

Despite the plenty of work on data-flow based testing, they are either inefficient or imprecise. Our work is the first one to leverage symbolic execution and software model checking techniques to achieve DFT efficiently and precisely.

2.2 Directed Symbolic Execution

Much research [31,61,64,95,97] has been done to guide path search toward a specified program location via symbolic execution. Do *et al.* [31] leverage data dependency analysis to guide the search to reach a particular program location, while we use dominator analysis. Ma *et al.* [61] suggest a call chain backward search heuristic to find a feasible path, backward from the target program location to the entry. However, it is difficult to adapt this approach on data-flow testing, because it requires that a function can be decomposed into logical parts when the target locations (*e.g.* the *def* and the *use*) are located in the same function. But decomposing a function itself is a nontrivial task. Zamfir *et al.* [97] narrow the path search space by following a *limited* set of critical edges and a statically-necessary combination of intermediate goals. On the other hand, our approach finds a set of cut points from the program entry to the target locations, which makes path exploration more efficient. Xie *et al.* [95] integrate fitness-guided path search strategy with other heuristics to reach a program point. The proposed strategy is only efficient for those problems amenable to its fitness functions. Marinescu *et al.* [64] use a shortest distance-based guided search method (like the adapted SDGS heuristic in our evaluation) with other heuristics to quickly reach the line of interest in patch testing. In contrast, we combine several search heuristics to guide the path exploration to traverse two specified program locations (*i.e.*, the *def* and *use*) for data flow testing.

2.3 Detecting Infeasible Test Objectives

As for detecting infeasible test objectives, early work uses constraint-based technique [44,71]. Offutt and Pan *et al.* [71] extract a set of path constraints that encode the test objectives from the program under test. Infeasible test objectives can be identified if the constraints do not have solutions. Recent work by Beckman *et al.* [10], Baluda *et al.* [6–8], Bardin *et al.* [9] use weakeast precondition to identify infeasible statements and branches. For example, Baluda *et al.* use model refinement with weakest precondition to exclude infeasible branches;

Bardin *et al.* applies weakest precondition with abstract interpretation to eliminate infeasible objectives. Marcozzi *et al.* [63] also use weakest precondition to identify polluting test objectives (including infeasible, duplicate and subsumed) for condition, MC/DC and weak mutation coverage. In contrast, our testing framework mainly use the CEGAR-based model checking technique to identify infeasible def-use pairs for data-flow testing. One close work is from Daca *et al.* [28], who combine concolic testing (CREST) and model checking (CPAchecker) to find a test suite *w.r.t.* branch coverage. Our work has some distinct differences with theirs. First, they target at branch coverage, while we enforce data-flow testing. Second, they directly modify the existing generic path search strategies of CREST, and backtrack the search if the explored direction has been proved as infeasible by CPAchecker. As a result, the performance of their approach (*i.e.*, avoid unnecessary path explorations) may vary across different search strategies due to the paths are selected in different orders. In contrast, we implement a designated search strategy to guide symbolic execution, and realize the reduction approach directly on model checkers. Although our approach is simple, it can treat model checkers as black-box tools without any modification and seamlessly integrate with KLEE. Model checking techniques have recently been adapted to aid software testing [15,37].

3 Problem Definition, Preliminaries and Challenges

3.1 Problem Definition

Definition 1 (Program Paths). *Two kinds of program paths, i.e., control flow paths and execution paths are distinguished during data-flow testing. Control flow paths are the paths from the control flow graph of the program under test, which abstract the flow of control. Execution paths are driven by concrete program inputs, which represent dynamic program executions. Both of them can be represented as a sequence of control points (denoted by line numbers), e.g.,* $l_1, \ldots, l_i, \ldots, l_n$.

Definition 2 (Def-use Pair). *The test objective of data-flow testing is referred as a def-use pair, denoted by* $du(l_d, l_u, v)$. *Such a pair appears when there exists a control flow path that starts from the variable definition statement* l_d *(or the def statement in short), and then reaches the variable use statement* l_u *(or the use statement in short), but no statements on the subpaths from* l_d *to* l_u *redefine the variable* v.

In particular, two kinds of def-use pairs are distinguished. For a def-use pair (l_d, l_u, v), if the variable v is used in a computation statement at l_u, the pair is a *computation-use* (*c-use* for short), denoted by $dcu(l_d, l_u, v)$. If v is used in a conditional statement (*e.g.*, an `if` or `while` statement) at l_u, the pair is a *predicate use* (*p-use* for short). At this time, two def-use pairs appear and can be denoted by $dpu(l_d, (l_u, l_t), v)$ and $dpu(l_d, (l_u, l_f), v)$, where (l_u, l_t) and (l_u, l_f) represents the *true* and the *false* edge of the conditional statement, respectively.

Definition 3 (Data-flow Testing). *Given a def-use pair $du(l_d, l_u, v)$ in program P under test, the goal of data-flow testing*[1] *is to find an input t that induces an execution path p that covers the variable definition statement at l_d, and then covers variable use statement at l_u, but without covering any redefinition statements w.r.t v, i.e., the subpath from l_d to l_u is a def-clear path. The requirement to cover all def-use pairs at least once is called all def-use coverage criterion*[2] *in data-flow testing.*

In particular, for a c-use pair, t should cover l_d and l_u; for a p-use pair, t should cover l_d and its true or false branch, *i.e.*, (l_u, l_t) and (l_u, l_f), respectively.

3.2 Symbolic Execution

Our data-flow testing approach is mainly built on the symbolic execution technique. The idea of symbolic execution (SE) was initially described in [26,58]. Recent significant advances in the constraint solving techniques have made SE possible for testing real-world program by systematically exploring program paths [20]. Specifically, two variants of modern SE techniques exist, *i.e.*, *concolic testing* (implemented by DART [43], CUTE [79], CREST [16], CAUT [85], *etc*) and *execution-generated testing* (implemented by EXE [19] and KLEE [18]), which mix concrete and symbolic execution together to improve scalability. In essence, SE uses *symbolic values* in place of *concrete values* to represent input variables, and represent other program variables by the *symbolic expressions* in terms of symbolic inputs. Typically, SE maintains a *symbolic state* σ, which maps variables to (1) the symbolic expressions over program variables, and (2) a symbolic path constraint pc (a quantifier-free first order formula in terms of input variables), which characterizes the set of input values that can execute a specific program execution path p. Additionally, σ maintains a program counter that refers to the current instruction for execution. At the beginning, σ is initialized as an empty map and pc as *true*. During execution, SE updates σ when an assignment statement is executed; and forks σ when a conditional statement (*e.g.*, **if**(e) s_1 **else** s_2) is executed. Specifically, SE will create a new state σ' from the original state σ, and updates the path constrain of σ' as $pc \wedge \neg(e)$, while updates that of σ as $pc \wedge (e)$. σ and σ', respectively, represent the two program states that fork at the *true* and *false* branch of the conditional statement. By querying the satisfiability of updated path constraints, SE decides which one to continue the exploration. When an *exit* or certain runtime error is encountered, SE will terminate on that statement and the concrete input values will be generated by solving the corresponding path constraint.

[1] In this paper, we focus on the problem of *classic data-flow testing* [39,89], *i.e.*, finding an input for a given def-use pair at one time. We do not consider the case where some pairs can be accidentally covered when targeting one pair, since this has already been investigated in collateral coverage-based approach [65,66].

[2] We follow the all def-use coverage defined by Rapps and Weyuker [75,76], since almost all of the literature that followed uses or extends this definition, as revealed by a recent survey [86].

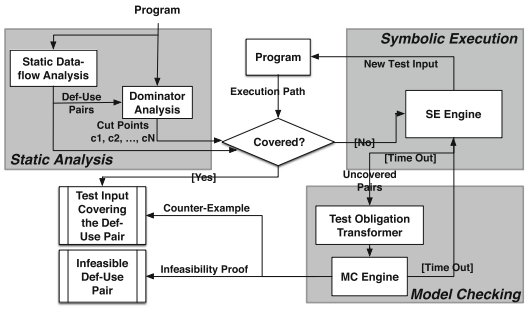


Fig. 1. Workflow of the combined approach for data-flow testing, which combines symbolic execution and software model checking (the CEGAR-based model checking in particular).

Challenges. Although SE is an effective test case generation technique for traditional coverage criteria, it faces two challenges in our context:

1. The SE-based approach by nature faces the notorious path-explosion problem. Despite the existence of many generic search strategies, it is challenging, in reasonable time, to find an execution path from the whole path space to cover a given pair.
2. The test objectives from data-flow testing include *feasible* and *infeasible* pairs. A pair is *feasible* if there exists an execution path which can pass through it. Otherwise it is *infeasible*. Without prior knowledge about whether a target pair is feasible or not, the SE-based approach may spend a large amount of time, in vain, to cover an infeasible def-use pair.

Section 4 will give an overview of our approach, and illustrate how our combined approach tackles these two challenges via an example in Fig. 2.

4 Approach Overview

Figure 1 shows the workflow of our combined approach for data-flow testing. It takes as input the program source code, and follows the three steps below to achieve automated, efficient DFT. (1) The *static analysis* module uses data-flow analysis to identify def-use pairs, and adopts dominator analysis to analyze the sequence of cut points for each pair (see Sect. 5.1). (2) For each pair, the *symbolic execution* module adopts the cut point-guided search strategy to efficiently find an execution path that could cover it within a specified time bound (see Sect. 5.2). (3) For the remaining uncovered (possibly infeasible) pairs, the *software model checking* module encodes the test obligation of each def-use pair into the program under test, and enforces reachability checking (also within a time bound) on each of them. The model checker can eliminate infeasible ones with proofs and may also identify feasible ones (see Sect. 5.3). If the testing resource permits, the framework can iterate between (2) and (3) by lifting the time bound

Table 1. Running steps of the enhanced symbolic execution approach for data-flow testing.

Steps	Pending Path	Priority Queue	Selected Path	Path Constraint (pc)
1	**1**: l_{4T}, **2**: l_{4F}	$(l_4, 2)^{\mathbf{1}}$, $(l_4, 2)^{\mathbf{2}}$	**1**	$y > 0$
2	**2**: l_{4F}, **3**: l_{4T}, l_{9T}, **4**: l_{4T}, l_{9F}	$(l_9, 1)^{\mathbf{4}}$, $(l_4, 2)^{\mathbf{2}}$, $(l_9, 4)^{\mathbf{3}}$	**4**	$y > 0 \wedge y == 0$
3	**2**: l_{4F}, **3**: l_{4T}, l_{9T}	$(l_4, 2)^{\mathbf{2}}$, $(l_9, 4)^{\mathbf{3}}$	**3**	$y > 0 \wedge y \neq 0$
4	**2**: l_{4F}, **5**: l_{4T}, l_{9T}, l_{9T}, **6**: l_{4T}, l_{9T}, l_{9F}	$(l_4, 2)^{\mathbf{2}}$, $(l_9, 1)^{\mathbf{5}}$, $(l_9, 1)^{\mathbf{6}}$	prune **5,6**, select **2**	$y \leq 0$
5	**7**: l_{4F}, l_{9T}, **8**: l_{4F}, l_{9F}	$(l_9, 4)^{\mathbf{7}}$, $(l_9, 1)^{\mathbf{8}}$	**8**	$y < 0 \wedge y \neq 0$
6	**7**: l_{4F}, l_{9T}, **9**: l_{4F}, l_{9F}, l_{13T}, **10**: l_{4F}, l_{9F}, l_{13F}	$(l_{13}, 1)^{\mathbf{9}}$, $(l_9, 4)^{\mathbf{7}}$, $(l_{13}, \infty)^{\mathbf{10}}$	**9**	$y \leq 0 \wedge y \neq 0$
7	**7**: l_{4F}, l_{9T}, **10**: l_{4F}, l_{9F}, l_{13F} **11**: $l_{4F}, l_{9F}, l_{13T}, l_{14T}$, **12**: $l_{4F}, l_{9F}, l_{13T}, l_{14F}$	$(l_{13}, 1)^{\mathbf{12}}$, $(l_9, 4)^{\mathbf{7}}$, $(l_{13}, \infty)^{\mathbf{10}}$, $(l_{14}, \infty)^{\mathbf{11}}$	**12**	$y == 0 \wedge x \neq 0$

to continue test those remaining uncovered pairs. By this way, our framework outputs test cases for feasible test objectives, and weeds out infeasible ones by proofs—without any false positives.

