# A Survey on Data Flow Testing

Ting Su* Ke Wu*
*Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
Email: tsuletgo@gmail.com, sei_wk2009@126.com

***Abstract*—Data flow testing (DFT) focuses on the flow of data through a program, *i.e.*, the interactions between variable definitions and their uses. The motivation behind is to verify the correctness of defined variable values by observing their corresponding uses produce the desired results. The original conception of DFT was introduced by Herman in 1976. Since then, a set of data flow-based coverage criteria have been proposed. Researchers have also conducted a number of studies to analyze its complexities and effectiveness from both theoretical and empirical perspectives. The conclusion that DFT is more effective in fault detection than control flow-based testing techniques (*e.g.*, statement or branch testing) stimulates various approaches to pursue efficient and automated DFT.**

**This survey presents an overview of data flow testing. It summarizes some challenges behind DFT, and systematically investigates various general approaches developed to automate it, especially in test data generation. The approaches presented include: random testing, collateral coverage-based testing, search-based testing, symbolic execution-based testing and model checking-based testing. A number of techniques used to track data flow coverage are also discussed. Moreover, an overview of DFT applications are presented (including software fault localization, web security testing, and specification consistency checking), as well as some recent research directions and advancements in this field. The paper concludes with a discussion on the possibilities of combining different approaches to build a more efficient and practical DFT framework.**

## I. Overview

*Data flow testing* (DFT) is a family of testing strategies based on selecting paths from the program's control flow in order to explore sequences of events related to the status of variables or data objects. It fills the gaps between all path testing and branch/statement testing with the aim to pinpoint the potential data-flow anomalies.

In contrast to control flow-based testing (*e.g.*, statement or branch testing), data flow testing focuses on the flow of data through a program, *i.e.*, the interactions between variable definitions and their uses. In particular, a variable value (generated by a *definition*) usually propagates along a control flow path to participate in some following computations (a *use* of this variable). Such the relation between the definition and its use of a variable is referred as a *def-use pair* (or, *def-use association*, generally called as a *test objective* in white-box testing). The aim of (dynamic) data flow testing is to find concrete program execution paths to exercise all these kinds of def-use pairs in the program with different strategies (*i.e.*, data flow-based *coverage criteria*). The motivation behind is to verify the correctness of defined variable values by observing their corresponding uses can produce the desired results.

The original conception of data flow testing was introduced by Herman [1] in 1976. He claims that data flow testing can test a program more thoroughly and reveal more subtle software bugs. After that various slightly different notions of data flow-based coverage criteria [2]–[6] have been proposed and investigated. The main reason for this diversity is that there are different ways of exercising definition-use relations as well as various adaptions in procedural and object-oriented programming languages. The effectiveness of DFT is later justified by several empirical studies [7]–[12], which have showed data flow-based coverage criteria outclass control flow-based criteria (*e.g.*, statement or branch coverage). Moreover, in an online software testing knowledge center organized by Khannur [12], it is reported that "the number of bugs detected by putting the criteria of 90% data coverage were capable to find defects those were twice as high as those detected by 90% branch coverage criteria" in practice.

In the past decades, data flow testing has been increasingly and widely studied (illustrated in Figure 1). There has been much research work on various testing techniques seeking to make this testing approach more practical. However, there is little survey work in the literature on data flow testing. Edvardsson [13] presents an early survey on automated test generation and identifies several challenges in that field. Anand *et al.* [14] recently give an orchestrated survey on the most prominent techniques for automated test data generation. But none of these techniques are discussed in the context of data flow testing. An introductory chapter of data flow testing can be found in the book by Beizer [15] as well as in the book by Pezzè *et al.* [16]. They introduce the basic conceptions and difficulties in enforcing DFT but have not discussed how to automate it. Considering the benefits from data flow testing, we believe that for both academic researchers and industrial practitioners it is highly desirable to review the existing techniques, recognize the difficulties, and gain perspectives for future research directions in the field of data flow testing. So, we present this first survey on data flow testing: we constructed a data flow testing publication repository, which includes total 89 papers from 1976∼2015. We searched the online repositories of the main technical publishers, and collected valid papers which have these keywords "data flow testing", "def-use pairs", "data-flow relations", "data flow testing + analysis", "data flow testing + test generation", "data flow testing + coverage" in either their titles or abstracts. Then we went through each reference of these papers to find the missing publications using the same

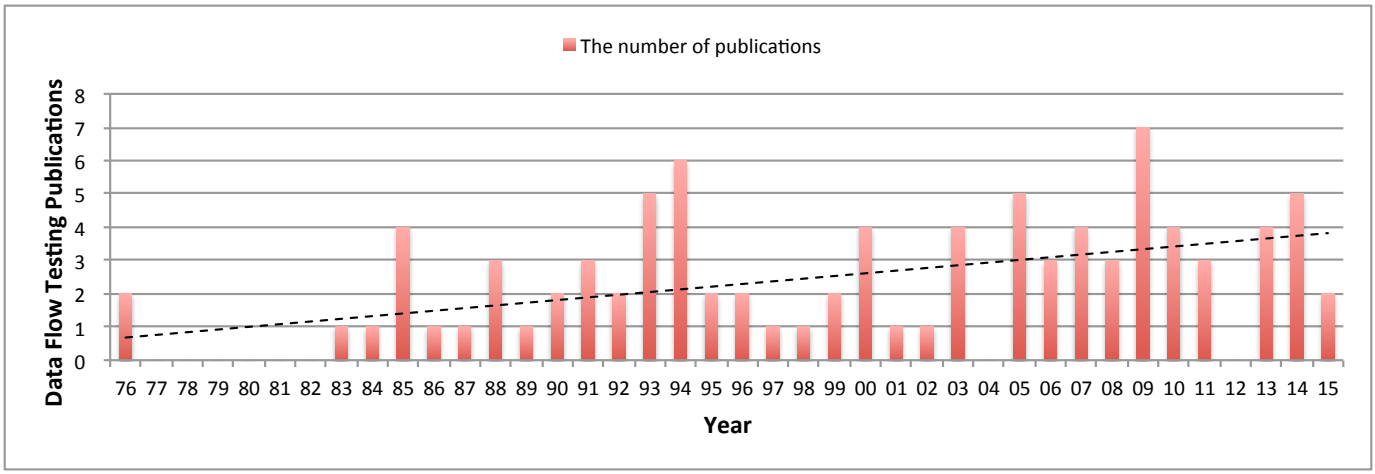



Fig. 1. Data flow testing publications from 1976∼2015.

keyword rules. The repository is now available online[1].

The process of data flow testing consists of three basic steps: (1) data flow analysis (identify def-use pairs), (2) generate test data, (3) track data flow coverage. Here, in this survey, we focus on the latter two steps, *i.e.*, various general approaches to automating test data generation and techniques used to track data flow coverage. We do not plan to investigate data flow analysis, since it is an independent research topic from DFT and has been investigated in [17], [18]. Generally, data flow testing can be performed at two conceptual levels, i.e., *static* and *dynamic* data flow testing. The former is used to identify potential data-flow anomalies [19] without executing source codes, while the latter requires actual program executions. Here, we focus on investigating testing approaches in the later context, *i.e.*, dynamic data flow testing. Without ambiguity, we simply use *data flow testing* to denote *dynamic data flow testing* throughout the paper.

***Organization of this survey*** The reminder of this survey is organized as follows. Section II introduces some fundamental conceptions in DFT and gives an illustrative example. The traditional testing process of DFT (*i.e.*, static data-flow analysis, test data generation and coverage tracking) is also presented. Some known difficulties in enforcing data flow testing are summarized, which explains why DFT is not as popular as control flow testing in practice. Section III investigates various general approaches developed to automate data flow-based test data generation that can be found in the present literature, which include random testing, collateral coverage-based testing, search-based testing, symbolic execution-based testing, and model checking-based testing. Their strengths and weaknesses are discussed as well as the directions of future efforts. Section IV surveys the approaches of coverage tracking in DFT. Section V surveys the applications of DFT. Recent research directions and advancements are presented in Section VI, and the possibilities of combining different approaches to build a more efficient and practical DFT framework are discussed in Section VII. Section VIII concludes this survey.

## II. Data Flow Testing

This section introduces some fundamental conceptions in data flow testing. It then discusses the basic process of data flow testing and the difficulties from which it suffers.

### A. Fundamental Conceptions

A program *path* can be denoted as a sequence of control points[2], written in the form $l_1, l_2, \ldots, l_n$. We distinguish two kinds of paths. A *control flow path* is a sequence of control points along the control flow graph of a program; an *execution path* is a sequence of executed control points driven by the program input.

Following the classic definition from Herman [1], a *def-use pair* $du(l_d, l_u, x)$ occurs when there exists at least one control flow path from the assignment (*i.e. definition*, or *def* in short) of variable $x$ at control point $l_d$ to the statement at control point $l_u$ where the same variable $x$ is used (*i.e. use*) on which no redefinitions of $x$ appear (*i.e.* the path from the *def* to the *use* is *def-clear*). In particular, data flow testing adequacy distinguishes two types of uses. When the variable is used in a computational or output statement, its use is referred as a *computation use* (or *c-use*). When the variable is used in a predicate, its use is called as a *predicate use* (or *p-use*).

*Definition 1 (Data Flow Testing):* Given a def-use pair $du(l_d, l_u, x)$ in program $P$, the goal of data flow testing is to find an input $t$ that induces an execution path passing through $l_d$ and then $l_u$ with no intermediate redefinitions (*i.e.*, *kills*) of $x$ between $l_d$ and $l_u$. We say this test case $t$ *satisfies* the pair $du$. The requirement to cover all def-use pairs at least once is called *all def-use coverage criterion*, which means at least one def-clear path of each pair should be covered.

[1]tingsu.github.io/files/dftbib.html

[2]Here, we use line numbers to denote control points in a program.

```
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 }
```

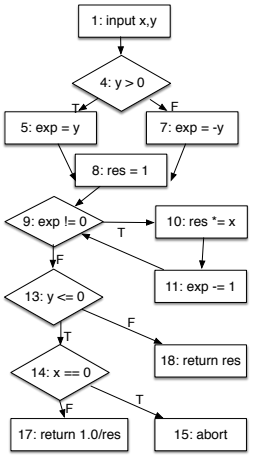



Fig. 2. An example: *power*.

### B. An Illustrative Example

Figure 2 shows an example program *power*, which takes as input two integers $x$ and $y$ and outputs $x^y$. Its control flow graph (CFG) is shown in the right column in Figure 2. Figure 3 shows the definitions and uses of the variables in *power*, and the corresponding def-use pairs. We can see this example program has total 19 statements, 8 branches and 15 def-use pairs.

For example, the followings are two def-use pairs *w.r.t.* the variable $res$:

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

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

Here $du_1$ is a def-use pair because the definition *w.r.t.* the variable $res$ (at Line 8) can reach the corresponding use (at Line 17) through the control flow path $l_8, l_9, l_{13}, 1_{14}, l_{17}$. It is a feasible pair as well because a test input can be found to satisfy $du_1$. For example, $t = (x \mapsto 1, y \mapsto 0)$ can induce an execution path $p = l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{16}, l_{17}$, which covers $du_1$ (*cf.* Definition 1). For $du_2$, it is a def-use pair because its definition (at Line 8) can reach the corresponding use (at Line 18) through the path $l_8, l_9, l_{13}, 1_{18}$. However it is an *infeasible* pair: if there were a test input that could reach the *use*, it must satisfy $y>0$ at $l_{13}$. Since $y$ has not been modified in the code, $y>0$ also holds at $l_4$. As a result, $res$ will be redefined at $l_{10}$ since the loop guard at $l_9$ is *true*. Clearly, no such a path exists for this pair which can both avoid redefinitions in the loop and reach the *use*.

### C. Data Flow Testing

A lot of work has been done to tackle the problem of data flow testing. We classified the main research topics in this field and compute their respective percentages in the literature (showed in Figure 4). We find that researchers have made great efforts in these two directions: test data generation and empirical analysis. It is not surprising since

| Line | Def | C-use | P-use | du |
|---|---|---|---|---|
| $l_1$ | $x, y$ | | | |
| $l_4$ | | | $y$ | $(l_1, l_4, y)$ |
| $l_5$ | $exp$ | $y$ | | $(l_1, l_5, y)$ |
| $l_7$ | $exp$ | $y$ | | $(l_1, l_7, y)$ |
| $l_8$ | $res$ | | | |
| $l_9$ | | | $exp$ | $(l_5, l_9, exp)$, $(l_7, l_9, exp)$ |
| $l_{10}$ | $res$ | $res, x$ | | $(l_8, l_{10}, res)$, $(l_1, l_{10}, x)$ |
| $l_{11}$ | $exp$ | $exp$ | | $(l_5, l_{11}, exp)$, $(l_7, l_{11}, exp)$ |
| $l_{13}$ | | | $y$ | $(l_1, l_{13}, y)$ |
| $l_{14}$ | | | $x$ | $(l_1, l_{14}, x)$ |
| $l_{17}$ | | $res$ | | $(l_8, l_{17}, res)$, $(l_{10}, l_{17}, res)$ |
| $l_{18}$ | | $res$ | | $(l_8, l_{18}, res)$, $(l_{10}, l_{18}, res)$ |

Fig. 3. The definitions and uses of variables in Figure 2, and their corresponding def-use pairs.

