A Survey on Data Flow Testing

Ting Su, East China Normal University
Ke Wu, East China Normal University
Weikai Miao, East China Normal University
Geguang Pu, East China Normal University
Jifeng He, East China Normal University
Yuting Chen, Shanghai Jiaotong University
Zhendong Su, University of California, Davis

Data flow testing (DFT) is a family of testing strategies designed to verify correct interactions between each program variable's definition and its uses. Such a test objective of interest is referred to as a *def-use pair*. The goal of DFT is to exercise every pair *w.r.t.* different testing criteria (*i.e.*, *data flow coverage criteria*). Herman introduced the concept of DFT in 1976. Since then, various data flow-based coverage criteria have been proposed. A number of studies have also been conducted, both theoretically and empirically, to analyze DFT's complexity and effectiveness. The effectiveness of DFT in fault detection has stimulated various approaches to pursue efficient and automated data flow testing.

This survey presents a detailed overview of data flow testing, including challenges and approaches in enforcing and automating it: (1) It classifies and discusses techniques for test data generation, such as random testing, collateral coverage-based testing, search-based testing, symbolic execution-based testing and model checking-based testing; (2) it discusses techniques for tracking data flow coverage; (3) it summarizes recent advances and discusses future research directions toward more practical data flow testing; and (4) it presents several DFT applications, including software fault localization, web security testing, and specification consistency checking.

Categories and Subject Descriptors: D.2.5 [**Testing and Debugging**]: Testing tools, Symbolic execution; D.2.4 [**Software/Program Verification**]: Model checking

General Terms: Algorithms, Reliability, Experimentation

Additional Key Words and Phrases: Data flow testing, Coverage criteria, Test data generation, Coverage tracking

1. INTRODUTION

Data flow testing (DFT) is a family of testing strategies, which selects paths from the program's control flow to exercise the definition-use relations *w.r.t.* 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, data flow testing focuses on the flow of data, *i.e.*, the interactions between variable definitions and their uses.

The original conception of data flow testing was introduced by Herman [Herman 1976] in 1976. He claims that data flow testing can test a program more thoroughly and reveal more subtle software bugs. Since then various slightly different notions of data flow-based coverage criteria [Rapps and Weyuker 1982; Laski and Korel 1983; Rapps and Weyuker 1985; Clarke et al. 1989; Frankl and Weyuker 1988; Harrold and Rothermel 1994] have been proposed and investigated. The main reason for this diversity lies in the different ways of exercising definition-use relations as well as different

Ting Su is partially supported by Chinese NSF grant No. 91118007; Geguang Pu is partially supported by Chinese NSF grant No. 61402178; Weikai Miao is partially supported by Chinese NSF grant No. 61361136002 and Shanghai STC Project grant 14YF1404300; Zhendong Su is partially supported by United States NSF Grants 1117603, 1319187, and 1349528. Author's addresses: Ting Su, Ke Wu, Weikai Miao, Geguang Pu and Jifeng He, Shanghai Key Laboratory of Trustworthy Computing, Software Engineering Institute, East China Normal University, Shanghai, China; Yuting Chen, Software School, Shanghai Jiaotong University, Shanghai, China; Zhendong Su, Department of Computer Science, University of California, Davis, California, USA.