4.1 Illustrative Example

Figure 2 shows an example program *power*, which accepts two integers x and y, and outputs the result of x^y. The right sub-figure shows the control flow graph of *power*.

Step 1: Static Analysis. For the variable res (it stores the computation result of x^y), the static analysis procedure can find two typical def-use pairs with their cut points:

$$du_1 = (l_8, l_{17}, res) \tag{1}$$

$$du_2 = (l_8, l_{18}, res) \tag{2}$$

Below, we illustrate how our combined approach can efficiently achieve DFT on these two def-use pairs—SE can efficiently cover the feasible pair du_1, and CEGAR can effectively conclude the infeasibility of du_2.

Step 2: SE-Based Data-Flow Testing. When SE is used to cover du_1, assume under the classic depth-first search (DFS) strategy [16,18,43,79,85,88] the *true* branches of the new execution states (ESs) are always first selected, we can get an execution path p after unfolding the `while` loops n times.

$$p = l_4, l_5, l_8, \underbrace{l_9, l_{10}, l_{11},\ l_9, l_{10}, l_{11}, \ldots,}_{\text{repeated } n \text{ times}} l_9, l_{13}, l_{14}, l_{15} \tag{3}$$

Here p already covers the definition statement (at l_8) *w.r.t.* the variable res. In order to cover the use statement (at l_{17}), SE will exhaustively execute program paths by exploring the remaining unexecuted branch directions. However, the *path (state) explosion problem*—hundreds of branch directions exist (including

```
1  double power(int x,int y
       ){
2      int exp;
3      double res;
4      if (y>0)
5        exp = y;
6      else
7        exp = -y;
8      res=1;
9      while (exp!=0){
10       res *= x;
11       exp -= 1;
12     }
13     if (y<=0)
14       if(x==0)
15         abort;
16       else
17         return 1.0/res;
18     return res;
19 }
```

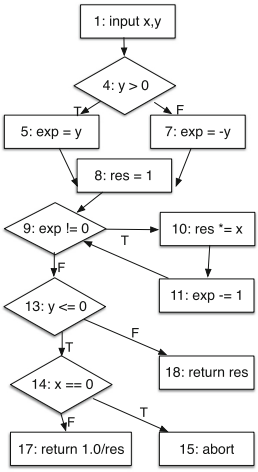


Fig. 2. An example: *power*.

those branches from the new explored paths)—will drastically slow down data-flow testing.

To mitigate this problem, the key idea of our approach is to reduce unnecessary path exploration and provide more guidance during execution. To achieve this, we designed a novel *cut-point guided search algorithm* (CPGS) to enhance SE, which leverages several key elements to prioritize the selection of ESs. First, we introduce *a guided search algorithm*, which leverages two metrics: (i) *cut points*, a sequence of control points that must be traversed through for any paths that could cover the target pair. For example, the cut points of du_1 are $\{l_4, l_8, l_9, l_{13}, l_{14}, l_{17}\}$. These critical points are used as intermediate goals during the search to narrow down the exploration space of SE. (ii) *instruction distance*, the distance between an ES and a target search goal in terms of number of program instructions on the control flow graph. Intuitively, an ES with closer (instruction) distance toward the goal can reach it more quickly. For example, when SE reaches l_9, it can fork two execution states, *i.e.*, following the *true* and the *false* branches. If our target goal is to reach l_{13}, the *false* branch will be prioritized since it has 1-instruction distance toward l_{13}, while the opposite branch has 3-instruction distance. Second, CPGS is enhanced by *a backtrack strategy* based on the number of executed instructions, which reduces the likelihood of trapping in tight loops. Third, we also introduce *a redefinition path pruning technique*, which detects and removes redundant ESs.

Table 1 shows the steps taken by our cut-point guided search algorithm to cover du_1. At the beginning, SE forks two ESs for the `if` statement at l_4, which

produces two *pending paths*[3], *i.e.*, l_{4T} and l_{4F}[4]. In detail, we maintain a tuple $(c, d)^{\mathbf{i}}$ that records the two aforementioned metrics for each pending path **i** in a priority queue, where c is the deepest covered cut point, and d is the shortest distance between the corresponding ES and the next target cut point. In each step, we choose the pending path **i** with the optimal value (c, d). For example, in Step **1**, Path **1** and Path **2** have the same values $(l_4, 2)$, and thus we randomly select one path, *e.g.*, Path **1**.

Later, in Step **2**, Path **1** produces two new pending paths, Path **3** and Path **4**. We choose Path **4** since it has the best value: it has sequentially covered the cut points $\{l_4, l_8, l_9\}$, and it is closer to the next cut point l_{13} than Path **3** on the control flow graph, so it is more likely to reach l_{13} more quickly. However, its *pc* is unsatisfiable. As a result, we give up exploring this pending path, and choose Path **3** (because it covers more cut points than Path **1**) in the next Step **3**, which induces Path **5** and Path **6**. At this time, our algorithm detects the variable *res* is redefined at l_{10} on Path **5** and Path **6**, according to the definition of DFT, it is useless to explore these two paths. So, Path **5** and Path **6** are pruned. This *redefinition path pruning* technique can rule out these invalid paths to speed up DFT. Note despite only two pending paths are removed in this case, a number of potential paths have actually been prevented from execution (see the example path in (3)), which can largely improve the performance of our search algorithm.

We choose the only remaining Path **2** to continue the exploration, which produces Path **7** and Path **8** in Step **5**. Again, we choose Path **8** to explore, which induces Path **9** and **10** in Step **6**. Here, for Path **10**, since it cannot reach the next target point l_{14}, its distance is set as ∞. As last, Path **9** is selected, and our algorithm finds Path **12** which covers du_1, and by solving its path constraint $y == 0 \wedge x \neq 0$, we can get one test input, *e.g.*, $t = (x \mapsto 1, y \mapsto 0)$, to satisfy the pair. The above process is enforced by the cut-point guided search, which only takes 7 steps to cover du_1. As we will demonstrate in Sect. 7, the cut point-guided search strategy is more effective for data-flow testing than the existing state-of-the-art search algorithms.

Step 3: CEGAR-Based Data-Flow Testing. In data-flow testing, classic data-flow analysis techniques [23,47,72] statically identify def-use pairs by analyzing data-flow relations. However, due to its conservativeness and limitations, infeasible pairs may be included, which greatly affects the effectiveness of SE for DFT. For example, the pair du_2 is identified as a def-use pair since there exists a def-clear control-flow path (*i.e.*, l_8, l_9, l_{13}, l_{18}) that can start from the variable definition (*i.e.*, l_8) and reach the use (*i.e.*, l_{18}). However, du_2 is infeasible (*i.e.*, no test inputs can satisfy it): If we want to cover its use statement at l_{18}, we cannot take the true branch of l_{13}, so $y > 0$ should hold. However, if $y > 0$, the variable *exp* will be assigned a positive value at l_5 by taking the true branch of l_4, and the redefinition statement at l_{10} *w.r.t.* the variable *res* will be executed.

[3] An pending path indicates a not fully-explored path (corresponding to an unterminated state).

[4] We use the line number followed by T or F to denote the *true* or *false* branch of the `if` statement at the corresponding line.

```
1  double power(int x, int y){
2     bool cover_flag = false;
3     int exp;
4     double res;
5     ...
6     res=1;
7     cover_flag = true;
8     while (exp!=0){
9        res *= x;
10       cover_flag = false;
11       exp -= 1;
12    }
13    ...
14    if(cover_flag) check_point();
15    return res;
16 }
```

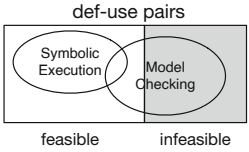
Fig. 3. The transformed function *power* with the test requirement encoded in highlighted statements.

As a result, such a path that covers the pair and avoids the redefinition at the same time does not exist, and du_2 is an infeasible pair. It is rather difficult for SE to conclude the feasibility unless it checks all program paths, which however is almost impossible due to infinite paths in real-world programs.

To counter the problem, our key idea is to reduce the data-flow testing problem into the path reachability checking problem in software model checking. We encode the test obligation of a target def-use pair into the program under test, and leverage the power of model checkers to check its feasibility. For example, in order to check the feasibility of du_2, we instrument the test requirement into the program as shown in Fig. 3. We first introduce a boolean variable *cover_flag* at l_2, and initialize it as `false`, which represents the coverage status of this pair. After the definition statement, the variable *cover_flag* is set as `true` (at l_7); *cover_flag* is set as `false` immediately after all the redefinition statements (at l_{10}). We check whether the property *cover_flag*==*true* holds (at l_{14}) just before the use statement. If the check point is reachable, the pair is feasible and a test case will be generated. Otherwise, the pair is infeasible, and will be excluded in the coverage computation. As we can see, this model checking based approach is flexible and can be fully automated.

Combined SE-CEGAR Based Data-Flow Testing. In data-flow testing, the set of test objectives include feasible and infeasible pairs. As we can see from the above two examples, SE, as a dynamic path-based exploration approach, can efficiently cover feasible pairs; while CEGAR, as a static software model checking approach, can effectively detect infeasible pairs (may also cover some feasible pairs).

The figure below shows the relation of these two approaches for data-flow testing. The white part represents the set of feasible pairs, and the gray part the set of infeasible ones. The SE-based approach is able to cover feasible pairs efficiently, but in general, due to the path explosion problem, it cannot detect infeasible pairs (this may waste a lot of testing time). The CEGAR-based approach is able to identify infeasible pairs efficiently (but may take more time to cover feasible ones). As a result, it is beneficial to combine these two techniques to complement each other with their strengths. Section 7 will demonstrate our observations, and validate that the combined approach can indeed achieve more efficient data-flow testing by reducing testing time as well as improving coverage.



5 Our Approach

This section explains the details of our approach. Our approach includes three steps: (1) static analysis, (2) symbolic execution based data flow testing and (3) software model checking based data flow testing.

5.1 Static Analysis

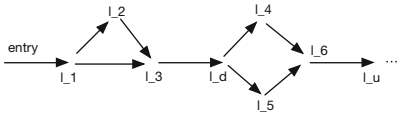
To improve the performance of SE-based data-flow testing, we use dominator analysis to analyze a set of *cut points* to effectively guide path exploration. In the following, we give some definitions.