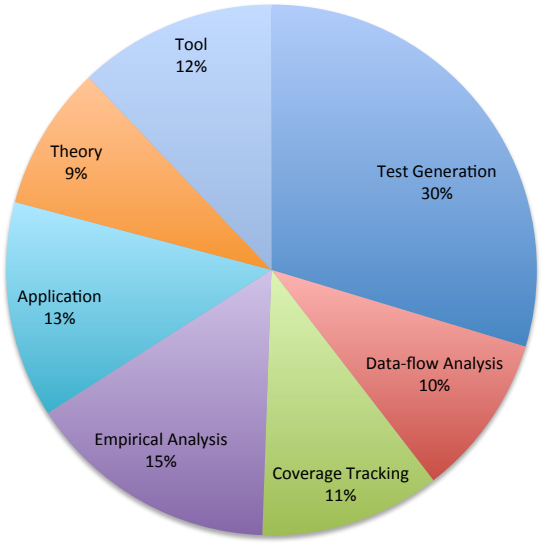



Fig. 4. Percentage of each research topic in the literation of data flow testing.

test data generation is the most time-consuming and important activity in software testing which demands high automation while empirical analysis is a common approach to analyze the effectiveness and complexities of a new testing strategy. In addition, data-flow analysis, coverage tracking and applications also attract wide research interests.

Among these topics, data-flow analysis, test data generation and coverage reporting are three basic testing phases in the traditional process of data flow testing (illustrated in Figure 5), which totally occupies more than 50% research efforts.

In the data-flow analysis phase, a data-flow analysis algorithm takes as input the program $P$ under test to compute test objectives (*i.e.*, def-use pairs). Here, a classic reaching definition algorithm [20] can be used. In the literature, Harrold *et al.* [21] use a standard iterative data-flow analysis to compute definition-use relations. Pande *et al.* [22] use an extended reaching definition analysis [23] to handle programs with single-level pointers. Harrold *et al.* [6] extend data-flow analysis for object-oriented languages, which considers the definition-use relations through instance variables when the public methods of a class are invoked in arbitrary order. Chatterjee *et al.* [24]

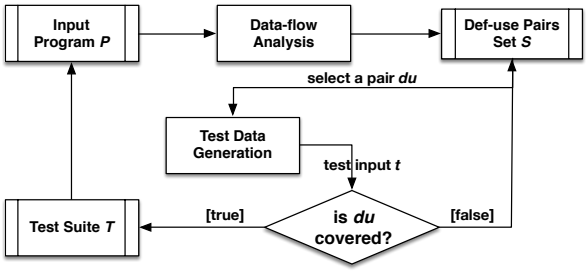



Fig. 5. The traditional process of data flow testing

use a flow- and context-sensitive algorithm to compute def-use pairs for object-oriented libraries. Souter *et al.* [25] and Denaro *et al.* [26] also extend classic data-flow analysis to object-oriented programs.

In the test data generation phase, a testing approach is adopted to generate a test input $t$ to satisfy a target def-use pair $du$. The task is to find an execution path which can cover the pair through a def-clear path.

In the coverage tracking phase, the test input $t$ is executed against the program $P$ and decide whether the pair $du$ is covered or not. If the $du$ is covered, $t$ is incorporated into the resulting test suite $T$. The testing process continues until all pairs are satisfied or the testing budgets (*e.g.*, testing time) are used up. At last, the resulting test suite $T$ will be replayed against the program $P$ to check correctness with the help of test oracles.

### D. The Difficulties of Data Flow Testing

Despite the effectiveness of DFT in fault detection, several difficulties prevent it from finding wide application in industrial practice. These difficulties originate from its testing process, and make it less practical than control flow-based testing.

***Unscalable Data-flow Analysis*** A data-flow analysis algorithm is demanded in DFT to identify def-use pairs from the program under test. However, it is not easy for a data-flow analysis procedure to be scalable against large real-world programs, especially when all program features are taken into consideration (*e.g.*, *aliases*, *arrays, structs* and *class objects*). A suitable approximation has to be made to trade off between precision and scalability. Continuous efforts are endeavored in this static analysis field. In contrast, identifying statements or branches in control flow testing is much easier.

***Path Explosion*** DFT imposes constraints on paths instead of program structs as in control flow testing, *i.e.*, it requires finding the *def-clear* execution paths to exercise the definition-use relations. However, real-world programs usually has a large path space (which is also known as the *path-explosion problem*). It is challenging, in reasonable time, to find one or some execution paths from the whole path space to satisfy a pair.

***Infeasible Test Objectives*** Due to the conservativeness of data-flow analysis, test objectives consists of *feasible* pairs as well as *infeasible* ones (*e.g.*, the pair $(l_8, l_{18}, res)$ in Section II-B is infeasible). 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, a testing approach may spend a large amount of time, in vain, to cover an infeasible def-use pair.

***Large Test Objectives*** The number of test objectives *w.r.t.* data-flow coverage criteria is much larger than that of control-flow criteria, as a result, more testing efforts are required in the process of test case generation, test oracle checking, and coverage tracking. For example, when a tester need to derive test cases *w.r.t.* data flow criteria: she has to write a test case to cover a variable definition and its corresponding use without variable redefinitions, which is more difficult than just cover a statement or branch.

Among these difficulties, the problems of path explosion and infeasible test objectives are not unique in DFT but also exist in other structural testings. But they are exacerbated by the large set of test objectives in DFT. Although it is impossible to completely solve these problems due to their undecidability, with existing advancements in DFT, the process of data flow testing can be automated and the testing cost can be mitigated, as this survey will show.

## III. Approaches for Test Data Generation

This section presents various approaches to automating data flow-based test data generation that can be found in the present literature. We classify these approaches into five groups and respectively explain them in independent sections. They include random testing (Section III-A), collateral coverage-based testing (Section III-B), search-based testing (Section III-C), symbolic execution-based testing (Section III-D), and model checking-based testing (Section III-E) and some other approaches (Section III-F).

We compute the percentage[3] of each testing approach used in the literature (showed in Figure 6). We find that the search-based testing approach (including genetic algorithm and optimization algorithm) is the most widely studied approach, which occupies 50%. The collateral coverage-based approach and random testing are also popular. But more sophisticated testing approaches, *e.g.*, symbolic execution and model checking are less investigated.

### A. Random Testing

Random testing [27] is one of most widely-used and cost-effective testing approach. In the classic implementation, test inputs are randomly picked from valid ranges *w.r.t.* program specifications and later executed against the program under test. In object-oriented systems, a test case is a sequence of methods and constructor invocations [28], and random testing is adapted to randomly generate these sequences as tests for the classes under test. Several work [29]–[32] has adopted this random testing technique as an easily-implemented but efficient approach for data flow testing.

[3]If some paper use more than one testing approaches, we count all of them in.

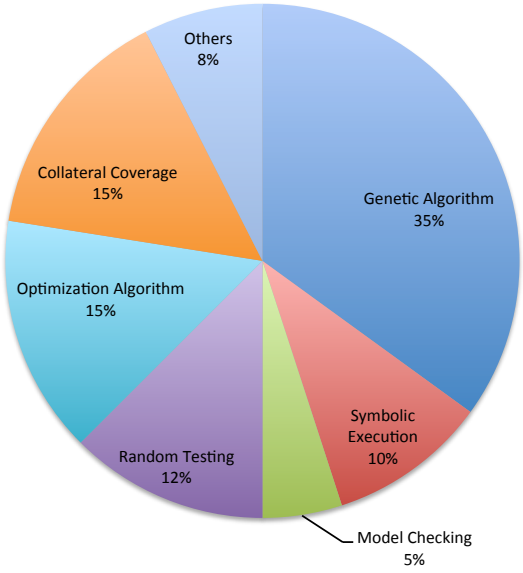



Fig. 6. Percentage of publications using each testing approach on data flow-based test data generation.

***Discussion*** Despite its easy implementation and cost-effectiveness, random testing could only distinguish limited set of program behaviors. High data flow testing coverage is thus usually out of its reach. But some modern techniques (*e.g.*, adaptive random testing [33], [34] and feedback-directed random testing [28]) have been developed to enhance this approach.

### *B. Collateral Coverage-based Approach*

In software testing, *collateral coverage* has been exploited to optimize test suite generation [35], [36], which is based on such an observation: the test case that satisfies a target test objective can also "accidentally" cover other test objectives. Thus, if these covered test objectives are excluded and invest the testing budgets on the remaining uncovered objectives, the resulting test suite size can be reduced as well as the cost of test case execution and test oracle checking.

Similarly, when exercising a program in an attempt to satisfy a given testing criterion (*e.g.*, branch coverage), there will also be a concurrent accrual of coverage *w.r.t.* other coverage criteria (*e.g.*, data-flow coverage), which is also a form of collateral coverage [37]. The reason is that one test objective may imply (or subsume) another one even when these two test objectives are from different testing criteria. For example, Figure 7 shows the subsumption relations between different testing coverage criteria. The criterion at the tail of an arrow subsumes the criterion at the head (*e.g.* the branch criterion subsumes the statement criterion). Since the subsumption relation is transitive, it actually defines the relations between various coverage criteria. Here, the shadowed criteria are seven types of data flow testing criteria (refer to [3], [5] for their detailed definitions), which emphasize different ways to exercise definition-use

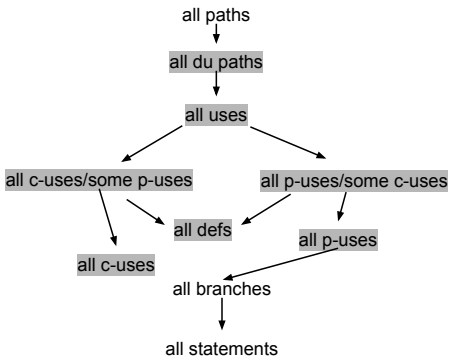



Fig. 7. Hierarchy of structural characteristics of a program.

relations. Among them, all-uses coverage is all def-use coverage (*cf.* Definition 1 in Section II-A).

The collateral coverage-based idea has been attempted to tackle data flow testing [37]–[42]. Malevris *et al.* [37] investigate the level of collateral coverage of data flow criteria when branch testing is intended. In the empirical study, they select paths from the control flow graph to fulfill all branches coverage on 59 units written in different programming languages (including Fortran, Pascal, C and Java), and measure the concurrently achieved data flow coverage *w.r.t.* seven data flow criteria (the ones shadowed in Figure 7). The study reveals that the actual coverage of each data flow criteria can be predicated as a function of the number of selected paths *w.r.t.* branch coverage testing and the number of actually feasible paths therein. Moreover, they also find: First, the data flow coverage appears to be independent of the language in which a unit is encoded. Second, the level of collateral coverage can be predicated a prior in estimating the possible testing budgets demanded by DFT. Third, it could be more cost effective to deploy branch testing before data flow testing since parts of data flow-based test objectives can be covered when branch testing.

Inspired by the idea of collateral coverage, Santelices *et al.* [38] propose an approach to automatically infer data flow coverage from branch coverage. In the static analysis phase, an inferability analysis is used to classify def-use pairs into three categories, *i.e.*, *inferable* (the coverage can always be inferred from branch coverage), *conditionally inferable* (the coverage can be inferred from branch coverage in some program executions, but not in all executions), and *non-inferable* (the coverage can never be determined with certainty from branch coverage). Later, during dynamic test suite execution, the branch coverage is recorded against three types of entities, *i.e.*, the definitions, the uses and the kills of def-use pairs. At last, the coverage reporting phase takes as input the results from both the static and dynamic analysis, it reports def-use pairs as *definitely covered*, *possibly covered*, or *not covered*. Although this approach may lose some coverage precision, the prominent benefit is that the overhead of runtime coverage tracking can be greatly mitigated since the program is instrumented at branch

coverage level instead of data flow coverage level.