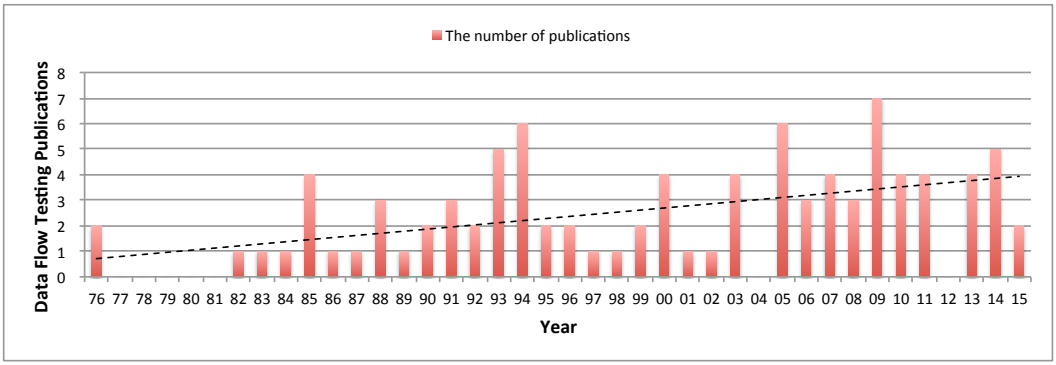


Fig. 1: Data flow testing publications from 1976 to 2015 (The dotted curve indicates the trend of increasing research interests).

adaptions in procedural and object-oriented programming languages. The effectiveness of DFT is justified by several empirical studies [Frankl and Weiss 1993; Foreman and Zweben 1993; Weyuker 1993; Hutchins et al. 1994; Frankl and Iakounenko 1998; Khannur 2011], which have shown data flow-based coverage criteria outclass control flow-based criteria (*e.g.*, statement or branch coverage). Moreover, the online software testing knowledge center organized by Khannur [Khannur 2011], reports that in practice "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 the past four decades, data flow testing has been increasingly and extensively studied (illustrated in Figure 1). Much research effort has been endeavored to achieve practical and efficient DFT. However, little work in the literature gives a deep investigation or analysis on its state-of-the-art. Edvardsson [Edvardsson 1999] presents an early survey on automated test generation and identifies several challenges in that field. Anand *et al.* [Anand et al. 2013] conduct a comprehensive survey on several widely adopted techniques in automated test data generation. But none of these techniques are discussed in the context of DFT. An introductory chapter of data flow testing can be found in many testing tutorials, *e.g.*, the books by Beizer [Beizer 1990] and by Pezzè *et al.* [Pezzè and Young 2007]. They introduce the basic conceptions and identify the challenges but have not discussed its automation.

Despite the ability of DFT in detecting data interaction faults, a big gap between real-world programs and the practicality of proposed DFT techniques still exists. Thus, we believe, for both academic researchers and industrial practitioners, it is highly desirable to review the current research state, recognize the difficulties in its enforcement, and point future research directions to narrow the gap. In order to provide a systematic overview of DFT, we start from the three basic phases of DFT, *i.e.*, 1) data-flow analysis, 2) test data generation, and 3) data flow coverage computation, and concentrate on the techniques used in the latter two steps. We do not plan to investigate deep on data-flow analysis techniques, since it is an independent research topic from DFT and has been already investigated in [Kennedy 1979; Wögerer 2005; Tok et al. 2006].

To this end, we present this first survey on data flow testing: we set up a DFT publication repository, which contains total 93 papers from 1976 to 2015. We searched the online paper repositories, and collected valid papers which contain these keywords "data flow testing", "def-use pairs", "data-flow relations", "data flow testing + analy-

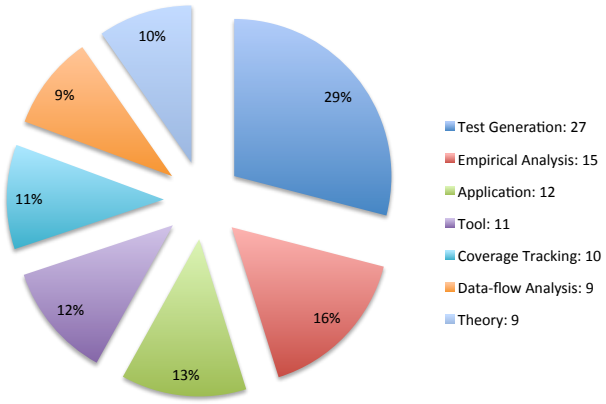


Fig. 2: Percentage of each research topic in the literation of data flow testing (total 93 papers).