Definition 4 (Dominator). *In a control-flow graph, a node m dominates a node n if all paths from the program entry to n must go through m, which is denoted as $m \gg n$. When $m \neq n$, we say m strictly dominates n. If m is the unique node that strictly dominates n and does not strictly dominate other nodes that strictly dominate n, m is an immediate dominator of n, denoted as $m \gg^I n$.*

Definition 5 (Cut Point). *Given a def-use pair $du(l_d, l_u, v)$, its cut points are a sequence of critical control points $c_1, \ldots, c_i, \ldots, c_n$ that must be passed through in succession by any control flow paths that cover this pair. The latter control point is the immediate dominator of the former one,* i.e., *$c_1 \gg^I \ldots c_i \gg^I l_d \gg^I \ldots c_n \gg^I l_u$. Each control point in this sequence is called a cut point.*

Note the *def* and the *use* statement (*i.e.*, l_d and l_u) of the pair itself also serve as the cut points. These cut points are used as the intermediate goals during path search to narrow down the search space. For illustration, consider the figure below: Let $du(l_d, l_u, v)$ be the target def-use pair, its cut points are $\{l_1, l_3, l_d, l_6, l_u\}$. Here the control point l_2 is not a cut point, since the path

$l_1, l_3, l_d, l_4, l_6, l_u$ can be constructed to cover the pair. For the similar reason, the control points l_4 and l_5 are not its cut points.



In practice, we use standard iterative data-flow analysis [47,72] to identify def-use pairs from the program under test. We give the implementation details in Sect. 6.

Algorithm 1: SE-based Data-flow Testing

Input: $du(l_d, l_u, x)$: a given def-use pair
Input: $C = \{c_1, c_2, \ldots, c_n\}$: the cut points of du
Output: input t that satisfies du or nil if none is found within the given time bound

```
1  let W be a worklist of execution states
2  let ES_0 be the initial execution state
3  W ← W ∪ {ES_0}
   // the core process of symbolic execution
4  repeat
5      ExecutionState ES ← selectState(W)
6      while ES.instructionType!=FORK or EXIT do
7          ES.executeInstruction()
8      if ES.instructionType=EXIT then W ← W \ {ES}
9      if ES.instructionType = FORK then
10         Instruction Fr = ES.currentInstruction;
11         ExecutionState ES' ← new executionState(ES)
12         ES.newNode ← Fr(T)
13         ES'.newNode ← Fr(F)
14         W ← W ∪ {ES'}
15     PendingPath p ← ES.path
16     if p covers du then return t ← getTestCase(ES)
       // the redefinition path pruning heuristic
17     if variable x (in du) is redefined by p then
18         W ← W \ {ES, ES'}
19 until W.size()=0 or timeout()
   // the core algorithm of execution state selection
20 Procedure selectState(reference worklist W)
21 let ES' be the next selected execution state
   // j is the index of a cut point, w is the state weight
22 j ← 0, w ← ∞
23 foreach ExecutionState ES ∈ W do
24     PendingPath pp ← ES.path
       // c_1, ..., c_i are sequentially-covered, while c_{i+1} not yet
25     i ← index of the cut point c_i on pp
26     StateWeight sw ← distance(es, c_{i+1})^{-2} + instructionsSinceCovNew(es)^{-2}
27     if i > j ∨ (i == j ∧ sw > w) then
28         ES' ← ES, j ← i, w ← sw
29 W ← W \ {ES'}
30 return ES'
```

5.2 SE-Based Approach for Data-Flow Testing

This section explains the symbolic execution-based approach for data-flow testing. Algorithm 1 gives the details. This algorithm takes as input a target def-use pair du and its cut points C, and either outputs the test case t that satisfies du, or *nil* if it fails to find a path that can cover du.

It first selects one execution state ES from the worklist W which stores all the execution states during symbolic execution. It then executes the current program instruction referenced by ES, and update ES according to the instruction type (Lines 6–14, *cf.* Sect. 3.2). Basically, one instruction can be one of three types: *sequential instruction* (*e.g.*, assignment statements), *forking instruction* (*e.g.*, `if` statements, denoted as $FORK$), and *exit instruction* (*e.g.*, program exits or runtime errors, denoted as $EXIT$). When it encounters sequential instructions, ES is updated accordingly by function *executeInstruction* (Lines 6–7). Specifically, function *executeInstruction* will internally (1) execute the current instruction, and (2) update ES (including the symbolic state, the reference to next instruction and the corresponding instruction type). When it encounters $FORK$ instructions, one new execution state ES' will be created. The two states ES and ES' will explore both sides of the fork, respectively, and the corresponding subpaths of ES and ES' will be updated to $ES.path{+}Fr(T)$ and $ES.path{+}Fr(F)$, respectively (Lines 9–14). Here, Fr denotes the forking point, and T and F represent the *true* and *false* directions, respectively. If the target pair du is covered by the pending path p of ES, a test input t will be generated (Line 16). If the variable x of du is redefined on p between the *def* and *use* statement, a redefinition path pruning heuristic will remove those invalid states (Lines 17–18, more details will be explained later). The algorithm will continue until either the worklist W is empty or the given testing time is exhausted (at Line 19).

The algorithm core is the state selection procedure, *i.e.*, *selectState* (detailed at Lines 20–30), which integrates several heuristics to improve the overall effectiveness. Figure 4 conceptually shows the benefits of their combination (the red path is a valid path that covers the pair), which can efficiently steer exploration towards the target pair, and reduce as many unnecessary path explorations as possible: (1) the cut point guided search guides the state exploration towards the target pair more quickly; (2) the backtrack strategy counts the number of executed instructions to prevent the search from being trapped in tight loops, and switches to alternative search directions; and (3) the redefinition path pruning technique effectively prunes redundant search space. In detail, we use Formula 4 to assign the weights to all states, and achieve the heuristics (1) and (2).

$$state\_weight(es) = (c_{max}, \frac{1}{d^2} + \frac{1}{i^2}) \tag{4}$$

where, ES is an execution state, c_{max} is the deepest covered cut point, d is the instruction distance toward the next uncovered cut point, and i is the number of executed instructions since the last new instruction have been covered. Below, we explain the details of each heuristic.

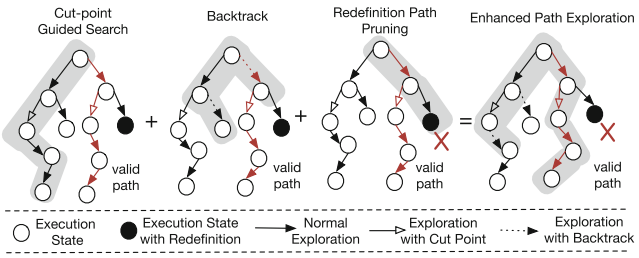


Fig. 4. Enhanced path exploration in symbolic execution: combine cut-point guided search, backtrack strategy and redefinition path pruning. Each subfigure denotes the execution tree generated by symbolic execution.

Cut Point Guided Search. The cut-point guided search strategy (at Lines 23–28) aims to search for the ES whose pending path has covered the deepest cut point, and tries to reach the next goal, *i.e.*, the next uncovered cut point, as quickly as possible. For an ES, its pending path is a subpath that starts from the program entry and reaches up to the program location of it. If this path has sequentially covered the cut point $c_1, c_2, \ldots, c_i$ but not c_{i+1}, c_i is the deepest covered cut point, and c_{i+1} is the next goal to reach. The strategy always prefers to select the ES that has covered the deepest cut point (at Lines 26–28, indicated by the condition $i > j$). The intuition is that the deeper cut point an ES can reach, the closer the ES toward the pair is.

When more than one ES covers the deepest cut point (indicated by the condition $i{==}j$ at Line 27), the ES that has the shortest distance toward next goal will be preferred (at Lines 26–28). The intuition is that the closer the distance is, the more quickly the ES can reach the goal. We use *dist(es,* c_{i+1}*)* to present the distance between the location of *es* and the next uncovered cut point c_{i+1}. The distance is approximated as the number of instructions along the shortest control-flow path between the program locations of *es* and c_{i+1}.

Backtrack Strategy. To avoid the execution falling into the tight loops, we assign an ES with lower priority if the ES is not likely to cover new instructions. In particular, for each ES, the function *instrsSinceCovNew*, corresponding to i in Formula (4), counts the number of executed instructions since the last new instruction is covered (at Line 26). The ES, which has a larger value of *instrsSinceCovNew*, is assumed that it has lower possibility to cover new instructions. Intuitively, this heuristic prefers the ES which is able to cover more new instructions, if a ES does not cover new instructions for a long time, the strategy will backtrack to another ES via lowering the weight of the current ES.

Redefinition Path Pruning. A redefinition path pruning technique checks whether the selected ES has redefined the variable x in *du*. If the ES is invalid (*i.e.*, its pending path has redefined x), it will be discarded and *selectState* will choose another one (at Lines 17–18). The reason is that, according to the

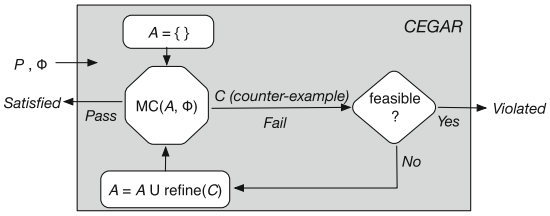


Fig. 5. Paradigm of CEGAR-based Model Checking

definition of DFT (*cf.* Definition 3), it is impossible to find def-clear paths by executing those invalid ESs.

Further, by utilizing the path-sensitive information from SE, we can detect variable redefinitions, especially caused by variable aliases, more precisely. Variable aliases appear when two or more variable names refer to the same memory location. So we designed a lightweight variable redefinition detection algorithm in our framework. Our approach operates upon a simplified three-address form of the original code[5], so we mainly focus on the following statement forms where variable aliases and variable redefinitions may appear:

- Alias inducing statements: (1) $p{:=}q$ ($*p$ is an alias to $*q$), (2) $p{:=}\&x$ ($*p$ is an alias to x)
- Variable definition statements: (3) $*p{:=}y$ ($*p$ is defined by y), (4) $v{:=}y$ (v is defined by y)

Here, p and q are pointer variables, x and y non-pointer variables, and ":=" the assignment operator.

Initially, a set A is maintained, which denotes the variable alias set *w.r.t.* the variable x of du. At the beginning, it only contains x itself. During path exploration, if the executed statement is (1) or (2), and $*q$ or $x \in A$, $*p$ will be added into A since $*p$ becomes an alias of x. If the executed statement is (1), and $*q \notin A$ but $x \in A$, $*p$ will be excluded from A since it becomes an alias of another variable instead of x. If the executed statement is (3) or (4), and $*p \in A$ or $x \in A$, the variable is redefined by another variable y.