Merlo *et al.* [39] exploit the coverage implication between def-use pairs and nodes (*i.e.*, statements) to achieve intra-procedural data flow testing. Pre- and post-dominator analysis is used to identify a set of nodes, whose coverage could imply the coverage of a subset of def-use pairs. The effectiveness of this technique was evaluated on a 16KLOC *Gnu find* tool.

Marré *et al.* [40], [41] propose an approach to identify a *minimal* set of def-use pairs such that the paths covering these pairs could cover all the pairs in the program. In other words, the coverage of the pairs outside the set can be inferred by the coverage of those pairs in this set. This set is called a *spanning set* and the pairs therein are called *unconstrained pairs*. The spanning set can help reduce the DFT cost (requiring fewer test cases) and the cardinality of this set can be used to estimate the testing cost as well.

Santelices *et al.* [42] present a subsumption algorithm for program entities of any type (*e.g.*, branches, def-use pairs, and call sequences) based on predicate conditions. A predicate condition is actually a simplified version of a path condition [43]. The system dependence graph is used to compute the predicate condition of an entity. Then, these conditions are used to construct a table representing the conditions for each entity that are necessary but may not sufficient for its coverage. At last, the algorithm uses this table and sufficiency information to create the subsumption hierarchy of entities for efficient coverage and tracking.

***Discussion*** The collateral coverage-based approach has several merits for data flow testing: optimize the test suite size as well as mitigate the coverage tracking overhead; estimate testing budgets; benefit the understanding of the relationships among entities at different levels of abstraction. However, this approach aims at using the coverage of low level program entities (*e.g.*, statements or branches) to infer the coverage of def-use pairs, it may fail to cover those pairs which can not be inferred.

### C. Search-based Testing Approach

In recent years, the adoption of the search-based approach [44] for automatic testing has been a growing interest for many researchers. This approach has been applied in functional testing (*e.g.*, structural testing) as well as non-functional properties checking (*e.g.*, worst-case execution-time analysis).

***Principal of Search-based Approach*** The search-based approach includes various metaheuristic search techniques [44], *e.g.*, hill climbing, simulated annealing, evolutionary algorithms. In principal, these techniques are high-level frameworks which utilize heuristics to find solutions to combinational problems at a reasonable computation cost. The problem of test data generation in general is undecidable, but it can be interpreted as a search problem in which it searches for desired values from program input domains to fulfill test requirements (*e.g.*, covering statements, branches or conditions). Various search techniques [45]–[48] have been applied in control flow-based testing (*e.g.*, branch and MC/DC testing [49], [50]).

At a high level, to adapt a metaheursitic search technique to a target problem, three basic decisions should be made. First, the problem solutions should be *encoded* in a suitable form so that they can be efficiently manipulated during the search. A good encoding method will ensure similar solutions in the unencoded space are also "neighbors" in the encoded space. Second, a *fitness or object function* should be set up which represents the aim of the target problem and supports the guidance of the search. It evaluates the effectiveness of individual solutions and thus has a critical impact on the successful application of the search technique. At last, a search procedure should be determined which formalizes the whole search algorithm.

***Genetic Algorithm-based Search*** The Genetic Algorithm (GA) [51] proposed in 1975 is a representative of metaheuristic search techniques, which is inspired by genetics and natural selection. In the context of test data generation, a GA starts from a population of candidate individuals (*i.e.*, test cases) and then use search operators (*e.g.*, selection, crossover and mutation) to generate the next promising test case. Selection chooses effective individuals from the population to do recombination (*i.e.*, crossover and mutation). Crossover between two individuals produces two offspring which share some genetic material from both parents while mutation introduces slight changes to a small proportion of the populations. During test data generation process, these three operations will be continuously used in the GA until some desired test cases are found within constrained testing budgets.

To tackle the data flow testing problem, several GA-based testing methods have been proposed in recent years [29], [30], [52]. Girgis [29] first uses the GA for data flow testing *w.r.t.* all-uses coverage on Fortran programs. In this work, it uses the encoding method proposed by Michalewicz [53] to encode the program test input (*i.e.*, test case). In [29], the GA uses the ration between the number of the covered def-use paths by a test case and the total number of def-use paths as the fitness function. In essence, this fitness function exclusively uses coverage information to determine the effectiveness of an individual test data. This GA-based method works as follows: Initially, it generates a set of test cases encoded in the binary string form. It then uses two methods (a random selection and a roulette wheel algorithm [53]-based selection) respectively to pick effective individuals for recombination (*i.e.*, crossover and mutation). After a predefined count of iterations, the GA will output a set of desired test cases which has covered the def-use paths and the remaining uncovered def-use pairs.

Later, Ghiduk *et al.* [30] find there are some pitfalls inside the fitness function in [29]. The fitness function cannot find the closeness of the test cases when it gives the same value for all test cases that cover the same number of def-use paths and '0' for all test cases that do not cover any def-use paths. As a result, it may lose useful information when selecting promising individuals for recombination. To solve this problem, following the similar procedure in [29], they propose a new multi-objective fitness function. This function evaluates the fitness of test data based on its relation, through dominance [54],

to the definition and use in the data-flow requirement. In particular, it considers a def-use pair as two objectives, *i.e.*, the *def* and the *use*. To evaluate the closeness of a test case *w.r.t.* a target def-use pair, it uses the missed nodes of the dominance paths against these two objectives. The function is set up based on two observations: (1) a test case that covers the *def* is closer than a test case that does not cover both *def* and *use* or covers the *use* only; (2) a test case that misses the *def* or *use* is closer than a test case that misses the *def* or *use* and does not try again to cover it. They follow such a testing method as targeting one def-use pair at one time, which can fulfill a specific test requirement at one time. In the evaluation, they find this GA-based approach is much more effective than random testing, which requires less search time and fewer program iterations.

Vivanti *et al.* [52] use the genetic algorithm to do data flow testing on object-oriented programs. For testing classes in object-oriented programs, a test case is represented as a sequence of method calls [55]. Following the conception of testing on classes [6], they identify three kinds of def-use pairs, *i.e.*, *intra-method pairs*, *inter-method pairs* and *intra-class pairs*. And they use a "node-node" fitness function [56], where the search is first guided toward reaching the first node (*i.e.*, the *def* node), and then from there on toward reaching the second node (*i.e.*, the *use* node). However, the authors find that when targeting individual test objectives at one time, testers face the issue of reasonably distributing the testing resources among all test objectives. Further, for infeasible test objectives, testing resources invested on them will be wasted by definition. To counter these problems, instead of using the classic way of targeting one pair at one time, they apply the whole test suite generation [36] in data flow testing, which optimizes sets of test cases toward covering all test objectives. This approach is expected to less affected by infeasible test objectives. Through the evaluation on SF100 corpus of classes [57], they confirm the test objectives of data-flow testing are much more than those of control flow-based testing (*e.g.*, branch testing) but the resulting test suite is more effective in fault detection.

Denaro *et al.* [32] also use a similar genetic algorithm to augment initial test suites with data flow-based test data in object-oriented systems.

There exist other efforts to apply GA for data flow testing. Liaskos *et al.* [58], [59] hybridize GA with the artificial immune systems [60] (AIS) algorithm to fulfill data flow testing on Java library classes. This combined technique shows its potential in improving the testing performance against GA alone.

Baresi *et al.* developed a GA-based testing tool, Testful [61], [62], for structural testing on Java classes. This GA variant uses a multi-objective fitness function and works both at class and method level. The former puts class objects in useful states, used by the latter to exercise the uncovered code parts of the class. In [63], they try to apply this GA variant to cover def-use pairs. The author points out explicitly exercising def-use pairs for object-oriented programs can often be rewarded because it can correctly relates methods that cooperate with each other by exchanging data (*e.g.*, through objects' fields). Other efforts include [64], [65], which also use GA to automate data flow testing but only evaluate on small examples.

***Optimization-based Search*** There also have been attempts at using optimization-based search techniques to tackle the DFT problem. Nayak *et al.* [66] and Singla *et al.* [67], [68] use particle swarm optimization to do data flow testing. Ghiduk [69] uses ant colony optimization to fulfill data flow testing. All these optimization algorithms are inspired by natural behaviors and simulates them to find optimal solutions. All of these approaches has only been evaluated on toy programs.

***Discussion*** The search-based approach has demonstrated its potential in automatic test data generation for various coverage criteria. In principal, it is also less affected by the problem of finding solutions to non-linear constraints and floating point inputs [70]–[72] than the constraints solving-based approaches (*e.g.*, symbolic execution). But its testing performance heavily depends on the underlying fitness functions, and thus requires careful design and optimization.

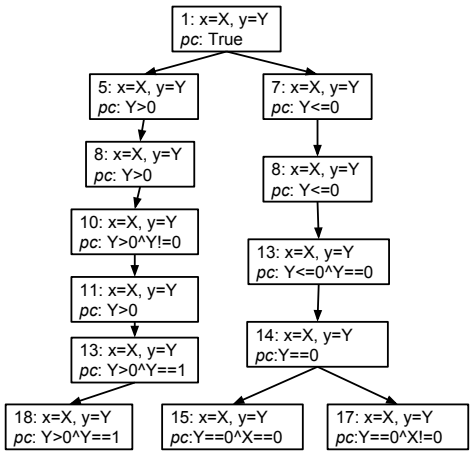
### *D. Symbolic Execution-based Approach*

Symbolic execution is a classic program analysis technique, which was proposed by King [73] in 1976. Due to recently impressive progress in constraint solvers (*e.g.*, Z3 [74], STP [75], Yices [76]) as well as the combination of both concrete and symbolic execution (referred as *dynamic symbolic execution* or *concolic testing* [77], [78]) make it possible to perform automatic path-based testing on real-world programs. Many symbolic execution-based testing tools [77]–[84] bloomed up from both academic and commercial areas.

***Principal of Symbolic Execution*** The symbolic execution technique is a heavy-weight program analysis technique. It uses symbolic values, instead of concrete values, as the program inputs. As a result, the values of program variables can be represented as the symbolic expressions of those inputs. During the symbolic execution of the program under test, at any point, a program state includes (1) the symbolic values of program variables. (2) a *path constraint* (*pc*) over symbolic inputs in the form of a boolean formula which need to be satisfied to reach this program point. (3) a program counter which denotes the next program statement to execute. It works as follows: The *pc* is updated with new constraints over inputs at each branch point during the execution. If the new *pc* is unsatisfiable, the exploration of the corresponding path will terminate. Otherwise, the execution will continue along this branch point such that any solution of the *pc* will execute out the corresponding path. In particular, when both two directions (*i.e.*, branches) of a conditional statement are feasible, the path exploration will fork and continue on. At this time, a search strategy [79], [80], [85] should be adopted to specify the prioritization on search directions. This classic approach of symbolic execution is also referred as *static* symbolic execution (SSE). In Figure 8, we illustrate the symbolic execution on the example program in Figure 2 (Section II). Here, three program paths are explored and the test inputs are generated by solving the collected

| Path | *pc* | Test Input |
|---|---|---|
| $l_4, l_5, l_8, l_9, l_{10}, l_{11}, l_9, l_{13}, l_{18}$ | $y > 0 \wedge y == 1$ | $x \mapsto 1, y \mapsto 1$ |
| $l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{15}$ | $y == 0 \wedge x == 0$ | $x \mapsto 0, y \mapsto 0$ |
| $l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{17}$ | $x! = 0 \wedge y == 0$ | $x \mapsto 1, y \mapsto 0$ |

(a)




(b)

Fig. 8. (a) the test inputs and the path constraints corresponding to different program paths, (b) the corresponding symbolic execution tree. (a) and (b) are based on the Function *power* in Figure 2

path constraints (Figure 8(a)). The execution tree is also given (Figure 8(b)).

***Static Symbolic Execution-based Approach*** Girgis [86] first uses a similar *static* symbolic execution system to generate data flow-based test data. This approach first generates a set of program paths from the control flow graph (CFG) of the program under test *w.r.t.* a certain control-flow criterion (*e.g.*, branch coverage). Since the loops from the CFG may generate infinite program paths, it uses a subset of paths called as *ZOT-subet* to approximate the whole path space by requiring paths to traverse loops zero, one and two times. It then concentrates on those executable paths which can satisfy concerned def-use pairs. In this system, for a tester, she can determine the path feasibility by checking whether the path constraint collected along this path is satisfiable or not. By solving the path constraints of those feasible paths in the ZOT-subet, this system can provide a test suite which fulfills the given data flow testing criterion.