sis", "data flow testing + test generation", "data flow testing + coverage" in either their titles or abstracts. Then, following this same keyword rules, we went through each reference of these papers to collect the missing publications. The repository is now available online[1]. We classify them into seven main categories:

— ***Test Data Generation*** Studies on general approaches or techniques developed to automate data flow-based test generation.
— ***Data-flow Analysis*** Studies on techniques used to analyze data flow relations (*i.e.*, def-use pairs) in the context of different programming languages and their features.
— ***Coverage Tracking (Computation)*** Studies on techniques used to track data flow coverage, *i.e.*, decide which def-use pairs are satisfied.
— ***Empirical Analysis*** Studies on analyzing the complexities in enforcing data flow testing as well as comparing its fault detection effectiveness with other coverage criteria (*e.g.*, statement, branch, mutation testing).
— ***Application*** Studies on applying data flow testing to other research fields, *e.g.*, software fault localization, web security testing, and specification consistency checking.
— ***Theory*** Studies on the fundamental theory and theoretical analysis on data-flow coverage criteria.
— ***Tool*** Studies on building, illustrating, and evaluating data flow testing tools.

Note that some papers may be involved in more than one categories; for example, a paper may present a tool and also propose a new approach to coverage computation. We assign each paper to one category according to its main objective. Therefore, our classification, to some extent, may be subjective. Nevertheless, we believe the percentage of each research topic shown in Figure 2 fairly represents the current research state in DFT.

The reminder of this survey is organized as follows. Section 2 gives an overview of DFT with an illustrative example, followed by the introduction of DFT's basic testing process and the summary of various challenges in DFT's enforcement. Section 3 investigates general approaches to DFT's test data generation, and discusses their principles, strengths and weaknesses. Section 4 surveys DFT's coverage tracking techniques and tools. Recent research advancements in DFT are discussed in Section 5, and in

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

Section 6, we propose an efficient and practical DFT framework. Section 7 presents DFT's several applications. Section 8 concludes and discusses future work.

2. OVERVIEW OF DATA FLOW TESTING

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

2.1. 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 types 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 [Herman 1976], 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, in data flow testing, two types of uses are distinguished. If the variable is used in a computational or output statement, its use is referred as a *computation use* (or *c-use*). If the variable is used in a predicate, its use is called as a *predicate use* (or *p-use*).

Definition 2.1 (*Data Flow Testing*). Given a def-use pair $du(l_d, l_u, x)$ in program P, the aim 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.

2.2. An Example

Figure 3 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 3. Figure 4 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:

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

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

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 2.1). 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 du_2 is *infeasible*: 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*.