5.3 CEGAR-Based Approach for Data-Flow Testing

Counter-example guided abstract refinement (CEGAR) is a well-known software model checking technique that statically proves program correctness *w.r.t.* properties (or specifications) of interest [56]. Figure 5 shows the basic paradigm of CEGAR, which typically follows an *abstract-check-refine* paradigm. Given the program P (*i.e.*, the actual implementation) and a safety property ϕ of interest, CEGAR first *abstracts* P into a model A (typically represented as a finite

[5] We use CIL as the C parser to transform the source code into an equivalent simplified form using the –dosimplify option, where one statement contains at most one operator.

automaton), and then *checks* the property ϕ against A. If the abstract model A is error-free, then so is the original program P. If it finds a path on the model A that violates the property ϕ, it will check the feasibility of this path: is it a genuine path that can correspond to a concrete path in the original program P, or due to the result of the current coarse abstraction? If the path is feasible, CEGAR returns a counter-example path C to demonstrate the violation of ϕ. Otherwise, CEGAR will utilize this path C to *refine* A by adding new predicates, and continue the checking until it either finds a genuine path that violates ϕ or proves that ϕ is always satisfied in P. Or since this model checking problem itself is undecidable, CEGAR does not terminate and cannot conclude the correctness of ϕ.

To exploit the power of CEGAR, our approach reduces the problem of data flow testing to the problem of model checking. The CEGAR-based approach can operate in two phases [12] to generate tests, *i.e.*, *model checking* and *tests from counter-examples*. (1) It first uses model checking to check whether the specified program location l is reachable such that the predicate of interest q (*i.e.*, the safety property) can be satisfied at that point. (2) If l is reachable, CEGAR will return a counter-example path p that establishes q at l, and generate a test case from the corresponding path constraint of p. Otherwise, if l is not reachable, CEGAR will conclude no test inputs can reach l.

The key idea is to encode the test obligation of a target def-use pair into the program under test. We instrument the original program P to P', and reduce the test generation for P to path reachability checking on P'. In particular, we follow three steps: (1) We introduce a variable *cover_flag* into P, which denotes the cover status of the target pair, and initialize it as *false*. (2) The variable *cover_flag* is set as *true* immediately after the *def* statement, and set as *false* immediately after all redefinition statements. (3) Before the *use* statement, we set the target predicate as *cover_flag*==*true*. As a result, if the use statement is reachable when the target predicate holds, we can obtain a counter-example (*i.e.*, a test case) and conclude the pair is feasible. Otherwise, if unreachable, we can safely conclude that the pair is infeasible (or since the problem itself is undecidable, the algorithm does not terminate, and gives *unknown*).

Generability of the SMC-Based Approach. This reduction approach is flexible to implement on any CEGAR-based model checkers. It is also applicable for other software model checking techniques, *e.g.*, Bounded Model Checking (BMC). Given a program, BMC unrolls the control flow graph for a fixed number of k steps, and checks whether the property p at a specified program location l is violated or not. Different from the modern (dynamic) symbolic execution techniques, BMC executes on pure symbolic inputs without using any concrete input values, and usually aims to systematically checking reachability within given bounds. Different from CEGAR, BMC searches on all program computations without abstraction and typically backtracks the search when a given (loop or search depth) bound is reached. Although BMC in general cannot prove infeasibility as certain, in Sect. 7 we will show the BMC-based approach can still serve as a heuristic-criterion to identify hard-to-cover (probably infeasible) pairs and

particularly effective for specific types of programs. In fact, the infeasible pairs concluded by BMC can be regarded as valid modulo the given checking bounds.

6 Framework Design and Implementation

We realized our hybrid data-flow testing framework for `C` programs. In our original work [84], we implemented the SE-based approach on our own concolic testing based tool CAUT [85,91], while in this article we built the enhanced SE-based approach on KLEE [18], a robust execution-generated testing based symbolic execution engine, to fully exhibit its feasibility. As for the SMC-based approach, we instantiated it with two different types of software model checking techniques, *i.e.*, CEGAR and BMC. In all, our framework combines the SE-based and SMC-based approaches together to achieve efficient DFT.

In the static analysis phase, we identify def-use pairs, cut points, and related static program information (*e.g.*, variable definitions and aliases) by using CIL [70] (C Intermediate Language), which is an infrastructure for C program analysis and transformation. We first build the control-flow graph (CFG) for each function in the program under test, and then construct the inter-procedural CFG (ICFG) for the whole program. We perform standard iterative data-flow analysis techniques [47,72], *i.e.*, reaching definition analysis, to compute def-use pairs. For each variable use, we compute which definitions on the same variable can reach the use through a def-clear path on the control-flow graph. A def-use pair is created as a test objective for each use with its corresponding definition. We treat each formal parameter variable as defined at the beginning of its function and each argument parameter variable as used at its function call site (*e.g.*, library calls). For global variables, we treat them as initially defined at the beginning the entry function (*e.g.*, the *main* function), and defined/used at any function where they are defined/used.

In the current implementation, we focus on the def-use pairs with local variables (intra-procedural pairs) and global variables (inter-procedural pairs). Following prior work on data-flow testing [89], we currently do not consider the def-use pairs induced by pointer aliases. Thus, we may miss some def-use pairs, but we believe that this is an independent issue (not the focus of this work) and does not affect the effectiveness of our testing approach. More sophisticated data-flow analysis techniques (*e.g.*, dynamic data-flow analysis [30]) or tools (*e.g.*, Frama-C [59]) can be used to mitigate this problem.

Specifically, to improve the efficiency of state selection algorithm (*cf.* Algorithm 1) in KLEE, we use the priority queue to sort execution states according to their weights. The algorithmic complexity is $\mathcal{O}(n \log n)$ (n is the number of execution states), which is much faster than using a list or array. The software model checkers are used as black-box to enforce data-flow testing. The benefit of this design choice is that we can flexibly integrate any model checker without any modification or adaption. CIL transforms the program under test into a simplified code version, and encodes the test requirements of def-use pairs into the program under test. Both SE-based and SMC-based tools takes as input

the same CIL-simplified code. Function stubs are used to simulate C library functions such as string, memory and file operations to improve the ability of symbolic reasoning. To compute the data-flow coverage during testing, we implement the classic *last definition* technique [52] in KLEE. We maintain a table of def-use pairs, and insert probes at each basic block to monitor the program execution. The runtime routine records each variable that has been defined and the block where it was defined. When a block that uses this defined variable is executed, the last definition of this variable is located, we check whether the pair is covered. Our implementations are publicly available at [87].

7 Evaluation

This section aims to evaluate whether our combined testing approach can achieve efficient data-flow testing. In particular, we intend to investigate (1) whether the core SE-based approach can quickly cover def-use pairs; (2) whether the SMC-based reduction approach is feasible and practical; and (3) whether the combined approach can be more effective for data-flow testing.

7.1 Research Questions

- ***RQ1:*** In the data-flow testing *w.r.t.* all def-use coverage, what is the performance difference between different existing search strategies (*e.g.*, DFS, RSS, RSS-MD2U, SDGS) and CPGS (our cut point guided path search strategy) in terms of testing time and number of covered pairs for the SE-based approach?
- ***RQ2:*** How is the practicability of the CEGAR-based reduction approach as well as the BMC-based approach in terms of testing time and number of identified feasible and infeasible pairs?
- ***RQ3:*** How efficient is the combined approach, which complements the SE-based approach with the SMC-based approach, in terms of testing time and coverage level, compared with the SE-based approach or the SMC-based approach alone?

7.2 Evaluation Setup

Testing Environment. All evaluations were run on a 64bit Ubuntu 14.04 physical machine with 24 processors (2.60 GHz Intel Xeon(R) E5-2670 CPU) and 94 GB RAM.

Framework Implementations. The SE-based approach of our hybrid testing framework was implemented on KLEE (v1.1.0), and the SMC-based approach was implemented on two different software model checking techniques, CEGAR and BMC. In particular, we chose three different software model checkers[6], *i.e.*, BLAST [13] (CEGAR-based, v2.7.3), CPAchecker [14] (CEGAR-based, v.1.7)

[6] We use the latest versions of these model checkers at the time of writing.

and CBMC [24] (BMC-based, v5.7). We chose different model checkers, since we intend to gain more overall understandings of the practicality of this reduction approach. Note that the CEGAR-based approach can give definite answers of the feasibility, while the BMC-based approach is used as a heuristic-criterion to identify hard-to-cover (probably infeasible) pairs.

Program Subjects. Despite data-flow testing has been continuously investigated in the past four decades, the standard benchmarks for evaluating data-flow testing techniques are still missing. To this end, we took substantial efforts and dedicatedly constructed a repository of benchmark subjects by following these steps. *First*, we collected the subjects from prior work on data-flow testing. We conducted a thorough investigation on all prior work (99 papers [83] in total) related to data-flow testing, and searched for the adopted subjects. After excluding the subjects whose source codes are not available or not written in C language, we got 26 unique subjects [32,35,39,42,54,57,66–69,75,76,80,81,84,92,94]. We then manually inspected these programs and excluded 19 subjects which are too simple, we finally got 7 subjects. These 7 subjects include mathematical computations and classic algorithms. *Second*, we included 7 Siemens subjects from SIR [82], which are widely used in the experiments of program analysis and software testing [48,54,90]. These subjects involve numeral computations, string manipulations and complex data structures (*e.g.*, `pointers`, `structs`, and `lists`). *Third*, we further enriched the repository with the subjects from the SV-COMP benchmarks [33], which are originally used for the competition on software verification. The SV-COMP benchmarks are categorized in different groups by their features (*e.g.*, concurrency, bit vectors, floats) for evaluating software model checkers. In order to reduce potential evaluation biases in our scenario, we carefully inspected all the benchmarks and finally decided to select subjects from the "Integers and Control Flow" category based on these considerations: (1) the subjects in this category are real-world (medium-sized or large-sized) OS device drivers (*cf.* [11], Sect. 4), while many subjects in other categories are hand-crafted, small-sized programs; (2) the subjects in this category have complicated function call chains or control-flow structures, which are more appropriate for our evaluation; (3) the subjects do not contain specific features that may not be supported by KLEE (*e.g.*, concurrency, floating point numbers). We finally selected 16 subjects in total from the *ntdrivers* and *ssh* groups therein (we excluded other subjects with similar control-flow structures by diffing the code). The selected subjects have rather complex control-flows. For example, the average cyclomatic complexity of functions in the `ssh` group exceeds 88.5[7] (computed by *Cyclo* [1]) *Fourth*, we also included three core program modules from the industrial projects from our research partners. The first one is an engine management system (*osek_control*) running on an automobile operating system conforming to the OSEK/VDX standard. The second one is a satellite gesture control program (*space_control*). The third one is a control program (*subway_control*) from a subway signal. All these three industrial pro-

[7] Cyclomatic complexity is a software metric that indicates the complexity of a program. The standard software development guidelines recommend the cyclomatic complexity of a module should not exceeded 10.