For the example program in Figure 2, this approach first statically explores as many paths as possible *w.r.t.* a control-flow criterion (*e.g.*, branch coverage). Assume it finds a static path $p = l_4, l_5, l_8, l_9, l_{10}, l_{11}, l_9, l_{13}, l_{18}$. Here $p$ traverses the loop (which locates between $l_9$ and $l_{12}$) one time and statically covers the $dua(l_{10}, l_{18}, res)$. The solution $(x \mapsto 0, y \mapsto 1)$ to its corresponding *pc*, (*i.e.*, $y == 1 \wedge y > 0$) can satisfies the pair.

***Discussion*** The static symbolic execution-based approach is a path-based exploration method, which is easy to be adapted for data flow testing. It can systematically explore paths to find ones which can satisfy target def-use pairs. But it has some limitations as well: 1) It uses a control-flow criterion as a coverage metric to guide path exploration, which can mitigate the path explosion problem but may also miss paths for some def-use pairs. For example, three paths in Figure 8(a) have already covered all branches in the function *power*, but the def-use pair $du(l_{10}, l_{17}, res)$ is not satisfied (a new test input $(x \mapsto 1, y \mapsto -1)$ corresponding to the path $l_4, l_7, l_8, l_9, l_{10}, l_{11}, l_9, l_{13}, l_{14}, l_{17}$ can cover this pair). The reason is that a control-flow criterion may not subsume a data-flow criterion. 2) Since it is a static approach without dynamic execution information, some approximations have to be made during data flow testing. For example, it can not precisely reason about which concrete element is referred by $a[x]$ ($a$ is an array and $x$ is an index variable) because the concrete value of $x$ is unknown. One method is to treat $a[x]$ as a use of the whole array $a$.

***Dynamic Symbolic Execution-based Approach*** In the *static* symbolic execution technique, it is not always possible to solve any path constraint, especially when the constraint involves native library functions or non-linear operations such as multiplication or division. The ability of symbolic execution to find more feasible paths is limited by the fact that solving the general class of constraints is undecidable [14].

To counter this problem, Godefroid and Koushik *et al.* [77], [78] interleave the symbolic execution with concrete execution (*i.e.*, *dynamic* symbolic execution (DSE)) to systematically explore program paths. The basic idea is that this hybrid technique collects the path constraint along an execution path (same as static symbolic execution), which is instead triggered by concrete program inputs. If the path constraint become too complex and out of the reach of the constraint solver, these concrete values can be later used to simplify it by value substitution.

In particular, the DSE-based approach starts with an execution path triggered by an initial test input and then iterates the following: from an execution path $p = l_1, \ldots, l_{i-1}, l_i, \ldots, l_n$, DSE picks an executed branch (*i.e.* a branching node[4]) of a *conditional* statement at $l_i$ (the choice depends on an underlying search strategy). It then solves the path constraint collected along $l_1, \ldots, l_{i-1}$ conjuncted with the *negation* of the executed branch condition at $l_i$ to find a new test input. This input will be used as a new test case in the next iteration to generate a new path $p' = l_1, \ldots, l_{i-1}, \bar{l_i}, \ldots$, which deviates from the original path $p$ at $\bar{l_i}$ (the opposite branch direction of the original executed branch at $l_i$), but shares the same path prefix $l_1, \ldots, l_{i-1}$ with $p$. In the context of DFT, if a target def-use pair is covered by this new path $p'$, we will obtain the test case which satisfies this pair. Otherwise, the process will continue until a termination condition (*e.g.* a time bound is reached or the whole path space has been explored) is met.

[4]A *branching node* is an execution instance of an branch in the original code. When a conditional statement is inside a loop, it can correspond to multiple branching nodes along an execution path.

Su *et al.* [31] first adapt this dynamic symbolic execution technique to do data flow testing on top of a DSE engine, CAUT [48], [84], [87], [88]. The DSE algorithm is further enhanced by an efficient guided path search strategy. In their approach, data flow testing is treated as a target search problem. It first finds out a set of *cut points* which must be passed through by any paths to cover a def-use pair. These cut points can narrow down the path search space and guide the path exploration to reach the pair as quickly as possible. To further accelerate the testing performance, it uses a shortest-distance branch first heuristic (which prioritizes a branch direction which has the shortest instruction distance toward a specified target) from directed symbolic execution approaches [89], [90]) and a redefinition path pruning technique (motivated by the definition of data flow testing, *i.e.*, no redefinitions can appear on the sub-path between the *def* and the *use*).

For the example program in Figure 2, assume the target def-use pair is $du(l_8, l_{17}, res)$. DSE starts by taking an arbitrary test input $t$, *e.g.* $t = (x \mapsto 0, y \mapsto 42)$. This test input triggers an execution path $p$

$$p = l_4, l_5, l_8, \underbrace{l_9, l_{10}, l_{11},\ l_9, l_{10}, l_{11}, \ldots,}_{\text{repeated } 42 \text{ times}} l_9, l_{13}, l_{18} \quad (3)$$

which already covers the *def* of $du_1$ at $l_8$. To cover its *use*, the classical DSE approach (*e.g.* with depth-first or random path search [79]) will systematically flip branching nodes on $p$ to explore new paths until the *use* is covered. However, the problem of *path explosion* — hundreds of branching nodes on path $p$ (including nodes from new generated paths from $p$) can be flipped to fork new paths — could make the exploration very slow. In [31], two techniques mentioned before are used to tackle this challenge.

First, the *redefinition pruning* technique is used to remove invalid branching nodes: $res$ is redefined on $p$ at $l_{10}$, so it is useless to flip the branching nodes after the redefinition point (the generated paths passing through the redefinition point cannot satisfy the pair, *cf. Definition* 1). To illustrate, the branching nodes that will not be flipped are crossed out on $p$ and the rest are highlighted in (4). As a result, a large number of *invalid* branching nodes are pruned.

$$p = \boxed{l_4}, l_5, l_8, \underbrace{\boxed{l_9}, l_{10}, l_{11},\ \cancel{l_9}, l_{10}, l_{11}, \ldots,}_{\text{repeated } 42 \text{ times}} \cancel{l_9}, \cancel{l_{13}}, l_{18} \quad (4)$$

Second, a *cut point-guided search strategy* [31] is used to decide which branching node to select first. The *cut points w.r.t.* a pair is a sequence of control points that must be passed through when searching for a path to cover the pair. For example, the cut points of $du_1(l_8,\ l_{17},\ res)$ are $\{l_4,\ l_8,\ l_9,\ l_{13},\ l_{14},\ l_{17}\}$. Since the path $p$ in (4) covers the cut points $l_4$, $l_8$ and $l_9$, the uncovered cut point $l_{13}$ is set as the next search goal. From $p$, there are two unflipped branching nodes, $4F$ and $9F$ (denoted by their respective line numbers followed with $T$ or $F$ to represent the *true* or *false* branch direction). Because $9F$ is closer to cut point $l_{13}$ than $4F$, so $9F$ is flipped. As a result, a new test input $t = (x \mapsto 0, y \mapsto 0)$ can be generated and leads to a new path $p' = l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{15}$. Now the path $p'$ has covered the cut points $l_4, l_8, l_9, l_{13}$ and $l_{14}$ and the uncovered cut point $l_{17}$ becomes the goal. From all remaining unflipped branching nodes, *i.e.* $4F$, $13F$ and $14F$, the branching node $14F$ is chosen because it has the shortest distance toward the goal. Consequently, a new test input $t = (x \mapsto 1, y \mapsto 0)$ is generated which covers all cut points, and $du_1(l_8,\ l_{17},\ res)$ itself.

***Discussion*** The DSE-based approach is a *dynamic* path-based exploration approach, and can achieve more efficient data flow testing than the SSE-based approach. In addition, variable redefinitions caused by aliases can be detected more easily and precisely with the dynamic execution information. However, this approach is still incapable of checking infeasible pairs. Because the DSE-based or SSE-based approach is an explicit path-based testing approach, it is impossible to draw a conclusion on the feasibility of a pair before all program paths are explored. The path explosion problem makes it even more difficult. Without prior knowledge about whether a target pair is feasible or not, these testing approaches (including those we have discussed before) may spend a large amount of time, in vain, to cover an infeasible pair.

### *E. Model Checking-based Approach*

***Principal of Model Checking*** Model checking [91] is a classic approach for formal verification, which includes various techniques (*e.g.*, *explicit model checking* [92], *symbolic model checking* [93] and *bounded model checking* [94]). An important ability of model checking is to construct witnesses or counterexamples when property checking.

At a high level, a model checker takes as input an operational specification of the system under consideration. Meanwhile, a temporal logic formula, expressed in some specification languages, *e.g.*, LTL (Linear Temporal Logic [95]) or CTL (Computation Tree Logic [96]), is accepted to describe some safety property of interest. After that, the model checker searches the entire state space of the system and check whether the property is violated or not. If the property does not hold in some state, then a counter-example will be reported to demonstrate the violation. Otherwise, the property is concluded as satisfied (*i.e.*, not violated). As a result, this model checking approach can be exploited for testing purposes [97] especially when those counter-examples produced by some model checker are interpreted as test cases, which can help a human analyzer to identify and fix the fault.

***WCTL-based Model Checking*** In classic model checking, the verification task usually works on an abstract model, called as a Kripke structure $M = (S, S_0, T, L)$, where (1) $S$ is a set of program states, (2) $S_0 \subseteq S$ is an initial state set, (3) $T \subseteq S \times S$ is a total transition relation, that is, for each $s \in S$ there is a $s' \in S$ such that $(s, s') \in T$. (4) $L : S \to 2^{AP}$ is a labeling function that maps each state $s$ to a set of atomic propositions that hold in $s$.

CTL can be used to express temporal properties of interest, which is based on the Kripke structure. Here we give a brief

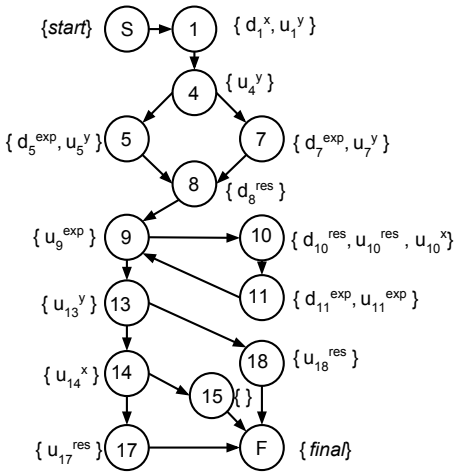



Fig. 9. The Data Flow Graph for the Function *power* in Figure 2.

and informal description of CTL (refer to [96] for details): CTL formulas are built from path qualifiers, modal operators, and standard logical operators. The path qualifiers include **A** (for all paths) and **E** (for some path) and the modal operators include **X** (next time), **F** (eventually), **G** (always), and **U** (until). For a CTL formula $f$ and a state $s$ of Kripke structure $M$, we write $K, q \models f$ if $q$ satisfies $f$ (or briefly written as $q \models f$). For example, "$q \models EXp$" states that there is a path from $q$ such that $p$ holds at the next state; "$q \models \mathbf{EF}p$" states that there is a path from $q$ such that $p$ holds sometime in the future; "$q \models \mathbf{EG}p$" states that there is a path from $q$ such that $p$ holds globally in the future; "$q \models \mathbf{E}[p_1 \mathbf{U} p_2]$" states that there is a path from $q$ such that $p_1$ holds until $p_2$ holds and $p_2$ eventually holds in the future.

Hong *et al.* [98] first use such a Kripke structure-based model checking approach to do data flow testing via a CTL-based model checker. The test obligations of def-use pairs are expressed via WCTL formulas, which are a subclass of CTL formulas. As a result, this approach reduces the problem of data flow testing to the problem of finding witnesses for a set of logical formulas.

A CTL formula $f$ is a WCTL formula if (1) $f$ only contains temporal operators **EX**, **EF** and **EU** (2) for each sub-formula of $f$ of the form $f_1 \wedge f_2 \wedge \cdots \wedge f_n$, every conjunct $f_i$ except at most one is atomic proposition. In their approach, they denote the flow graph $G$ of the program under test as $G = (V, v_s, v_f, A)$ where $V$ is a set of vertices in $G$, $v_s \in V$ is the start vertex, $v_f \in V$ is the final vertex, and $A$ is a finite set of arcs. Here, a vertex represents a statement and an arc denotes a flow of control between two statements. The set of variables that is defined at a vertex $v$ is denoted as $DEF(v)$ and the set of variables that is used at a vertex $v$ is denoted as $USE(v)$. As a result, they view the flow graph $G$ as a Kripke structure $M(G) = (V, v_s, L, A \cup \{(v_f, v_f)\})$, where $L(v_s) = \{start\}$, $L(v_f) = \{final\}$, and $L(v) = DEF(v) \cup USE(v)$ for every $v \in V \setminus \{v_s, v_f\}$.