[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 }
```

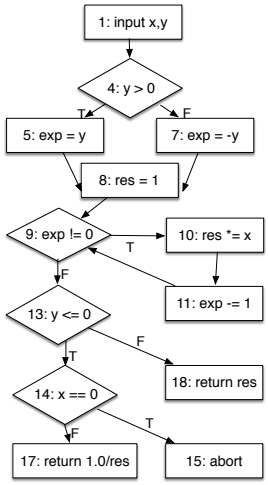


Fig. 3: An example: *power*.

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. 4: The definitions and uses of the variables in Figure 3, and their corresponding def-use pairs.

2.3. Basic Testing Process

Data flow testing consists of three basic phases, *i.e.*, data-flow analysis, test data generation and coverage tracking (illustrated in Figure 5), which totally occupy more than 50% research efforts as shown in Figure 2.

— ***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).
— ***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 Coverage Tracking Phase***: The test input t is executed against the program P for covering the pair du. If du is covered, t is incorporated into the resulting test suite T.

The whole 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 test oracles. In particular, in the data-flow analysis phase, classic reaching definition algorithms [Allen and Cocke 1976] can

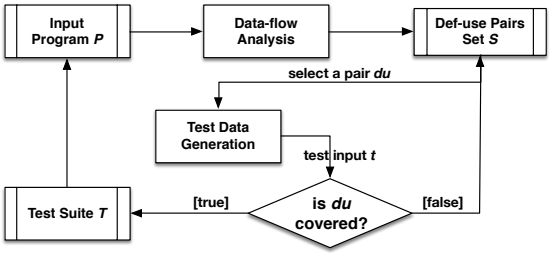


Fig. 5: The basic process of data flow testing

be used. For example, Harrold *et al.* [Harrold and Soffa 1994] use a standard iterative data-flow analysis to compute definition-use relations; Pande *et al.* [Pande et al. 1994] use an extended reaching definition analysis [Aho et al. 1986] to handle programs with single-level pointers; Chatterjee *et al.* [Chatterjee and Ryder 1999] use a flow- and context-sensitive algorithm to compute def-use pairs for object-oriented libraries; Harrold *et al.* [Harrold and Rothermel 1994] extend data-flow analysis for object-oriented languages, which considers the definition-use relations through instance variables; Souter *et al.* [Souter and Pollock 2003] and Denaro *et al.* [Denaro et al. 2008] also extend classic data-flow analysis to object-oriented programs.

2.4. Difficulties

Despite the promise of DFT in detecting data-flow faults, several difficulties [Weyuker 1990; Denaro et al. 2013] prevent it from finding wide application in industrial practice.

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 have 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 some complicate program constructs (*e.g.*, loops) and the conservativeness of data-flow analysis, test objectives consist of *feasible* pairs as well as *infeasible* ones (*e.g.*, the pair (l_8, l_{18}, res) in Section 2.2 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, on covering 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 needs to derive test cases *w.r.t.* data flow criteria,

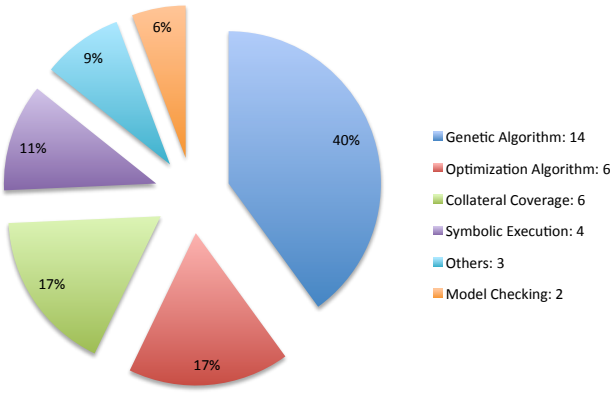


Fig. 6: Percentage of publications using each testing approach on data flow-based test data generation.

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. In spite of the undecidability of these problems, with the help of the existing techniques and recent advancements, DFT can be automated and its testing cost can be mitigated, as this survey will demonstrate.

3. APPROACHES TO TEST DATA GENERATION

This section presents various approaches to data flow-based test data generation that can be found in the present literature. From the publication repository, we found the test generation problem has been continuously concerned in academic over the past 20 years. Total 27 technical research papers are related with test data generation. We classified them into five main groups according to their testing approaches: *random testing*, *collateral coverage-based testing*, *search-based testing*, *symbolic execution-based testing*, and *model checking-based testing*.

We compute the percentages[3] of each testing approach used in these papers (showed in Figure 6). We found 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.

In the following, we will detail these approaches in independent sections. Random testing is explained in Section 3.1, collateral coverage-based testing in Section 3.2, search-based testing in Section 3.3, symbolic execution-based testing in Section 3.4, and model checking-based testing in Section 3.5. At last, some other approaches are discussed in Section 3.6.

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

3.1. Random Testing

Random testing [Bird and Munoz 1983] is one of the most widely-used and cost-effective testing approaches. 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 [Pacheco et al. 2007], and random testing is adapted to randomly generate these sequences as tests for the classes under test. Several work [Girgis 2005; Ghiduk et al. 2007; Su et al. 2015] has adopted this random testing technique as an easily-implemented but efficient approach for data flow testing.