Table 2. Subjects of the constructed data-flow testing benchmark repository

Subject	#ELOC	#DU	Description
factorization	43	47	compute factorization
power	11	11	compute the power x^y
find	66	99	permute an array's elements
triangle	32	46	classify an triangle type
strmat	67	32	string pattern matching
strmat2	88	38	string pattern matching
textfmt	142	73	text string formatting
tcas	195	86	collision avoidance system
replace	567	387	pattern matching and substitution
totinfo	374	279	compute statistics given input data
printtokens	498	240	lexical analyzer
printtokens2	417	192	lexical analyzer
schedule	322	118	process priority scheduler
schedule2	314	107	process priority scheduler
kbfiltr	557	176	`ntdrivers` group
kbfiltr2	954	362	`ntdrivers` group
diskperf	1,052	443	`ntdrivers` group
floppy	1,091	331	`ntdrivers` group
floppy2	1,511	606	`ntdrivers` group
cdaudio	2,101	773	`ntdrivers` group
s3_clnt	540	1,677	`ssh` group
s3_clnt_termination	555	1,595	`ssh` group
s3_srvr_1a	198	574	`ssh` group
s3_srvr_1b	127	139	`ssh` group
s3_srvr_2	608	2,130	`ssh` group
s3_srvr_7	624	2,260	`ssh` group
s3_srvr_8	631	2,322	`ssh` group
s3_srvr_10	628	2,200	`ssh` group
s3_srvr_12	696	3,125	`ssh` group
s3_srvr_13	642	2,325	`ssh` group
osek_control	4,589	927	one module of engine management system
space_control	5,782	1,739	one module of satellite gesture control software
subway_control	5,612	2,895	one module of subway signal control software

grams were used in our previous research work [74,85,98], and have complicated data-flow interactions. *Finally*, to ensure each tool (where our framework is built upon) can correctly reason these subjects, we carefully read the documentation of each tool to understand their limitations, manually checked each program and added necessary function stubs (*e.g.*, to simulate such C library functions as string, memory, and file operations) but without affecting their original program logic and structures. This is important to reduce validation threats, and also provides a more fair comparison basis. In total, we got 33 subjects with different characteristics, including mathematical computation, classic algorithms, utility programs, device drivers and industrial control programs. These subjects allow us to evaluate diverse data-flow scenarios. Table 2 shows the detailed statistics of these subjects, which includes the executable lines of code (computed by *cloc* [2]), the number of def-use pairs (including intra- and inter-procedural pairs), and the brief functional description.

Search Strategies for Comparison. To our knowledge, there exists no specific guided search strategies on KLEE to compare with our strategy. Thus, we compare our cut-point guided search strategy with several existing search strategies. In particular, we chose two generic search strategies (*i.e.*, depth-first and random search), one popular (statement) coverage-optimized search strategy. In addition, we implemented one search strategy for directed testing on KLEE, which is proposed by prior work [61,64,97]. We detail them as follows.

- *Depth First-Search (DFS)*: always select the latest execution state from all states to explore, and has little overhead in state selection.
- *Random State Search (RSS)*: randomly select an execution state from all states to explore, and able to explore the program space more uniformly and less likely to be trapped in tight loops than other strategies like DFS.
- *Coverage-Optimized Search (COS)*: compute the weights of the states by some heuristics, *e.g.*, the minimal distance to uncovered instructions (*md2u*) and whether the state recently covered new code (*covnew*), and randomly select states *w.r.t.* these weights. These heuristics are usually interleaved with other search strategies in a round-robin fashion to improve their overall effectiveness. For example, *RSS-COS:md2u* (*RSS-MD2U* for short) is a popular strategy used by KLEE, which interleaves *RSS* with *md2u*.
- *Shortest Distance Guided Search (SDGS)*: always select the execution state that has the shortest (instruction) distance toward a target instruction in order to cover the target as quickly as possible. This strategy has been widely applied in single target testing [61,64,97]. In the context of data-flow testing, we implemented this strategy in KLEE by setting the *def* as the first goal and then the *use* as the second goal after the *def* is covered.

7.3 Case Studies

We conducted three case studies to answer the research questions. Note that in this paper we focus on the classic data-flow testing [39,89], *i.e.*, targeting one def-use pair at one run. In ***Study 1***, we answer ***RQ1*** by comparing the performance of different search strategies that were implemented on KLEE. In detail, we use two metrics: (1) *number of covered pairs*, *i.e.*, how many def-use pairs can be covered; and (2) *testing time*, *i.e.*, how long does it take to cover the pair(s) of interest. The *testing time* is measured by the median value and the semi-interquartile range (SIQR)[8] of the times consumed on those covered (*feasible*) pairs[9].

In the evaluation, the maximum allowed search time on each pair is set as 5 min. Under this setting, we observed all search strategies can thoroughly test

[8] SIQR = (Q3-Q1)/2, which measures the variability of testing time, where Q1 is the lower quartile, and Q3 is the upper quartile.

[9] In theory, the symbolic execution-based approach cannot identify infeasible pairs unless it enumerates all possible paths, which however is impossible in practice. Therefore, we only consider the testing times of covered (feasible) pairs for performance evaluation.

each subject (*i.e.*, reach their highest coverage rates). To mitigate the algorithm randomness, we repeat the testing process 30 times for each program/strategy and aggregate their average values as the final results for all measurements.

In ***Study 2***, we answer ***RQ2*** by evaluating the practicability of the SMC-based reduction approach on two different model checking techniques, CEGAR and BMC. Specifically, we implemented the reduction approach on three different model checkers, BLAST, CPAchecker and CBMC. We use the following default command options and configurations according to their user manuals and the suggestions from the tool developers, respectively:

BLAST: `ocamltune blast -enable-recursion -cref -lattice -noprofile -nosserr -quiet`
CPAchecker: `cpachecker -config config/predicateAnalysis.properties -skipRecursion`
CBMC: `cbmc --slice-formula --unwind nr1 --depth nr2`

We have not tried to particularly tune the optimal configurations of these tools for different subjects under test, since we aim to investigate the practicability of our reduction approach in general. Specifically, BLAST and CPAchecker are configured based on predicate abstraction. For BLAST, we use an internal script `ocamltune` to improve memory utilization for large programs; for CPAchecker, we use its default predicate abstraction configuration predicateAnalysis.properties. We use the option -enable-recursion of BLAST and -skipRecursion of CPAchecker to set recursion functions as *skip*. Due to CBMC is a bounded model checker, it may answer *infeasible* for actual *feasible* pairs if the given checking bound is too small. Thus, we set the appropriate values for the –unwind and –depth options, respectively, for the number of times loops to be unwound and the number of program steps to be processed. Specially, we determine the parameter values of –unwind and –depth options by a binary search algorithm to ensure that CBMC can identify as many pairs as possible within the given time bound. This avoids wasting testing budget on unnecessary path explorations, and also achieves a more fair evaluation basis. Therefore, each subject may be given different parameter values (the concrete parameter values of all subjects are available at [87]).

Specifically, we use two metrics: (1) *number of feasible, infeasible, and unknown pairs*; and (2) *testing (checking) time of feasible and infeasible pairs* (denoted in medians). The maximum testing time on each def-use pair is constrained as 5 min (*i.e.*, 300 s, the same setting in RQ1). For each def-use pair, we also run 30 times to mitigate algorithm randomness.

In ***Study 3***, we answer ***RQ3*** by combining the SE-based and SMC-based approaches. We interleave these two approaches as follows: the SE-based approach (configured with the cut point-guided path search strategy and the same settings in ***RQ1***) is first used to cover as many pairs as possible; then, for the remaining uncovered pairs, the SMC-based approach (configured with the same settings in ***RQ2***) is used to identify infeasible pairs (may also cover some feasible pairs). We continue the above iteration of the combined approach until the maximum allowed time bound (5 min for each pair) is used up. Specifically, we

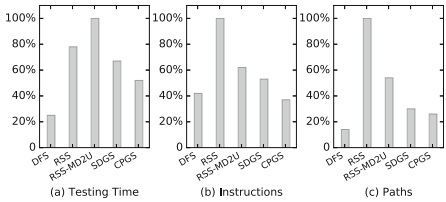


Fig. 6. Performance of each search strategy in terms of total testing time, number of executed program instructions, and number of explored program paths (normalized in percentage) on all 33 subjects.

increase the time bound by 3 times at each iteration, *i.e.*, 10 s, 30 s, 90 s and 300 s.

Specifically, we use two metrics: (1) *coverage rate*; and (2) *total testing time*, *i.e.*, the total time required to enforce data-flow testing on all def-use pairs of one subject. The coverage rate C is computed by Formula 5, where *nTestObj* is the total number of pairs, and *nFeasible* and *nInfeasible* are the number of identified feasible and infeasible ones, respectively.

$$C = \frac{nFeasible}{nTestObj - nInfeasible} \times 100\% \tag{5}$$

In all case studies, the testing time was measured in CPU time via the *time* command in Linux. In particular, the testing time did not include IO operations for logging the testing results. We tested 31,634 ELOC with 28,354 pairs in total. It took us nearly one and half months to run the experiments and analyze the results.

7.4 Study 1

Table 3 shows the detailed performance statistics of different search strategies. The column *Subject* represents the subject under test, DFS, RSS, RSS-MD2U, SDGS, CPGS, respectively, represent the search strategies. For each subject/strategy, it shows the number of covered def-use pairs (denoted by N), the median value of testing times (denoted by M) and the semi-interquartile range of testing times (denoted by $SIQR$) on all covered pairs. In particular, for each subject, we underscore the strategy with lowest median value. The last row gives the total number of covered pairs. From Table 3, we can observe (1) Given enough testing time for all strategies (*i.e.*, 5 min for each pair), CPGS covers 4215, 2152, 1320 and 1563 more pairs, respectively, than DFS, RSS, RSS-MD2U and SDGS. It means CPGS achieves 40%, 21.3%, 12.1%, 14.6% higher data-flow coverage than these strategies, respectively. (2) By comparing the median values of CPGS with those of other strategies, CPGS achieves more efficient data-flow testing in 14/33, 23/33, 32/33, 26/33 subjects than DFS, RSS, RSS-MD2U and SDGS,

Table 3. Performance statistics of different search strategies for data-flow testing (the testing time is measured in seconds).