Figure 9 shows the data flow graph of the example program in Figure 2. We use $d_l^x$ or $u_l^x$ to represent the variable $x$ is defined or used at program point $l$. In Figure 9, the set $DEF(l)$ and $USE(l)$ at the program point $l$ is given. In [98], it characters the test obligation of a def-use pair $du(l_d, l_u, x)$ as a WCTL formula in (5). Here, $def(x)$ is the disjunction of all definitions of $x$, which ensure the subpath between $l_d$ and $l_u$ is a def-clear path *w.r.t.* the variable $x$. Any given $(l, l', x)$ is a def-use pair if and only if the Kripke structure derived from the data flow graph satisfies the formula in (5). Note in this approach, it does not need to know beforehand whether there exists a control flow path between the location $l$ and $l'$ because the formula itself implicitly imposes this constraint.

$$\mathbf{wctl}(d_l^x, u_{l'}^x) = \mathbf{EF}(d_l^x \wedge \mathbf{EXE}[\neg def(x) \mathbf{U}(u_{l'}^x \wedge \mathbf{EF} final)]) \tag{5}$$

For the $du1(l_8, l_{17}, res)$ in (1), it can be expressed via the WCTL formula in (6). A possible witness to this formula is $l_2, l_3, l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{16}, l_{17}$.

$$\mathbf{wctl}(d_{l_8}^{res}, u_{l_{17}}^{res}) = \mathbf{EF}(d_{l_8}^{res} \wedge \mathbf{EXE}[\neg def(res) \mathbf{U}(u_{l_{17}}^{res} \wedge \mathbf{EF} final)]) \tag{6}$$

If the all def-use coverage criterion is required, a test suite should be generated via a set of WCTL formula in (7).

$$\{\mathbf{wctl}(d_l^x, u_{l'}^x) | d_l^x \in DEF(G), u_{l'}^x \in USE(G)\} \tag{7}$$

***Discussion*** This Kripke structure-based model checking approach has the following merits: (1) Since it works on an abstract model, this approach is language independent. It can even extend data flow testing on specification models [99], [100]. (2) This approach casts the data flow testing problem into the model model checking problem, which can benefit from advances in model checkers. However, it may also suffer from some limitations: (1) Theoretically, the worst case number of def-use pairs in this approach can be $O(n^2)$ where $n$ is the number of vertices (*i.e.*, statements) in the graph $G$. As a result, the number of formulas can be quadractic in the size of $G$. If it is applied into inter-procedural program-based testing, the whole graph $G$ built from all functions will contain a large set of vertices. The scalability of this approach could be affected. (2) In addition, this approach can not easily detect infeasible pairs because the abstract model it works on is not aware of underlying path constraints.

***CEGAR-based Model Checking*** Another software model checking approach, called **C**ounter**E**xample-**G**uided **A**bstraction **R**efinement-based (CEGAR) model checking [101]–[103], was proposed in 2002. Since then, it was applied to automatically check safety properties of OS device drivers [101], [104], [105] as well as generate structural test cases [106] (*e.g.*, statement or branch coverage). Several state-of-the-art CEGAR-based model checkers, have also been implemented, *e.g.*, SLAM [101], BLAST [104] and CPAchecker [105]. Given the program source code and a temporal safety specification, this CEGAR-based approach

either statically proves the program satisfies the specification or produces a counter-example path to demonstrate the violation.

The algorithm of the CEGAR-based model checking works on the *Control Flow Automata* (CFA) of the program under test, which is essentially derived from the Control Flow Graph (CFG). Formally, A CFA for a C program is defined as a tuple $(Q, q_0, X, Ops, \rightarrow)$, where $Q$ is a finite set of program locations, $q_0$ is the initial location, $X$ is a set of program variables, $Ops$ is a set of operations on $X$, and $\rightarrow \subseteq (Q \times Ops \times Q)$ is a finite set of edges labeled with operations. The operation set $Ops$ contains (1) *basic blocks* of instructions, *i.e.*, finite sequences of assignments; (2) *assume predicates* written as *assume(p)*, where $p$ is a logical expression over $X$, representing a condition that must be *true* for the labeled edge to be taken. Any program can be converted to this CFA representation [107], [108].

In [106], Beyer *et al.* suggest a CEGAR-based two-phases approach, *i.e.*, *model checking* and *tests from counterexamples*, to automatically generate structural test cases. This method first checks whether the program location $q$ of interest is reachable such that a predicate $p$ (i.e., a safety property) is true at $q$. From the program path which exhibits $p$ at $q$, a CEGAR-based model checker can generate a test case which witnesses the truth of $p$ at $q$. Similarly, it can also produce a test case indicating the falsehood of $p$ at $q$. If all program locations or branches are checked with the predicate $p$ set as *true*, statement or branch coverage can be elegantly achieved.

In particular, in the first phase, the CEGAR-based approach implements an abstract-check-refine loop, where abstraction is done lazily [102]. It starts from a set of abstraction predicates in CFA. If a path that reach the program location $q$ of interest is found, then it checks whether this path can correspond to a concrete program path in the original program. If the path turns out to be *infeasible*, the algorithm will automatically do refinement by consulting an underlying theorem prover to suggest additional abstraction predicates, which can rule out this infeasible path and continue the checking process. Otherwise, a concrete path that can reach the location $q$ is found. In the second phase, the approach collects the symbolic path constraint from the found path, which is then solved by a constraint solver to get the corresponding test case. As a result, this approach can generate test cases without user intervention, and without causing false positives.

Inspired by the work [106], Su *et al.* [31] further adapt this CEGAR-based model checking approach to achieve data flow testing (*w.r.t.* all def-use coverage). A simple but powerful program transformation method is invented to directly encode the test requirement into the program under test. It instruments the original program $P$ to $P'$ and reduce the problem of data flow testing to reachability checking on $P'$. A variable *cover_flag* is introduced and initialized to *false* before the *def* location of a target def-use pair. This flag is set to *true* immediately after the *def*. In oder to find a def-clear path from the *def* location to the *use* location, the *cover_flag* variable is set to *false* immediately after the other definitions on the same variable. Before the *use*, it sets the target predicate *p* as

```
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 1.0/res;
16 }
```