Discussion Random testing is cost-effective and easy to implement, but it could only distinguish limited set of program behaviors. As a result, without any optimizations, random testing usually cannot achieve satisfiable data flow coverage. But with the help of some optimization techniques (*e.g.*, adaptive random testing [Chen 2008; Ciupa et al. 2008; Lin et al. 2009; Arcuri and Briand 2011] and feedback-directed random testing [Pacheco et al. 2007]), random testing may become a competitive test generation approach for DFT.

3.2. Collateral Coverage-based Approach

In software testing, *collateral coverage* has been exploited to optimize test suite generation [Harman et al. 2010; Fraser and Arcuri 2013], 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 size of the resulting test suite can be reduced as well as the cost of test case execution and test oracle checking.

Similarly, when exercising a program to satisfy a given testing criterion (*e.g.*, branch coverage), test objectives *w.r.t.* other coverage criteria (*e.g.*, data-flow coverage) may also be accidentally covered, which is another form of collateral coverage [Malevris and Yates 2006]. The reason is that one test objective may imply (or subsume) another one even when these two test objectives are derived from different testing criteria. For example, Figure 7 shows the subsumption relations between different testing coverage criteria. The criterion at the arrowtail subsumes the criterion at the arrowhead (*e.g.* the branch criterion subsumes the statement criterion). Since the subsumption relation is transitive, it actually defines the relations between various coverage criteria. The shadowed criteria are seven types of data flow testing criteria (refer to [Rapps and Weyuker 1985; Frankl and Weyuker 1988] for their detailed definitions), which emphasize different ways to exercise definition-use relations. All-uses coverage is all def-use coverage (*cf.* Definition 2.1 in Section 2.1).

The collateral coverage-based idea has been attempted to tackle data flow testing [Malevris and Yates 2006; Santelices and Harrold 2007; Merlo and Antoniol 1999; Marré and Bertolino 1996; Marré and Bertolino 2003; Santelices et al. 2006]. Malevris *et al.* [Malevris and Yates 2006] investigate the level of data flow coverage 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 data flow coverage can be modeled as a function which takes as parameters the number of selected paths *w.r.t.* branch testing and the number of feasible paths therein. In addition, they also find: 1) The data flow coverage is independent of the language that a unit uses; 2) The level of collateral coverage can be predicated a prior in estimating the possible testing budgets demanded by DFT; and 3)

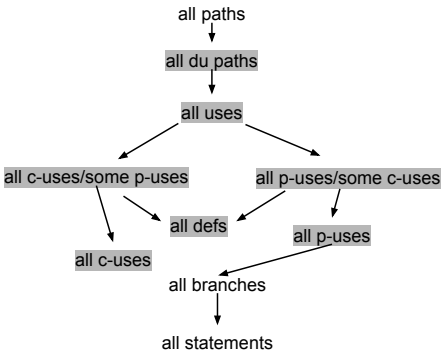


Fig. 7: Hierarchy of structural characteristics of a program.

Undertaking branch testing before data flow testing can be more cost-effective because parts of data flow-based test objectives will be covered during branch testing.

Inspired by the idea of collateral coverage, Santelices *et al.* [Santelices and Harrold 2007] 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 but not all program executions), and *non-inferable* (the coverage cannot be inferred from branch coverage). During the dynamic test suite execution stage, the branch coverage is recorded against three types of entities, *i.e.*, the definitions, the uses and the kills of def-use pairs. Finally, the coverage tracking 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.* [Merlo and Antoniol 1999] 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.* [Marré and Bertolino 1996; Marré and Bertolino 2003] 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.* [Santelices et al. 2006] present a subsumption algorithm for program entities of any type (*e.g.*, branches, def-use pairs, and call sequences) based on predicate conditions. This predicate condition is a special version of path condition [Robschink and Snelting 2002], computed from the system dependence graph, to represent the necessary but not sufficient condition of an entity for its coverage. A table which includes all these predicate conditions for each entity are constructed to create the subsumption relations of entities for efficient coverage tracking.

Discussion With the help of existing test data, the collateral coverage-based approach has several merits for data flow testing: 1) By only considering the unconstrained pairs, it can reduce the test suite size as well as the overhead of coverage tracking; 2) How many testing budgets should be allocated can be estimated through the number of unconstrained pairs; and 3) It can help understand the relationships between entities at different levels (*e.g.*, statements, branches, and def-use pairs) in the program.