Subject	DFS		RSS		RSS-MD2U		SDGS		CPGS	
	N	M (SIQR)	N	M (SIQR)	N	M (SIQR)	N	M (SIQR)	N	M (SIQR)
factorization	22	0.07 (0.01)	22	0.07 (0.01)	22	0.08 (0.02)	22	0.05 (0.01)	22	0.06 (0.01)
power	6	0.14 (0.00)	9	0.12 (0.01)	9	0.05 (0.01)	5	0.04 (0.00)	9	0.04 (0.00)
find	77	0.89(0.64)	49	0.19 (0.54)	52	0.26 (0.31)	51	0.63 (3.35)	56	0.22 (0.12)
triangle	22	0.24 (0.06)	22	0.24 (0.03)	22	0.26 (0.05)	22	0.25 (0.09)	22	0.13 (0.01)
strmat	26	2.84 (1.41)	30	0.10 (0.02)	30	0.13 (0.03)	30	0.12 (0.16)	30	0.10 (0.02)
strmat2	28	2.85 (1.40)	32	0.09 (0.01)	32	0.11 (0.02)	32	0.11 (0.03)	32	0.09 (0.02)
textfmt	37	0.16 (0.08)	33	0.05 (0.01)	33	0.11 (0.04)	34	0.06 (0.01)	34	0.06 (0.01)
tcas	55	0.13 (0.03)	55	0.21 (0.07)	55	0.67 (0.43)	55	0.16 (0.06)	55	0.14 (0.06)
replace	69	0.77 (0.14)	308	1.96 (15.23)	312	30.31 (21.97)	295	4.67 (5.46)	309	1.15 (3.58)
totinfo	13	0.52 (0.07)	24	0.42 (0.13)	24	0.64 (0.08)	24	0.42 (0.06)	26	0.52 (0.05)
printtokens	48	0.96 (0.62)	115	34.69 (24.16)	106	33.68 (22.59)	107	16.40 (25.53)	115	12.23 (20.21)
printtokens2	124	0.47 (0.32)	148	0.80 (3.72)	149	20.67 (18.42)	149	0.83 (3.48)	154	0.51 (1.43)
schedule	15	0.16 (0.03)	83	0.23 (3.98)	86	0.76 (5.05)	77	0.22 (1.67)	86	0.22 (1.84)
schedule2	14	0.15 (0.02)	78	0.20 (0.12)	78	0.48 (1.11)	77	0.21 (0.10)	77	0.21 (0.08)
cdaudio	562	3.13 (0.41)	562	3.27 (0.48)	562	15.54 (7.11)	562	3.77 (2.52)	562	3.08 (0.51)
diskperf	285	0.89 (0.19)	302	0.97 (0.21)	302	1.97 (4.80)	299	0.95 (0.23)	302	0.92 (0.18)
floppy	249	0.62 (0.11)	249	0.67 (0.11)	249	2.11 (1.76)	249	0.72 (0.14)	249	0.66 (0.13)
floppy2	510	2.22 (0.37)	510	2.14 (0.42)	510	6.44 (3.44)	510	3.62 (1.66)	510	2.03 (0.39)
kbfiltr	116	0.26 (0.05)	116	0.28 (0.05)	116	0.49 (0.41)	116	0.31 (0.04)	116	0.27 (0.05)
kbfiltr2	266	0.97 (0.15)	266	0.94 (0.18)	266	4.18 (3.51)	266	2.11 (1.08)	266	0.90 (0.20)
s3_srvr_1a	113	0.72 (0.17)	171	0.75 (0.19)	171	0.76 (0.19)	165	0.65 (0.17)	171	0.58 (0.18)
s3_srvr_1b	30	0.08 (0.02)	43	0.08 (0.02)	43	0.07 (0.02)	43	0.07 (0.02)	45	0.08 (0.02)
s3_clnt	647	9.64 (2.01)	648	11.93 (2.64)	647	22.91 (8.19)	633	12.45 (2.45)	648	10.32 (1.88)
s3_clnt_termination	333	9.20 (1.64)	332	8.81 (1.87)	332	12.14 (1.29)	332	9.56 (1.74)	414	6.54 (1.03)
s3_srvr_2	414	14.35 (2.67)	695	24.23 (17.42)	695	31.86 (15.44)	681	19.93 (8.00)	695	16.45 (3.76)
s3_srvr_7	420	16.29 (3.44)	710	27.82 (20.06)	710	34.99 (17.11)	686	26.93 (12.46)	815	19.47 (5.41)
s3_srvr_8	416	16.77 (3.10)	704	23.61 (14.03)	698	36.15 (16.39)	690	23.45 (7.21)	798	17.04 (4.23)
s3_srvr_10	431	15.26 (2.40)	683	21.34 (5.19)	683	30.21 (7.03)	664	20.73 (5.85)	683	18.37 (3.90)
s3_srvr_12	433	25.76 (3.84)	395	39.51 (21.68)	539	64.25 (38.99)	486	39.50 (18.04)	724	25.88 (10.08)
s3_srvr_13	437	15.69 (2.07)	489	25.25 (18.07)	558	33.78 (21.20)	572	23.98 (11.49)	744	15.77 (6.12)
osek_control	398	7.69 (2.47)	426	15.77 (14.68)	549	23.32 (17.39)	538	14.17 (6.17)	639	6.15 (3.23)
space_control	582	15.90 (7.76)	812	33.49 (20.61)	990	48.77 (23.08)	961	28.86 (15.78)	1,178	6.32 (7.09)
subway_control	827	13.44 (7.69)	967	42.76 (28.65)	1,290	68.61 (31.73)	1,244	38.11 (21.46)	1,654	10.72 (6.72)
Total	8,025	–	10,088	–	10,920	–	10,677	–	12,240	–

respectively. Note that the median value of DFS is low because it only covers many easily reachable pairs, which also explains why it achieves lowest coverage.

Figure 6 shows the performance of these search strategies on all 33 subjects in terms of total testing time, the number of executed program instructions, and the number of explored program paths (due to the data difference, we normalized them in percentage). Note these three metrics are all computed on the covered pairs. Apart from DFS (since it achieves rather low data-flow coverage), we can see CPGS outperforms all the other testing strategies. In detail, CPGS reduces testing time by 15–48%, the number of executed instructions by 16–63%, and the number of explored paths by 28–74%. The reason is that CPGS narrows down the search space by following the cut points and prunes unnecessary paths.

Answer to RQ1: *In summary, our cut-point guided search (CPGS) strategy performs the best for data-flow testing. It improves 12–40% data-flow coverage, and at the same time reduces the total testing time by 15–48% and the number of explored paths by 28–74%. Therefore, the SE-based approach, enhanced with the cut point guided search strategy, is efficient for data-flow testing.*

Table 4. Performance statistics of the SMC-based reduction approach $CEGAR_{BLAST}$, $CEGAR_{CPAchecker}$ and BMC_{CBMC} for data-flow testing (the testing time is measured in seconds), where * denotes the numbers in the corresponding columns are only valid modulo the given checking bound for BMC_{CBMC}.

Subject	$CEGAR_{BLAST}$					$CEGAR_{CPAchecker}$					BMC_{CBMC}				
	F	I	U	M_F	M_I	F	I	U	M_F	M_I	F	I*	U*	M_F	M_I*
factorization	35	4	8	0.04	0.20	26	4	17	3.26	3.04	41	6	0	0.34	0.28
power	9	2	0	0.03	0.49	9	2	0	3.10	2.97	9	2	0	0.13	0.12
find	85	12	2	6.44	3.22	74	14	11	4.37	3.60	77	22	0	0.29	0.29
triangle	22	24	0	0.04	0.69	22	24	0	3.09	2.83	22	24	0	0.11	0.11
strmat	30	2	0	1.81	1.39	30	2	0	4.67	2.98	30	2	0	0.15	0.15
strmat2	32	6	0	5.08	1.46	32	6	0	4.91	3.79	32	6	0	0.15	0.15
textfmt	47	18	8	10.08	13.90	53	20	0	12.69	5.50	53	20	0	3.84	3.95
tcas	55	31	0	1.35	1.31	55	31	0	4.08	3.43	55	31	0	0.13	0.13
replace	275	73	39	6.17	13.60	211	48	128	11.21	10.84	339	48	0	101.47	93.20
totinfo	–	–	279	–	–	76	24	179	14.80	11.50	69	209	1	54.36	7.68
printtokens	165	57	18	6.15	13.67	178	58	4	8.94	6.22	169	71	0	15.94	9.26
printtokens2	188	4	0	13.35	7.25	188	4	0	13.21	6.48	187	5	0	28.29	28.89
schedule	37	0	81	0.05	–	92	22	4	7.82	11.13	85	33	0	33.04	31.15
schedule2	33	0	74	0.04	–	42	0	65	7.32	–	35	55	17	189.03	205.14
cdaudio	544	179	50	0.41	0.81	–	190	583	–	6.36	566	207	0	1.50	1.58
diskperf	270	117	56	0.16	0.41	265	119	59	5.08	5.18	304	139	0	0.89	0.85
floppy	240	69	22	0.18	0.43	244	65	22	4.75	5.23	250	81	0	0.72	0.71
floppy2	497	82	27	0.33	0.59	501	79	26	5.28	5.68	511	95	0	1.51	1.35
kbfiltr	107	49	20	0.09	0.10	107	51	18	3.85	3.61	116	60	0	0.32	0.32
kbfiltr2	249	74	39	0.15	0.20	249	76	37	4.14	4.28	264	98	0	0.56	0.54
s3_srvr_1a	123	295	156	2.69	1.37	123	295	156	4.94	4.13	170	404	0	0.69	0.69
s3_srvr_1b	43	96	0	0.36	0.80	43	96	0	3.31	3.26	43	96	0	0.16	0.16
s3_clnt	625	969	83	14.62	4.86	661	1012	4	9.72	5.12	665	1012	0	39.62	41.52
s3_clnt_termination	540	964	91	15.16	4.35	582	1012	1	10.11	5.42	583	1012	0	22.57	24.02
s3_srvr_2	418	1034	678	3.50	5.21	698	1344	88	11.00	5.25	704	1420	6	102.85	128.09
s3_srvr_7	393	1073	794	3.34	4.78	712	1458	90	11.09	5.43	721	1538	1	100.42	124.45
s3_srvr_8	425	1183	714	3.98	5.07	701	1529	92	10.70	5.58	706	1604	12	107.31	137.57
s3_srvr_10	414	1060	726	5.00	32.16	678	1432	90	8.40	4.45	683	1517	0	125.92	111.44
s3_srvr_12	388	1611	1126	4.13	7.00	759	2231	135	9.86	6.19	758	2345	22	125.43	144.04
s3_srvr_13	431	1111	783	4.43	5.61	745	1500	80	10.04	4.55	737	1569	19	111.75	137.98
osek_control	607	150	170	9.43	8.09	645	199	87	7.72	6.54	623	277	27	52.76	65.12
space_control	1012	457	270	13.34	14.72	1156	495	88	9.85	10.57	1137	579	23	67.23	75.94
subway_control	1543	842	510	21.52	25.73	1793	1013	89	21.18	14.12	1787	1069	27	93.91	121.67
Total	9882	11648	6824	–	–	11750	14455	2153	–	–	12531	15656	155	–	–

7.5 Study 2

Table 4 gives the detailed performance statistics of the SMC-based reduction approach for data-flow testing, where "–" means the corresponding data does not apply or not available[10]. For each implementation instance, it shows the number of feasible (denoted by F), infeasible (denoted by I) and unknown (denoted by U) pairs, and the median of testing times on feasible and infeasible pairs (denoted by M_F and M_I, respectively). The last row gives the total number of feasible, infeasible, and unknown pairs. Note that BLAST and CPAchecker implement CEGAR-based model checking approach, thereby they can give the *feasible* or *infeasible* conclusion (or *unknown* due to undecidability of the problem) without any false positives. As for CBMC, it implements the bounded model checking technique, and in general cannot eliminate infeasible pairs as certain. Thus, the numbers of infeasible pairs identified by CBMC are only valid modulo the given checking bound. From the results, we can see CPAchecker and CBMC are more effective than BLAST in terms of feasible pairs as well as infeasible pairs. In detail, BLAST, CPAchecker and CBMC, respectively, cover 9882, 11750, 12531 feasible pairs, and identify 11648, 14455 and 15656 infeasible ones.

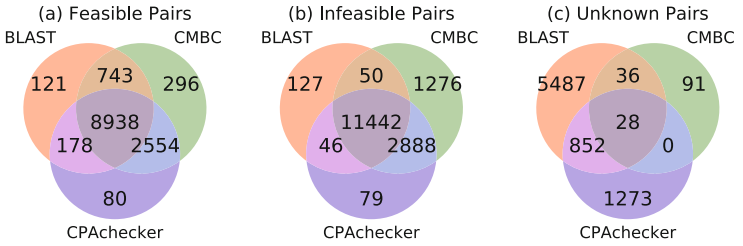


Fig. 7. Venn diagrams of (a) feasible, (b) infeasible and (c) unknown pairs concluded by the three model checkers BLAST, CPAchecker and CBMC for all subjects.