Fig. 10. The transformed function *power* for the def-use pair $du_1(l_8, l_{17}, res)$. The encoded test requirement in showed by the highlighted statements.

```
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. 11. The transformed function *power* for the def-use pair $du_2(l_8, l_{18}, res)$. The encoded test requirement in showed by the highlighted statements.

*cover_flag*==*true*. As a result, if the *use* location is reachable, we obtain a counterexample and conclude that the pair is feasible with a test case. Otherwise, the pair is proved as infeasible (or, since the problem is undecidable, the algorithm does not terminate within a constrained time bound, and only reports *unknown* as the result).

For the example program in Figure 2 and the two pairs $du_1$ and $du_2$ in (1) and (2), the transformed program encoded with these two test requirements are showed in Figure 10 and Figure 11, respectively. For the pair $du_1(l_8, l_{17}, res)$, Figure 10 shows the transformed function *power* and the encoded test requirement of $du_1$ in highlighted statements. The variable *cover_flag* is introduced at $l_2$. It is initialized to *false* and set as *true* immediately after the *def* at $l_7$, and set to *false* immediately after the other definitions on variable $res$ at $l_{10}$. Before the *use*, a checkpoint is set to verifiy whether *cover_flag* can be *true* at $l_{14}$. If the checkpoint is unreachable, it can prove that this pair is infeasible. Otherwise, a counter-example, *i.e.* a test case that covers this pair, can be generated. In this example, a possible path $l_2, l_3, l_4, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{16}, l_{17}$ can be found by the model checker. But for the pair $du_2(l_8, l_{18}, res)$ in Figure 11,

the model checker can quickly conclude no paths can reach the check point to witness the truth of *cover_flag*. So $du_2$ is an infeasible pair.

***Discussion*** This CEGAR-based model checking approach has several merits: (1) The data flow testing problem can be easily transformed into the reachability checking problem from the model checking perspective. Due to the conservativeness of static data-flow analysis, test objectives contain infeasible def-use pairs. This approach can easily detect and eliminate infeasible pairs without false positives, which can save much testing time as well. Furthermore, this approach can also generate counter-examples (*i.e.*, test cases) for feasible def-use pairs. The technique itself can further benefit from the further advances in the ability of CEGAR-based model checkers. (2) In this approach, the test requirement can be directly encoded into the program under test without manually writing temporal properties like CTL/WCTL formulas. It is more flexible and easy to implement than other model checking-based approaches. However, this approach may also have performance limitations. The CEGAR-based approach is essentially a *static* model checking approach, its actual performance may not be as high as other *dynamic* testing approaches (*e.g.*, dynamic symbolic execution-based approach [31]) when generating test cases for feasible pairs.

### *F. Other Approaches*

Khamis *et al.* [109] enhance the Dynamic Domain Reduction procedure [110] (DRR) to do data flow testing for Pascal. The DDR technique basically incorporates the ideas from symbolic execution, constraint-based testing, and dynamic test data generation. It starts from the initial domains of input variables as well as the program flow graph, and dynamically drives the execution through a specified path to reach the target test objective. During the path exploration, symbolic execution is used to progressively reduce the domain of input variables. And a search algorithm is used to find a set of values that can satisfy the path constraint. The authors [109] enhance this technique with some methods for handling loops and arrays. But it only illustrates the idea with some proof-of-conception examples, and its practicality is not clear.

Buy *et al.* [111] combine data-flow analysis, static symbolic execution and automated deduction to perform data flow testing. Symbolic execution first identifies the relation between the input and output values of each method in a class, and then collects the method preconditions from a feasible and def-clear path that can cover the target pair. An automated backward deduction technique is later used to find a sequence of method invocations (i.e., a test case) to satisfy these preconditions. However, little evidence is provided on the practicality of this approach. Later, Vincenzo *et al.* [112] extend this technique from single classes to address the problem of interclass testing, *i.e.*, test of interactions among classes. It incrementally generates test cases from simple classes to more complex classes.

Baluda *et al.* [113]–[115] proposed a novel approach called Abstract Refinement and Coarsening (ARC) to improve the accuracy of branch coverage testing by identifying infeasible branches. This approach is rooted from a property checking (model checking) algorithm [116], [117], which aims to either prove a faulty statement is unreachable, or produce a test case that executes the statement. Baluda *et al.* adapted this algorithm for structural testing, and enhanced it with "coarsening" to improve its scalability. In [114], Baluda claims this approach is independent from the coverage criteria and is particularly suitable for such coverage criteria that suffer greatly from the presence of infeasible test objectives as data flow testing criteria.

***Summarization*** Despite several challenges in identifying data flow-based test data, researchers have developed various general approaches to automating this process. We find total 27 technical research papers from the publication repository which are related with test generation. Table I summarizes the testing approaches used by these papers in chronological order. For each paper, the table lists the technique which it uses, the language which it supports, and the tool in which it was implemented. It reveals the test generation problem has been continuously concerned in academic over the past 20 years. The search-based testing and collateral coverage-based testing are the two most popular techniques used to automate test data generation. A reasonable explanation is that these two approaches are easy to be applied in DFT. The symbolic execution-based and model checking-based techniques received attention very early, but until recently did they find practical adoption. The fact that symbolic execution and model checking techniques are more complicated to implement than other approaches may explain this phenomenon. Additionally, both procedural language (*e.g.*, C) and object-oriented language (*e.g.*, Java) have respective supporting tools. However, none of these tools are commercial tools[5] More efforts are still needed to develop more efficient and easy-to-implement techniques.

## IV. Approaches for Coverage Tracking

This section discusses some approaches in the present literature used to track data flow coverage and summarizes available data flow coverage tools.

### *A. Coverage Tracking Techniques*

Test coverage is a common vehicle to measure how thoroughly software is tested and how much confidence software developers have in its reliability. Several techniques [21], [38], [127]–[130] have been developed to track the coverage of def-use pairs.

Frankl [127] proposes a *deterministic finite automata* (DFA)-based approach to track the coverage status of def-use pairs. In his approach, a pair $du$ is associated with a regular expression which describes a set of control flow paths covering it. Each automata associated with $du$ is checked against all execution paths. Once one path is accepted by some automata, the pair $du$ is set as covered. But this approach

[5] Matlab is used to simulate the optimization approach instead of directly generating test data.

TABLE I
A SUMMARIZATION OF DATA FLOW TESTING APPROACH

| Author, Year | Technique | Language | Tool |
|---|---|---|---|
| Girgis [86], 1993 | Symbolic Execution | C | - |
| Marré *et al.* [40], 1996 | Collateral Coverage | - | - |
| Merlo *et al.* [39], 1999 | Collateral Coverage | C | CANTO |
| Buy *et al.* [111], 2000 | Symbolic Execution, Backward Deduction | Java | - |
| Martena *et al.* [112], 2002 | Symbolic Execution, Backward Deduction | C++ | - |
| Marré *et al.* [41], 2003 | Collateral Coverage | C | - |
| Hong *et al.* [98], 2003 | Model Checking | Specification Language | SMV [118] |
| Oster [64], 2005 | Genetic Algorithm | Java | - |
| Girgis [29], 2005 | Random Testing, Genetic Algorithm | Fortran | - |
| Malevris *et al.* [37], 2006 | Collateral Coverage | Fortran, Pascal, C, Java | TESTBED, SPM, Volcano |
| Santelices *et al.* [42], 2006 | Collateral Coverage | - | - |
| Santelices *et al.* [38], 2007 | Collateral Coverage | Java | DuaF [119] |
| Ghiduk *et al.* [30], 2007 | Random Testing, Genetic Algorithm | C++ | - |
| Liaskos *et al.* [60], 2007 | Optimization Algorithm, Genetic Algorithm | Java | - |
| Liaskos *et al.* [58], 2007 | Genetic Algorithm | Java | - |
| Liaskos *et al.* [59], 2008 | Genetic Algorithm | Java | - |
| Deng *et al.* [65], 2009 | Genetic Algorithm | Java | - |
| Nayak *et al.* [66], 2010 | Optimization Algorithm | - | - |
| Ghiduk [69], 2010 | Optimization Algorithm | C++ | PCTDACO |
| Miraz [63], 2010 | Genetic Algorithm | Java | Testful [120] |
| Khamis *et al.* [109], 2011 | Dynamic Domain Reduction | Pascal | - |
| Singla *et al.* [67], 2011 | Optimization Algorithm | - | Matlab |
| Singla *et al.* [68], 2011 | Optimization Algorithm | - | Matlab |
| Vivanti *et al.* [52], 2013 | Genetic Algorithm | Java | Evosuite [121] |
| Girgis *et al.* [122], 2014 | Genetic Algorithm | C# | - |
| Su *et al.* [31], 2015 | Random Testing, Symbolic Execution, Model Checking | C | CAUT [123], BLAST [124], CPAchecker [125] |
| Denaro *et al.* [32], 2015 | Genetic Algorithm, Random Testing | Java | EvoSuite, Randoop [126] |

has to do special handling when the procedure under test recursively calls itself. Ostrand *et al.* [128] use a *memory tracking* technique to precisely determine which pairs are covered while Kamkar [131] use *dynamic slicing* to improve coverage precision.

Hogan and London [129] exploit code instrumentation to track data flow coverage, which is later known as the *last definition* technique. In their approach, a table of def-use relations is generated from the data flow graph and a probe is inserted at each code block. The runtime routine records each variable that has been defined and the block at which it was defined. When a block uses this defined variable is executed, the last definition of this variable is verified and the pair is set as covered.

Misurda *et al.* [130] propose a *demand-driven* strategy to track def-use pair coverage, which aims to improve the performance of the static instrumentation approach [129]. The approach works as follows: first, it determines all variable definitions in a test region and inserts *seed* probes at those definitions; second, when a definition is reached, *coverage* probes are inserted on demand at all its reachable uses; third, the probe for a use will be immediately deleted once this use is reached, and the pair, composed of the most recently visited definition and this use, is marked as covered.

Santelices and Harrold [38] develop an efficient *matrix-based* strategy to directly track data flow coverage. In this strategy, a *coverage matrix* is created, and is initialized as zeros, where each column corresponds to a variable use, and each cell in a column represents a definition for that use (*i.e.*, a cell corresponds to a def-use pair). Importantly, this layout helps reduce the runtime cost of probes by quickly accessing the matrix cell through two indices, *i.e.*, a use ID and a definition ID. At runtime, the probes track the last definition of a variable. At each use, a probe is also inserted, which will use the use ID and the last definition ID to update the coverage status of the pair in the matrix.

In [38], the same authors propose a novel *coverage inference* strategy, which uses the branch coverage to infer data flow coverage. Pairs are divided into tree types, *i.e.*, *inferable*, *conditionally inferable* and *non-inferable* pairs by using static analysis before dynamic execution. At runtime, this approach tracks branch coverage, which is a less costly code instrumentation. After test suite execution, it outputs actually covered and conditionally covered pairs. Readers can refer to the collateral coverage-based testing approach in Section III-B for details.

In order to make coverage tracking more scalable, Harrold [132] develops a technique on multi-processor systems to accept tests and produces parallelizable coverage tracking workload. The workload can be scheduled either statically or dynamically onto different architectural platforms. The evaluation on a shared memory multiprocessor system shows a good speedup can be obtained over the uniprocessor system.

There is also some work that exploits dynamic data flow analysis [31], [133] to improve the precision of tracking data flow coverage.

TABLE II
A SUMMARIZATION OF DATA FLOW COVERAGE TOOLS. "-" MEANS

| Tool | Language | Coverage | Infrastructure | Technique | Availibility |
|---|---|---|---|---|---|
| ATAC [134] | C/C++ | Intra | A Yacc-based Parser | Last definition | ✓ |
| Coverlipse [135] | Java | Intra | Eclipse | Path recording | ✓ |
| DaTeC | Java | intra/inter | Soot | - | |
| DuaF [119] | Java | Intra/Inter | Soot | Coverage inference | ✓ |
| CAUT [123] | C | Intra/Inter | CIL | Last definition | ✓ |
| ASSET | Pascal | Intra | - | Automata | |
| TACTIC | C | Intra | - | Memory tracking | |
| POKE-TOOL | C | Intra | - | Automata | |
| JaBUTi [136] | Bytecode | Intra/Inter | Bytecode tool | Last definition | ✓ |
| JMockit [137] | Java | Inter | ASM | - | ✓ |
| Jazz | Java | Intra | Eclipse, Jikes RVM | Demand-driven | |
| DFC | Java | Intra | Eclipse | - | |
| BA-DUA [138] | Java | Intra | ASM | Bitwise Algorithm | ✓ |

### B. Coverage Tools

There have been lots of robust coverage tools [139] at hand for statement and branch coverage, but the tools for data flow coverage are less developed. Table II summarizes the coverage tools for data flow testing, including ATAC [129], [140], Coverlipse, DaTec [26], [141], DuaF [38], CAUT [31], ASSET [5], [127], [142], TACTIC [143], POKE-TOOL [144], JaBUTi [145], JMockit, Jazz [146], DFC [147], [148], BA-DUA [149], [150]. For each tool, the table lists the language it supports, whether it tracks intra- or inter-procedure pairs or both, the analysis infrastructure it is based on, the coverage tracking technique it uses, and its availability. There are total 13 tools, and half of them are online available. However, none of these tools are commercial tools, which is also reported by Hassan *et al.* [151] and Araujo *et al.* [150] recently.

## V. APPLICATIONS

Data Flow Testing (DFT) has not only empirically demonstrated its effectiveness [7], [10], [12] in revealing software bugs, it has also found its usefulness in other applications. This section discusses three aspects of the applications of DFT: (1) software fault localization, (2) web application testing, and (3) specification consistency checking.

### A. Software Fault Localization

Software fault localization is a vital activity in program debugging to locate program errors and bugs, which is tedious and time consuming. Agrawal *et al.* [152] propose a novel method which combines DFT and execution slices together to achieve more efficient fault localization. Their work is based on such an assumption that the fault lies in the slice of a test case which fails on execution instead of which succeeds on execution. As a result, the attention is restrict to the statements in the failed slice that do not appear in the successful slice, *i.e.*, the *dice* (the set difference of two execution slices). A data flow testing tool ATAC [129], [140] is used to generate data flow tests. These tests are later used to detect seeded faults and calculate execution slices from a Unix sort program. In the evaluation, they found data flow tests could effectively detect those seeded errors and the dice could notably improve the fault localization performance.

Santelices *et al.* [153] propose a lightweight fault-localization technique, which uses different coverage criteria to detect suspicious faulty statements in a program. In their approach, they use tests from lightweight coverage entities including statements, branches and def-use pairs to investigate the benefits of different coverage types in fault localization. The study shows that different faults are found best by different coverage types, but their combination can achieve the overall best performance.

### B. Web Application Testing

In recent years, the rapid development of web applications enrich people's daily life. But testing web applications becomes a tough job when the architecture and implementation become more and more complicate. In the past years, several work has been done to tackle testing problems in web applications from the perspective of DFT.

Since the data in web applications can be stored in HTML documents, it could affect the data interactions between the client and the server. Liu *et al.* [154] extend the DFT method for web applications to check the correctness of such data interactions. In their approach, they propose a Web Application Test Model (WATM) to describe the application under test and a DFT structure model to capture the data flow information. In WATM, each part in the application will be modeled as an object which can be *client pages* for an HTML document, *server pages* for a Common Gateway Interface script and *components* for a Java applet or an ActiveX tool and *etc.* Each of these models is composed of attributes and operations to store the fundamental information. The DFT structural model uses four types of flow graphs to capture the relevant data flow information. After obtaining the data flow information, the data flow test cases will be generated to cover the intra-object, inter-object and inter-client aspects. In this way, DFT is extended to test web applications.

Qi *et al.* [155] develop a multiple agent-based DFT method to test web applications. They split the testing task into three levels, *i.e.*, a method level, an object level, and an object cluster level. Each test agent from these levels will construct a corresponding program model annotated with data-flow

information. In this way, the whole task of data-flow testing can be divided into subtasks and performed by these test agents.

Mei *et al.* [156] exploit DFT to test service-oriented workflow applications such as WS-BPEL applications. They find XPath plays an important role in workflow integration but may contain wrong data extracted from XML messages, which undermines the reliability of these applications. Thus, they develop a data structure called *XPath Rewriting Graph* (XRG) to model the XPath in WS-BPEL. And then they conceptually determine the def-use pairs in the XRG and propose a family of data flow testing criteria to test WS-BPEL applications.

### C. Specification Consistency Checking

Various specification models are widely used in software development to build reliable systems, which helps automatically generate a conforming implementation. As a result, checking model consistency is an important vehicle to ensure implementation correctness. Wang *et al.* [157] propose a DFT-based approach to check requirement model inconsistency. The approach can be summarized as four procedures: (1) construct the requirement model from system requirements, (2) construct relevant call sequences to cover the inter-method usages in these models, (3) obtain boolean constraints from these call sequences and derive a test suite, (4) check model consistency by applying this DFT-based test suite. Additionally, developers can compare their original understanding against the requirements through examining this test suite.

There are also some work that generate data flow-based test suites from specification models like SDL (Specification and Description Language) [100] and statecharts [99]. The resulting test suites provide the capability to check whether the implementation conforms to high-level specification models.

### D. Other Applications

Apart from the above applications, DFT has also been applied to test other programs or applications. Zhao [158] use DFT to test aspect-oriented programs. Harrold *et al.* [133] use DFT to check parallelized code. In addition, DFT was applied to test object-oriented libraries [24] and service choreography [159].

Additionally, we investigate the percentage of each language which data flow testing has been applied to (showed in Figure 12). We find several interesting phenomena: First, the object-oriented languages (*e.g.*, C++ and Java) are the most popular language in enforcing data flow testing. Second, specification languages and web services also attract wide research interests. Third, data flow testing is originally applied to procedural languages (*e.g.*, Fortran, Pascal, C), but in recent years, object-oriented programs gain more emphasis since DFT can help find more subtle faults when checking object states.

## VI. Recent Advancement

This section discusses three strands of recent advancement in data flow testing: (1) new coverage criterion, (2) dynamic data-flow analysis, and (3) efficient coverage tracking.

***New Coverage Criterion*** Hassan *et al.* [151] introduce a new family of coverage criteria, called *Multi-Point Stride Coverage*

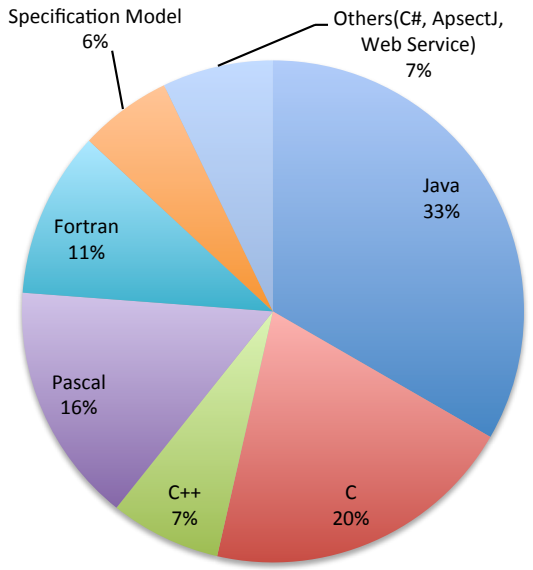



Fig. 12. Percentage of each language used in data flow testing.

(MPSC). Instrumentation for MPSC with gap $g$ and $p$ points records the coverage of tuples $(b_1, b_2, \ldots, b_p)$ of branches taken, where each branch in the tuple is the one taken $g$ branches after the previous one. The empirical evaluation shows this MPSC coverage, generalized from the branch coverage, can predict the effectiveness of a test suite with a similar or higher level of accuracy than the all def-use coverage. And the instrumentation for MPSC coverage is also usually more efficient than that for data flow coverage.

Bardin *et al.* [160] propose a *label coverage* criterion to emulate a number of advanced criteria (including statement, decision, multi-condition and weak mutation coverage criteria). This label coverage criterion is both expressive and amenable to efficient automation. The motivation is to bridge the gap between the coverage criteria supported by symbolic automatic test generation tools and the most advanced coverage criteria found in the literature. Although this label criterion currently cannot handle those criteria that impose constraints on paths (*e.g.*, data flow criteria), this work indicates a direction to come up with a more general coverage criterion to express data flow testing criteria, so that it can directly use a symbolic executor as a black-box tool to facilitate testing automation.

Alexander *et al.* [161] extend the classic data flow criteria to test and analyze the polymorphic relationships in object-oriented systems. The proposed new coverage criteria consider definitions and uses between state variables of classes, particularly in the presence of inheritance, dynamic binding, and polymorphic overriding of state variables and methods. The aim is to increase the fault detection ability of DFT in object-oriented programs.

***Dynamic Data-flow Analysis*** Denaro *et al.* [162], [163] investigate the limits of the traditional static data-flow analysis used in DFT. They use a *dynamic data flow analysis*

technique [32] to identify the relevant data flow relations by observing concrete program executions. This approach exploits the precise alias information available from concrete executions to relate memory data and class state variables with each other. As a result, it can be considerably precise than considering statically computed alias relations, which is the typical over-approximation when integrating alias information in static data-flow analysis.

The evaluation on five Java projects reveals that a large set of data flow relations (*i.e.*, the def-use pairs) are missed by the traditional static data flow analysis, which undermines the effectiveness of the previous data flow testing approaches. This dynamic data flow analysis technique actually sheds light on a new direction of data flow testing which can better encompass data flow-based test objectives.

***Efficient Coverage Tracking*** One factor precluding broad adoption of DFT attributes to the cost of tracking the coverage of def-use pairs by tests. Since DFT aims to achieve more comprehensive program testing, its runtime cost imposed by code instrumentation is considerably higher than that of other structural criteria. Some techniques [38], [130] have already been proposed to tackle this problem, which rely on complex computations and expensive data structures.

Inspired by classic solution for data flow problems such as *reaching definition* [23], Chaim and Araujo [149], [150] invent a Bitwise Algorithm (BA) algorithm, which uses bit vectors with bitwise operations to track data flow coverage for Java bytecode programs. For each instruction, this approach computes the defined and used variables (local variables or fields) by using known data-flow analysis techniques. After that, it instruments BA code at each instruction, which is used to determine the coverage of pairs. The BA code keeps track of three working sets, *i.e.*, the *alive* pairs, the current *sleepy* pairs and the *covered* pairs, which are updated during the execution of tests. These working sets are implemented in bit vectors and manipulated with efficient bitwise operations, whose sizes are given by the number of pairs of the method under test.

The authors also give the correctness proof [164] and the theoretical analysis, which show their algorithm is expected to require less memory and execution time than the previous demand-driven and matrix-based approaches [38], [130]. In their evaluation [149], this conclusion is further corroborated by simulating these instrumentation strategies [165]. In [150], this approach was further applied to tackle large systems with more than 200KLOCs and 300K pairs, and its execution overhead was comparable to that imposed by a popular control-flow testing tool.

## VII. A Hybrid Data Flow Testing Framework

Data flow testing helps test software program more thoroughly, but brings several challenges in its enforcement. In this section, we propose a hybrid data flow testing framework as showed in Figure 13, which combines various approaches surveyed to create a more practical DFT strategy. It is composed of three basic components, *i.e.*, a data-flow analyzer, a test data generator, and a coverage monitor.

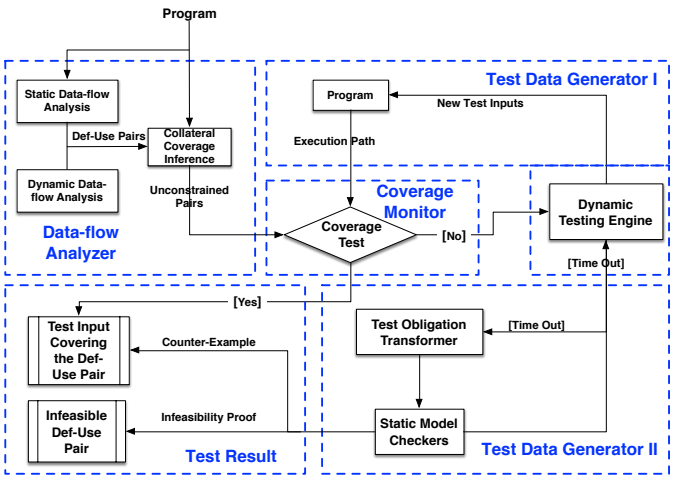



Fig. 13. The Hybrid Data Flow Testing Framework.

***Data-flow Analyzer*** The data-flow analyzer is responsible for identifying test objectives, *i.e.*, def-use pairs. It can use known static data-flow analysis techniques, as well as the *dynamic* data-flow analysis [32], [162] to better encompass def-use pairs. Additionally, the analyzer can use the collateral coverage-based approach to infer the coverage relation between def-use pairs themselves or between def-use pairs and other program structs (*e.g.*, statements, branches). The intention is to identify a minimal set of pairs (*i.e.*, *unconstrained* pairs), whose coverages imply the coverage of others, to save testing cost.

***Test Data Generator*** The test data generator aims to efficiently generate suitable test cases for def-use pairs. It consists of two types of testing approaches: 1) Various *dynamic* testing approaches can be adopted, including random testing, genetic/optimization-based testing and symbolic execution-based testing. These approaches are efficient to satisfy those easily coverable pairs and output the corresponding test cases. However, the limitation of these dynamic testing approaches is that they can only identify feasible test objectives but incapable of dealing with infeasible ones. 2) So, when the allocated testing budgets (*e.g.*, testing time) are used up, the model checking-based approaches are used to handle the remaining pairs (including infeasible ones). The test requirements imposed by these pairs are first transformed into acceptable forms for model checkers, and then infeasible pairs (as well as some feasible pairs) can be easily detected by this verification-based approach. For infeasible pairs, the infeasibility proofs are provided, which prevent testing time being wasted on them. If there are still uncovered pairs after (1) and (2), more testing budgets could be invested, and restart another round of the dynamic and static testing until the intended coverage level is achieved.

***Coverage Monitor*** The coverage monitor tracks the coverage

of test objectives in an efficient way. Such instrumentation approaches as the BA algorithm [150] can be adopted.

At the high level, given a program as input, this hybrid framework (1) *outputs test data for feasible test objectives* and (2) *eliminates infeasible test objectives — without any false positives*. It is hopefully to have better performance with the combination of strengths from its component approaches. The framework can also benefit from the most recent advancements in data-flow analysis, test data generation, and coverage tracking.

## VIII. CONCLUSION

In this survey, we present an overview of data flow testing and summarize several challenges in its enforcement. We systematically investigate various general approaches developed to automate test generation and different techniques which have been used to track data flow coverage. Some recent research directions and advancements in this field are also discussed. The applications of data flow testing are presented, which further demonstrates the usefulness of this testing strategy. With help of continuous research efforts, we believe data flow testing will become a more practical testing method to ensure software reliability.

## REFERENCES


[1] P. M. Herman, "A data flow analysis approach to program testing." *Australian Computer Journal*, vol. 8, no. 3, pp. 92–96, 1976.

[2] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. 9, no. 3, pp. 347–354, 1983.

[3] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[4] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Trans. Software Eng.*, vol. 15, no. 11, pp. 1318–1332, 1989.

[5] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[6] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *SIGSOFT FSE*, 1994, pp. 154–163.

[7] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Trans. Softw. Eng.*, vol. 19, no. 8, pp. 774–787, Aug. 1993.

[8] L. M. Foreman and S. H. Zweben, "A study of the effectiveness of control and data flow testing strategies," *Journal of Systems and Software*, vol. 21, no. 3, pp. 215–228, 1993.

[9] E. J. Weyuker, "More experience with data flow testing," *IEEE Trans. Software Eng.*, vol. 19, no. 9, pp. 912–919, 1993.

[10] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE*, 1994, pp. 191–200.

[11] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*, 1998, pp. 153–162.

[12] A. Khannur, *Software Testing - Techniques and Applications*. Pearson Publications, 2011.

[13] J. Edvardsson, "A survey on automatic test data generation," 1999.

[14] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[15] B. Beizer, *Software Testing Techniques*. International Thomson Computer Press, 1990.

[16] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[17] K. Kennedy, *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division, 1979.

[18] W. Wögerer, "A survey of static program analysis techniques," *Vienna University of Technology*, 2005.

[19] J. C. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Trans. Software Eng.*, vol. 5, no. 3, pp. 226–236, 1979.

[20] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, pp. 137–147, 1976.

[21] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 2, pp. 175–204, Mar. 1994.

[22] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural def-use associations for C systems with single level pointers," *IEEE Trans. Softw. Eng.*, vol. 20, no. 5, pp. 385–403, May 1994.

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

[24] B. Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Rutgers University, Tech. Rep. DCS-TR-382, March 1999.

[25] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object-oriented systems," *IEEE Trans. Software Eng.*, vol. 29, no. 11, pp. 1005–1018, 2003.

[26] G. Denaro, A. Gorla, and M. Pezzè, "Contextual integration testing of classes," in *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 246–260.

[27] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 75–84.

[29] M. R. Girgis, "Automatic test data generation for data flow testing using a genetic algorithm," *J. UCS*, vol. 11, no. 6, pp. 898–915, 2005.

[30] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *APSEC*, 2007, pp. 41–48.

[31] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *37th International Conference on Software Engineering, ICSE '15*, 2015.

[32] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *37th International Conference on Software Engineering, ICSE '15*, 2015.

[33] T. Y. Chen, R. G. Merkel, G. Eddy, and P. K. Wong, "Adaptive random testing through dynamic partitioning." in *QSIC*, 2004, pp. 79–86.

[34] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of java programs," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 221–232.