Figure 7 shows the venn diagrams of feasible, infeasible and unknown pairs concluded by the three model checkers BLAST, CPAchecker and CBMC. We can get several important observations: (1) The number of feasible and infeasible pairs identified by all the three model checkers accounts for the majority, occupying 69.2% and 71.9% pairs, respectively. It indicates both the CEGAR-based and BMC-based approaches are practical and can give consistent answers in most cases. (2) Although the infeasible pairs identified by the BMC-based approach are only valid modulo the given checking bound, we can see CBMC in fact correctly concludes a large portion of infeasible pairs. Compared with the infeasiblity results of CPAchecker, 91.8% (14,380/15,656) infeasible pairs identified by CBMC are indeed infeasible given appropriate checking bounds. Thus, the BMC-based approach can still serve as a heuristic-criterion to identify hard-to-cover (probably infeasible) pairs, and better prioritize testing efforts.

[10] BLAST hangs on *totinfo*, and CPAchecker crashes on parts of pairs from *cdaudio*.

(3) CPAchecker and CBMC have the largest number of overlapped pairs than the other combinations. They identify 94.7% feasible and 90.3% infeasible pairs, respectively. It indicates these two tools are more effective.

Answer to RQ2: *In summary, the SMC-based reduction approach is practical for data-flow testing. Both the CEGAR-based and BMC-based approaches can give consistent conclusions on the majority of def-use pairs. Specifically, the CEGAR-based approach can give answers for feasibility as certain, while the BMC-based approach can serve as a heuristic-criterion to identify hard-to-cover (probably infeasible) pairs when given appropriate checking bounds.*

7.6 Study 3

To investigate the effectiveness of our combined approach, we complement the SE-based approach with the SMC-based approach to do data-flow testing. Specifically, we realize this combined approach by interleaving these two approaches (the setting is specified in Sect. 7.3). Figure 8 shows the data-flow coverage achieved by KLEE, BLAST, CPAchecker, CBMC alone and their combinations (*e.g.*, the combination of KLEE and CPAchecker, denoted as KLEE+CPAchecker for short) on the 33 subjects within the same testing budget. We can see the combined approach can greatly improve data-flow coverage. In detail, KLEE only achieves 54.3% data-flow coverage on average for the 33 subjects, while KLEE+BLAST, KLEE+CPAchecker, and KLEE+CBMC, respectively, achieve 82.1%, 90.8%, and 99.5% coverage. Compared with KLEE, the combined approach instances, KLEE+BLAST, KLEE+CPAchecker, and KLEE+CBMC, respectively, improve the coverage by 27.8%, 36.5% and 45.2% on average. On the other hand, KLEE+BLAST improves coverage by 10% against BLAST alone, and KLEE+CPAchecker improves coverage by 7% against CPAchecker alone, respectively.

Figure 9 further shows the total testing time consumed by KLEE, BLAST, CPAchecker, CBMC and their combinations when achieving their peak coverage in Fig. 8. We can see that the combined approach can almost consistently reduce the total testing time on each subject. Specifically, compared with KLEE, the combined approach instances, KLEE+BLAST, KLEE+CPAchecker, and KLEE+CBMC, respectively, achieve faster data-flow testing in 30/33, 29/33, and 28/33 subjects, and reduce the total testing time by 78.8%, 93.6% and 20.1% on average in those subjects. Among the three instances of combined approach, KLEE+CPAchecker achieves the best performance, which reduces testing time by 93.6% for all the 33 subjects, and at the same time improves data-flow coverage by 36.5%. On the other hand, the combined approach instances, KLEE+BLAST and KLEE+CPAchecker, also reduce the total testing time of BLAST and CPAchecker by 23.8% and 19.9%, respectively.

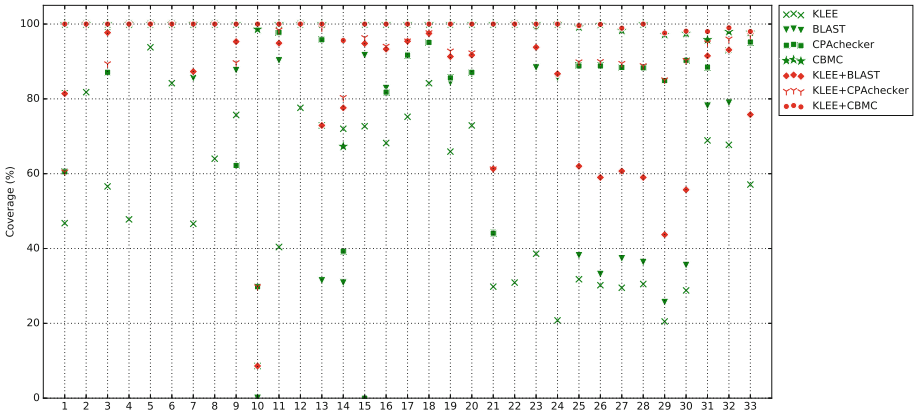


Fig. 8. Data-flow coverage achieved by KLEE, BLAST, CPAchecker, CBMC and their combinations (*i.e.*, KLEE+BLAST, KLEE+CPAchecker, KLEE+CBMC) within the same time budget. Each number on the X axis denotes the set of 33 subjects in our study. Note that the results of CBMC and KLEE+CBMC are only valid modulo the given checking bounds.

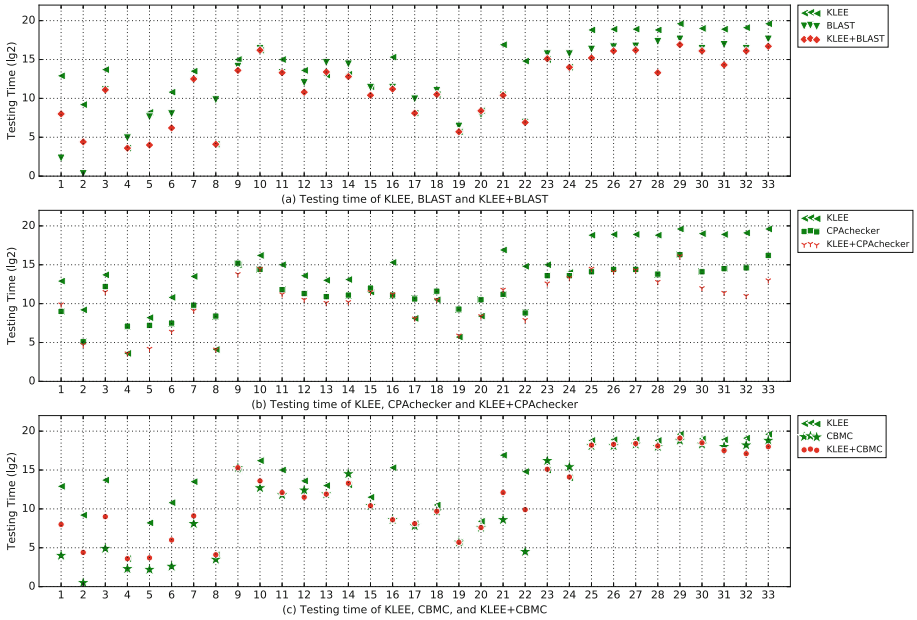


Fig. 9. Consumed time for data-flow testing by KLEE, BLAST, CPAchecker, CBMC and their combinations (*i.e.*, KLEE+BLAST, KLEE+CPAchecker, KLEE+CBMC) for reaching their respective highest coverage. Each point on the X axis denotes the set of 33 subjects in our study. Note that the Y axis uses a logarithmic scale.

Answer to RQ3: *In summary, the combined approach, which combines symbolic execution and software model checking, achieves more efficient data-flow testing. The model checking approach can weed out infeasible pairs that the symbolic execution approach cannot infer by 71.9%–97.2%. Compared with the SE-based approach alone, the combined approach can improve data-flow coverage by 27.8–45.2%. In particular, the instance KLEE+CPAchecker performs best, which reduces total testing time by 93.6% for all 33 subjects, and at the same time improves data-flow coverage by 36.5%. Compared with the CEGAR-based or BMC-based approach alone, the combined approach can also reduce testing time by 19.9–23.8%, and improve data-flow coverage by 7–10%.*

8 Conclusion

This paper introduces an efficient, combined data-flow testing approach. We designed a cut point guided search strategy to make symbolic execution practical; and devised a simple encoding of data-flow testing via software model checking. The two approaches offer complementary strengths: symbolic execution is more effective at covering feasible def-use pairs, while software model checking is more effective at rejecting infeasible pairs. Specifically, the CEGAR-based approach is used to eliminate infeasible pairs as certain, while the BMC-based approach can be used as a heuristic-criterion to identify hard-to-cover (probably infeasible) pairs when given appropriate checking bounds.

Acknowledgements. This work is in honor of Jifeng He's contribution to computer science, especially establishing the Unifying Theories of Programming (UTP). This work applies formal methods to support software testing, which was influenced by the work of Jifeng He. Ting Su, the lead author of this work, sincerely appreciate the academic guidance from his PhD supervisor Jifeng He.

References

1. Cyclo. http://www.gentoogeek.org/cyclo.html
2. ALDanial: cloc. GitHub (2018)
3. Alexander, R.T., Offutt, J., Stefik, A.: Testing coupling relationships in object-oriented programs. Softw. Test. Verif. Reliab. **20**(4), 291–327 (2010)
4. Ammann, P., Offutt, J.: Introduction to Software Testing, 1st edn. Cambridge University Press, New York (2008)
5. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, 16–18 January 2002, pp. 1–3 (2002)
6. Baluda, M., Braione, P., Denaro, G., Pezzè, M.: Structural coverage of feasible code. In: The 5th Workshop on Automation of Software Test, AST 2010, 3–4 May 2010, Cape Town, South Africa, pp. 59–66 (2010)

7. Baluda, M., Braione, P., Denaro, G., Pezzè, M.: Enhancing structural software coverage by incrementally computing branch executability. Software Qual. J. **19**(4), 725–751 (2011)
8. Baluda, M., Denaro, G., Pezzè, M.: Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Software Eng. **42**(5), 403–426 (2016)
9. Bardin, S., et al.: Sound and quasi-complete detection of infeasible test requirements. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, 13–17 April 2015, pp. 1–10 (2015)
10. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. IEEE Trans. Software Eng. **36**(4), 495–508 (2010)
11. Beyer, D.: Competition on software verification – (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_38
12. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 326–335. IEEE Computer Society, Washington, DC (2004)
13. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: applications to software engineering. Int. J. Softw. Tools Technol. Transf. **9**(5), 505–525 (2007)
14. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
15. Beyer, D., Lemberger, T.: Software verification: testing vs. model checking - a comparative evaluation of the state of the art. In: Strichman, O., Tzoref-Brill, R. (eds.) HVC 2017. LNCS, vol. 10629, pp. 99–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_7
16. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15–19 September 2008, L'Aquila, Italy, pp. 443–446 (2008)
17. Buy, U.A., Orso, A., Pezzè, M.: Automated testing of classes. In: ISSTA, pp. 39–48 (2000)
18. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symposium on Operating Systems Design and Implementation, pp. 209–224 (2008)
19. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, 30 October–3 November 2006, pp. 322–335 (2006)
20. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
21. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: Proceedings of the 25th International Conference on Software Engineering, 3–10 May 2003, Portland, Oregon, USA, pp. 385–395 (2003)
22. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. **30**(6), 388–402 (2004)
23. Chatterjee, B., Ryder, B.G.: Data-flow-based testing of object-oriented libraries. Technical report DCS-TR-382, Rutgers University (1999)

24. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
25. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, 29 March–2 April 2004, pp. 168–176 (2004)
26. Clarke, L.A.: A program testing system. In: Proceedings of the 1976 Annual Conference, Houston, Texas, USA, 20–22 October 1976, pp. 488–491 (1976)
27. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Trans. Software Eng. **15**(11), 1318–1332 (1989)
28. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven Concolic testing. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 328–347. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_16
29. Denaro, G., Margara, A., Pezzè, M., Vivanti, M.: Dynamic data flow testing of object oriented systems. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, 16–24 May 2015, vol. 1, pp. 947–958 (2015)
30. Denaro, G., Pezzè, M., Vivanti, M.: On the right objectives of data flow testing. In: IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, 31 March–4 April 2014, Cleveland, Ohio, USA, pp. 71–80 (2014)
31. Do, T., Fong, A.C.M., Pears, R.: Precise guidance to dynamic test generation. In: Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 5–12 (2012)
32. Eler, M.M., Endo, A.T., Durelli, V., Procópio-PR, C.: Covering user-defined data-flow test requirements using symbolic execution. In: Proceedings of the Thirteenth Brazilian Symposium On Software Quality (SBQS), pp. 16–30 (2014)
33. ETAPS: Competition on software verification (SV-COMP). ETAPS European Joint Conference on Theory & Practice of Software - TACAS 2017 (2017). https://sv-comp.sosy-lab.org/2017/
34. Foreman, L.M., Zweben, S.H.: A study of the effectiveness of control and data flow testing strategies. J. Syst. Softw. **21**(3), 215–228 (1993)
35. Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Trans. Softw. Eng. **19**(8), 774–787 (1993)
36. Frankl, P.G., Iakounenko, O.: Further empirical studies of test effectiveness. In: SIGSOFT 1998, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, 3–5 November 1998, pp. 153–162 (1998)
37. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test. Verification Reliab. **19**(3), 215–261 (2009)
38. Ghiduk, A.S.: A new software data-flow testing approach via ant colony algorithms. Univ. J. Comput. Sci. Eng. Technol. **1**(1), 64–72 (2010)
39. Ghiduk, A.S., Harrold, M.J., Girgis, M.R.: Using genetic algorithms to aid test-data generation for data-flow coverage. In: APSEC, pp. 41–48 (2007)
40. Girgis, M.R.: Using symbolic execution and data flow criteria to aid test data selection. Softw. Test. Verif. Reliab. **3**(2), 101–112 (1993)

41. Girgis, M.R.: Automatic test data generation for data flow testing using a genetic algorithm. J. UCS **11**(6), 898–915 (2005)
42. Girgis, M.R., Ghiduk, A.S., Abd-elkawy, E.H.: Automatic generation of data flow test paths using a genetic algorithm. Int. J. Comput. Appl. **89**(12), 29–36 (2014)
43. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223. ACM, New York (2005)
44. Goldberg, A., Wang, T., Zimmerman, D.: Applications of feasible path analysis to program testing. In: Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, 17–19 August 1994, pp. 80–94 (1994)
45. Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo, S.: Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, 7–9 April 2010, Workshops Proceedings, pp. 182–191 (2010)
46. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: SIGSOFT FSE, pp. 154–163 (1994)
47. Harrold, M.J., Soffa, M.L.: Efficient computation of interprocedural definition-use chains. ACM Trans. Program. Lang. Syst. **16**(2), 175–204 (1994)
48. Hassan, M.M., Andrews, J.H.: Comparing multi-point stride coverage and dataflow coverage. In: 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 172–181 (2013)
49. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, 16–18 January 2002, pp. 58–70 (2002)
50. Hierons, R.M., et al.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2), 1–76 (2009)
51. Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: Proceedings of the 25th International Conference on Software Engineering, 3–10 May 2003, Portland, Oregon, USA, pp. 232–243 (2003)
52. Horgan, J.R., London, S.: ATAC: a data flow coverage testing tool for C. In: Proceedings of Symposium on Assessment of Quality Software Development Tools, pp. 2–10 (1992)
53. Horgan, J.R., London, S.: Data flow coverage and the C language. In: Proceedings of the Symposium on Testing, Analysis, and Verification, pp. 87–97. TAV4, ACM, New York (1991)
54. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.J.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: ICSE, pp. 191–200 (1994)
55. Jamrozik, K., Fraser, G., Tillmann, N., de Halleux, J.: Augmented dynamic symbolic execution. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 254–257 (2012)
56. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. (CSUR) **41**(4), 21 (2009)
57. Khamis, A., Bahgat, R., Abdelaziz, R.: Automatic test data generation using data flow information. Dogus Univ. J. **2**, 140–153 (2011)
58. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)

59. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
60. Lakhotia, K., McMinn, P., Harman, M.: Automated test data generation for coverage: haven't we solved this problem yet? In: Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, pp. 95–104. IEEE Computer Society, Washington (2009)
61. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_11
62. Malevris, N., Yates, D.: The collateral coverage of data flow criteria when branch testing. Inf. Softw. Technol. **48**(8), 676–686 (2006)
63. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: 40th International Conference on Software Engineering, 27 May–3 June 2018, Gothenburg, Sweden (2018)
64. Marinescu, P.D., Cadar, C.: KATCH: high-coverage testing of software patches. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, 18–26 August 2013, pp. 235–245 (2013)
65. Marré, M., Bertolino, A.: Unconstrained duas and their use in achieving all-uses coverage. In: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 147–157. ISSTA 199. ACM, New York (1996)
66. Marré, M., Bertolino, A.: Using spanning sets for coverage testing. IEEE Trans. Softw. Eng. **29**(11), 974–984 (2003)
67. Mathur, A.P., Wong, W.E.: An empirical comparison of data flow and mutation-based test adequacy criteria. Softw. Test. Verif. Reliab. **4**(1), 9–31 (1994)
68. Merlo, E., Antoniol, G.: A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In: CASCON, p. 7 (1999)
69. Nayak, N., Mohapatra, D.P.: Automatic test data generation for data flow testing using particle swarm optimization. In: IC3 (2), pp. 1–12 (2010)
70. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_16
71. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. **7**(3), 165–192 (1997)
72. Pande, H.D., Landi, W.A., Ryder, B.G.: Interprocedural def-use associations for C systems with single level pointers. IEEE Trans. Softw. Eng. **20**(5), 385–403 (1994)
73. Pandita, R., Xie, T., Tillmann, N., de Halleux, J.: Guided test generation for coverage criteria. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, pp. 1–10. IEEE Computer Society, Washington (2010)
74. Peng, Y., Huang, Y., Su, T., Guo, J.: Modeling and verification of AUTOSAR OS and EMS application. In: Seventh International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, 1–3 July 2013, Birmingham, UK, pp. 37–44 (2013)
75. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: Proceedings of the 6th International Conference on Software Engineering, ICSE 1982, pp. 272–278. IEEE Computer Society Press, Los Alamitos (1982)

76. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Software Eng. **11**(4), 367–375 (1985)
77. Santelices, R., Harrold, M.J.: Efficiently monitoring data-flow test coverage. In: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, pp. 343–352. ACM, New York (2007)
78. Santelices, R.A., Sinha, S., Harrold, M.J.: Subsumption of program entities for efficient coverage and monitoring. In: Third International Workshop on Software Quality Assurance, SOQUA 2006, Portland, Oregon, USA, 6 November 2006, pp. 2–5 (2006)
79. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 263–272. ACM, New York (2005)
80. Singla, S., Kumar, D., Rai, H.M., Singla, P.: A hybrid PSO approach to automate test data generation for data flow coverage with dominance concepts. J. Adv. Sci. Technol. **37**, 15–26 (2011)
81. Singla, S., Singla, P., Rai, H.M.: An automatic test data generation for data flow coverage using soft computing approach. IJRRCS **2**(2), 265–270 (2011)
82. SIR Project: Software-artifact infrastructure repository. NC State University. http://sir.unl.edu/php/previewfiles.php. Accessed July 2016
83. Su, T.: A bibliography of papers and tools on data flow testing. GitHub (2017). https://tingsu.github.io/files/dftbib.html
84. Su, T., Fu, Z., Pu, G., He, J., Su, Z.: Combining symbolic execution and model checking for data flow testing. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, 16–24 May 2015, vol. 1, pp. 654–665 (2015)
85. Su, T., et al.: Automated coverage-driven test data generation using dynamic symbolic execution. In: Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, 30 June–2 July 2014, pp. 98–107 (2014)
86. Su, T., et al.: A survey on data-flow testing. ACM Comput. Surv. **50**(1), 5:1–5:35 (2017)
87. Su, T., Zhang, C., Yan, Y., Su, Z.: Towards efficient data-flow test data generation. GitHub (2019). https://tingsu.github.io/files/hybrid_dft.html
88. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
89. Vivanti, M., Mis, A., Gorla, A., Fraser, G.: Search-based data-flow test generation. In: IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, 4–7 November 2013, pp. 370–379 (2013)
90. Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., Yang, Z.: Dependence guided symbolic execution. IEEE Trans. Software Eng. **43**(3), 252–271 (2017)
91. Wang, Z., Yu, X., Sun, T., Pu, G., Ding, Z., Hu, J.: Test data generation for derived types in C program. In: TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29–31 July 2009, Tianjin, China, pp. 155–162 (2009)
92. Weyuker, E.J.: The complexity of data flow criteria for test data selection. Inf. Process. Lett. **19**(2), 103–109 (1984)
93. Weyuker, E.J.: More experience with data flow testing. IEEE Trans. Software Eng. **19**(9), 912–919 (1993)

94. Wong, W.E., Mathur, A.P.: Fault detection effectiveness of mutation and data flow testing. Software Qual. J. **4**(1), 69–83 (1995)
95. Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 359–368 (2009)
96. Yang, Q., Li, J.J., Weiss, D.M.: A survey of coverage-based testing tools. Comput. J. **52**(5), 589–597 (2009)
97. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: European Conference on Computer Systems, Proceedings of the 5th European Conference on Computer Systems, EuroSys 2010, Paris, France, 13–16 April 2010, pp. 321–334 (2010)
98. Zhang, C., et al.: SmartUnit: empirical evaluations for automated unit testing of embedded software in industry. In: 40th IEEE/ACM International Conference on Software Engineering, Software Engineering in Practice Track, ICSE 2018, 27 May–3 June 2018, Gothenburg, Sweden (2018)
99. Zhang, L., Xie, T., Zhang, L., Tillmann, N., de Halleux, J., Mei, H.: Test generation via dynamic symbolic execution for mutation testing. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), 12–18 September 2010, Timisoara, Romania, pp. 1–10 (2010)
100. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997)