[35] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, 2010, pp. 182–191.

[36] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[37] N. Malevris and D. F. Yates, "The collateral coverage of data flow criteria when branch testing," *Information & Software Technology*, vol. 48, no. 8, pp. 676–686, 2006.

[38] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 343–352.

[39] E. Merlo and G. Antoniol, "A static measure of a subset of intra-procedural data flow testing coverage based on node coverage." in *CASCON*, 1999, p. 7.

[40] M. Marré and A. Bertolino, "Unconstrained duas and their use in achieving all-uses coverage," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '96. New York, NY, USA: ACM, 1996, pp. 147–157.

[41] ——, "Using spanning sets for coverage testing," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 974–984, Nov. 2003.

[42] R. A. Santelices, S. Sinha, and M. J. Harrold, "Subsumption of program entities for efficient coverage and monitoring," in *Third International Workshop on Software Quality Assurance, SOQUA 2006, Portland, Oregon, USA, November 6, 2006*, 2006, pp. 2–5.

[43] T. Robschink and G. Snelting, "Efficient path conditions in dependence graphs," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 478–488.

[44] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test., Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[45] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, 2007, pp. 1098–1105.

[46] Z. Awedikian, K. Ayari, and G. Antoniol, "MC/DC automatic test input data generation," in *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, 2009, pp. 1657–1664.

[47] K. Ghani and J. A. Clark, "Automatic test data generation for multiple condition and MCDC coverage," in *The Fourth International Conference on Software Engineering Advances, ICSEA 2009, 20-25 September 2009, Porto, Portugal*, 2009, pp. 152–157.

[48] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao, "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, June 30 - July 2, 2014*, 2014, pp. 98–107.

[49] P. Ammann, A. J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *International Symposium on Software Reliability Engineering*, 2003, pp. 99–107.

[50] R. Inc, "Do-178b: Software considerations in airborne systems and equipment certification," *Requirements and Technical Concepts for Aviation*, December 1992.

[51] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[52] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *ISSRE*, 2013, pp. 370–379.

[53] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs (2Nd, Extended Ed.)*. New York, NY, USA: Springer-Verlag New York, Inc., 1994.

[54] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[55] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, 2004, pp. 119–128.

[56] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information & Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[57] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 178–188.

[58] K. Liaskos, M. Roper, and M. Wood, "Investigating data-flow coverage of classes using evolutionary algorithms," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, 2007, p. 1140.

[59] K. Liaskos and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," in *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, 2008, pp. 211–220.

[60] ——, "Automatic test-data generation: An immunological approach," in *Proceedings of Testing: Accademic and Industrial Conference - Practice and Research Techniques (TAIC PART '07)*, Windsor, UK, 10-14 September 2007, pp. 77–81.

[61] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: An evolutionary test approach for java," in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, 2010, pp. 185–194.

[62] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 281–284.

[63] M. Miraz, "Evolutionary testing of stateful systems: a holistic approach," Ph.D. dissertation, Politecnico di Milano, 2010.

[64] N. Oster, "Automated generation and evaluation of dataflow-based test data for object-oriented software." in *QoSA/SOQUA*, 2005, pp. 212–226.

[65] M. Deng, R. Chen, and Z. Du, "Automatic test data generation model by combining dataflow analysis with genetic algorithm," in *Pervasive Computing (JCPC), 2009 Joint Conferences on*, 2009, pp. 429 – 434.

[66] N. Nayak and D. P. Mohapatra, "Automatic test data generation for data flow testing using particle swarm optimization," in *IC3 (2)*, 2010, pp. 1–12.

[67] S. Singla, D. Kumar, H. M. Rai, and P. Singla, "A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts," *Journal of Advanced Science and Technology*, vol. 37, pp. 15–26, 2011.

[68] S. Singla, P. Singla, and H. M. Rai, "An automatic test data generation for data flow coverage using soft computing approach," *IJRRCS*, vol. 2, no. 2, pp. 265–270, 2011.

[69] A. S. Ghiduk, "A new software data-flow testing approach via ant colony algorithms," *Universal Journal of Computer Science and Engineering Technology*, vol. 1, no. 1, pp. 64–72, October 2010.

[70] K. Lakhotia, P. McMinn, and M. Harman, "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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 95–104.

[71] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "Flopsy - search-based floating point constraint solving for symbolic execution," in *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, 2010, pp. 142–157.

[72] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, "Symbolic path-oriented test data generation for floating-point programs," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 1–10.

[73] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[74] "Z3," http://z3.codeplex.com/.

[75] "STP," https://sites.google.com/site/stpfastprover/.

[76] "Yices," http://yices.csl.sri.com/index.shtml.

[77] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 213–223.

[78] K. Sen, D. Marinov, and G. Agha, "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*. New York, NY, USA: ACM, 2005, pp. 263–272.

[79] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.

[80] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[81] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *TAP*, 2008, pp. 134–153.

[82] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[83] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A symbolic execution extension to java pathfinder," in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, 2007, pp. 134–138.

[84] Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding, and J. Hu, "Test data generation for derived types in C program," in *TASE*, 2009, pp. 155–162.

[85] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, 2006, pp. 322–335.

[86] M. R. Girgis, "Using symbolic execution and data flow criteria to aid test data selection," *Softw. Test., Verif. Reliab.*, vol. 3, no. 2, pp. 101–112, 1993.

[87] X. Yu, S. Sun, G. Pu, S. Jiang, and Z. Wang, "A parallel approach to concolic testing with low-cost synchronization," *Electr. Notes Theor. Comput. Sci.*, vol. 274, pp. 83–96, 2011.

[88] T. Sun, Z. Wang, G. Pu, X. Yu, Z. Qiu, and B. Gu, "Towards scalable compositional test generation," in *QSIC*, 2009, pp. 353–358.

[89] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *EuroSys*, 2010, pp. 321–334.

[90] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS*, 2011, pp. 95–111.

[91] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[92] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, 1985, pp. 97–107.

[93] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, Pittsburgh, PA, USA, 1992, uMI Order No. GAX92-24209.

[94] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, 1999, pp. 193–207.

[95] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

[96] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, 1981, pp. 52–71.

[97] G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: a survey," *Softw. Test., Verif. Reliab.*, vol. 19, no. 3, pp. 215–261, 2009.

[98] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *ICSE*, 2003, pp. 232–243.

[99] H. S. Hong, Y. G. Kim, S. D. Cha, D. Bae, and H. Ural, "A test sequence selection method for statecharts," *Softw. Test., Verif. Reliab.*, vol. 10, no. 4, pp. 203–227, 2000.

[100] H. Ural, K. Saleh, and A. W. Williams, "Test generation based on control and data dependencies within system specifications in SDL," *Computer Communications*, vol. 23, no. 7, pp. 609–627, 2000.

[101] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *POPL*, 2002, pp. 1–3.

[102] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*, 2002, pp. 58–70.

[103] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *ICSE*, 2003, pp. 385–395.

[104] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast: Applications to software engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, pp. 505–525, Oct. 2007.

[105] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *CAV*, 2011, pp. 184–190.

[106] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating tests from counterexamples," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 326–335.

[107] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 213–228.

[108] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[109] A. Khamis, R. Bahgat, and R. Abdelaziz, "Automatic test data generation using data flow information," *Dogus University Journal*, vol. 2, pp. 140–153, 2011.

[110] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw., Pract. Exper.*, vol. 29, no. 2, pp. 167–193, 1999.

[111] U. A. Buy, A. Orso, and M. Pezzè, "Automated testing of classes." in *ISSTA*, 2000, pp. 39–48.

[112] V. Martena, A. Orso, and M. Pezzè, "Interclass testing of object oriented software," in *8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), 2-4 December 2002, Greenbelt, MD, USA*, 2002, pp. 135–144.

[113] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 59–66.

[114] M. Baluda, "Automatic structural testing with abstraction refinement and coarsening," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, 2011, pp. 400–403.

[115] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Enhancing structural software coverage by incrementally computing branch executability," *Software Quality Journal*, vol. 19, no. 4, pp. 725–751, 2011.

[116] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, 2008, pp. 3–14.

[117] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: a new algorithm for property checking." in *SIGSOFT FSE*, 2006, pp. 117–127.

[118] "SMV," http://www.cs.cmu.edu/~modelcheck/smv.html.

[119] "DuaF," http://www3.nd.edu/~rsanteli/duaf/.

[120] "Testful," https://code.google.com/p/testful/.

[121] "Evosuite," http://www.evosuite.org/.

[122] M. R. Girgis, A. S. Ghiduk, and E. H. Abd-elkawy, "Automatic generation of data flow test paths using a genetic algorithm," *International Journal of Computer Applications*, vol. 89, no. 12, pp. 29–36, March 2014.

[123] "CAUT," http://www.lab205.org/caut.

[124] "BLAST," http://forge.ispras.ru/projects/blast.

[125] "CPAchecker," http://cpachecker.sosy-lab.org/.

[126] "Randoop," http://www.evosuite.org/;http://code.google.com/p/randoop/.

[127] F. G. Frankl, "The use of data flow information for the selection and evaluation of software test data," Ph.D. dissertation, University of New York, NY, October 1987.

[128] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Symposium on Testing, Analysis, and Verification*, 1991, pp. 74–86.

[129] J. R. Horgan and London, "ATAC: A data flow coverage testing tool for C," in *Proceedings of Symposium on Assessment of Quality Software Development Tools*, 1992, pp. 2–10.

[130] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa, "Demand-driven structural testing with dynamic instrumentation," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 156–165.

[131] M. Kamkar, P. Fritzson, and N. Shahmehri, "Interprocedural dynamic slicing applied to interprocedural data flow testing," in *ICSM*, 1993, pp. 386–395.

[132] M. J. Harrold, "Perforining data flow testing in parallel," in *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, 1994, pp. 200–207.

[133] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," in *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 272–281.

[134] "ATAC," http://invisible-island.net/atac/atac.html.

[135] "Coverlipse," http://sourceforge.net/projects/coverlipse/.

[136] "JaBUTi," https://code.google.com/p/jabutimetrics/.

[137] "JMockit," http://jmockit.github.io/.

[138] "BA-DUA," https://github.com/saeg/ba-dua.

[139] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *Comput. J.*, vol. 52, no. 5, pp. 589–597, 2009.

[140] P. G. Frankl and E. J. Weyuker, "A data flow testing tool," in *Proceedings of the Second Conference on Software Development Tools, Techniques, and Alternatives*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, pp. 46–53.

[141] G. Denaro, A. Gorla, and M. Pezzè, "Datec: Contextual data flow testing of java classes," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, 2009, pp. 421–422.

[142] P. G. Frankl, S. N. Weiss, and E. J. Weyuker, *Asset: A system to select and evaluate tests*. Courant Institute of Mathematical Sciences, New York University, 1985.

[143] T. J. Ostrand and E. J. Weyuker, "Data flow-based test adequacy analysis for languages with pointers," in *Proceedings of the symposium on Testing, analysis, and verification*. ACM, 1991, pp. 74–86.

[144] M. Chaim, "Poke-tool-a tool to support structural program testing based on data flow analysis," *School of Electrical and Computer gineering, University of Campinas, Campinas, SP, Brazil*, 1991.

[145] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro, "Coverage testing of java programs and components," *Science of Computer Programming*, vol. 56, no. 1, pp. 211–230, 2005.

[146] J. Misurda, J. Clause, J. Reed, B. R. Childers, and M. L. Soffa, "Jazz: A tool for demand-driven structural testing," in *Compiler Construction*. Springer, 2005, pp. 242–245.

[147] I. Bluemke and A. Rembiszewski, "Dataflow approach to testing java programs," in *Dependability of Computer Systems, 2009. DepCos-RELCOMEX'09. Fourth International Conference on*. IEEE, 2009, pp. 69–76.

[148] ——, "Dataflow testing of java programs with DFC," in *Advances in Software Engineering Techniques - 4th IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 12-14, 2009. Revised Selected Papers*, 2009, pp. 215–228.

[149] M. L. Chaim and R. P. A. de Araujo, "An efficient bitwise algorithm for intra-procedural data-flow testing coverage," *Inf. Process. Lett.*, vol. 113, no. 8, pp. 293–300, 2013.

[150] R. P. A. de Araujo and M. L. Chaim, "Data-flow testing in the large," in *IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 81–90.

[151] M. M. Hassan and J. H. Andrews, "Comparing multi-point stride coverage and dataflow coverage," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 172–181.

[152] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," 1995.

[153] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 56–66.

[154] C.-H. Liu, D. C. Kung, and P. Hsia, "Object-based data flow testing of web applications," in *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*. IEEE, 2000, pp. 7–16.

[155] Y. Qi, D. Kung, and E. Wong, "An agent-based data-flow testing approach for web applications," *Information and software technology*, vol. 48, no. 12, pp. 1159–1171, 2006.

[156] L. Mei, W. Chan, and T. Tse, "Data flow testing of service-oriented workflow applications," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 371–380.

[157] C.-W. Wang and A. Cavarra, "Checking model consistency using data-flow testing," in *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*. IEEE, 2009, pp. 414–421.

[158] J. Zhao, "Data-flow-based unit testing of aspect-oriented programs," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*. IEEE, 2003, pp. 188–197.

[159] L. Mei, W. Chan, and T. Tse, "Data flow testing of service choreography," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 151–160.

[160] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leverage of symbolic atg tools to advanced coverage criteria," *ICST2014*, 2014.

[161] R. T. Alexander, J. Offutt, and A. Stefik, "Testing coupling relationships in object-oriented programs," *Softw. Test., Verif. Reliab.*, vol. 20, no. 4, pp. 291–327, 2010.

[162] G. Denaro, M. Pezzè, and M. Vivanti, "On the right objectives of data flow testing," in *ICST*, 2014, pp. 71–80.

[163] M. Vivanti, "Dynamic data-flow testing," in *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, 2014, pp. 682–685.

[164] M. L. Chaim and R. P. A. de Araujo, "Proof of correctness of the bitwise algorithm for intra-procedural data-flow testing coverage," School of Arts, Sciences and Humanities, University of Sao Paulo, Tech. Rep. PPgSI-001/2013, 2013. [Online]. Available: http://ppgsi.each.usp.br/arquivos/RelTec/PPgSI-001_2013.pdf

[165] M. L. Chaim, A. Accioly, D. M. Beder, and M. Morandini, "Evaluating instrumentation strategies by program simulation," 2011.