However, since this approach uses the coverage of low level program entities (*e.g.*, statements or branches) to infer the coverage of high level entities (*e.g.*, def-use pairs), it may fail to identify test data for those pairs whose coverage cannot be easily inferred. Moreover, it cannot distinguish infeasible pairs either.

3.3. Search-based Testing Approach

In recent years, the adoption of the search-based approach [McMinn 2004] for automated 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).

Principle of Search-based Approach The search-based approach includes various metaheuristic search techniques [McMinn 2004], *e.g.*, hill climbing, simulated annealing, and evolutionary algorithms. They are high-level frameworks, and utilize heuristic strategies to identify solutions to such combinational problems as test case generation. 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).

In general, three basic decisions should be made in adapting a metaheursitic search technique to a concrete problem. First, the problem solutions should be *encoded* in a suitable form such that they can be efficiently manipulated during the search. Second, a *fitness or objective 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. Finally, a search procedure should be determined which formalizes the whole search algorithm.

Genetic Algorithm-based Search The Genetic Algorithm (GA) [Holland 1992] proposed in 1975 is a representative of metaheuristic search techniques, which is inspired by genetics and natural selection. During test data generation, a GA starts from a population of candidate individuals (*i.e.*, test cases) and then uses 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 independent individuals produces two new test cases which share genetic material from parents, while mutation adds small changes to a 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 [Girgis 2005; Ghiduk et al. 2007; Vivanti et al. 2013]. Girgis [Girgis 2005] 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 [Michalewicz 1994] to encode the program test input (*i.e.*, test case). In [Girgis 2005], the GA uses the ration between the number of the covered def-use paths and the total number of def-use paths as the fitness function. Actually, this fitness function exclusively uses coverage information to determine the effectiveness of

an individual test data. This GA-based method works as follows. First, it generates a set of test cases encoded in the binary string form. Then it uses two methods (a random selection and a roulette wheel algorithm [Michalewicz 1994]-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 have covered the def-use paths and the remaining uncovered def-use pairs.

Ghiduk *et al.* [Ghiduk et al. 2007] later find there are some pitfalls inside the fitness function in [Girgis 2005]. For example, in the following situations, the fitness function fails to identify the closeness of the test cases: 1) If two test cases cover the same number of def-use paths, they will be given the same fitness value; and 2) If a test case do not cover any def-use paths, it will be given '0' as its fitness value. As a result, it may lose useful information when selecting promising individuals for recombination. To solve this problem, following the similar procedure in [Girgis 2005], they propose a new multi-objective fitness function. This function evaluates the fitness of test data based on its relation, through dominance [Lengauer and Tarjan 1979], 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; and 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.* [Vivanti et al. 2013] 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 [Tonella 2004]. Following the conception of testing on classes [Harrold and Rothermel 1994], 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 [Wegener et al. 2001], where the search is first guided toward reaching the first node (*i.e.*, the *def* node), and then from there 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. Moreover, for infeasible test objectives, testing resources invested on them will be wasted. To overcome these problems, instead of using the classic way of targeting one pair at one time, they apply the whole test suite generation [Fraser and Arcuri 2013] in data flow testing, which optimizes sets of test cases toward covering all test objectives. This approach is expected to be less affected by infeasible test objectives. Through the evaluation on SF100 corpus of classes [Fraser and Arcuri 2012], they confirm that 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.* [Denaro et al. 2015] also use a similar genetic algorithm to augment initial test suites with data flow-based test data in object-oriented systems. Liaskos *et al.* [Liaskos et al. 2007; Liaskos and Roper 2008] hybridize GA with the artificial immune systems [Liaskos and Roper 2007] (AIS) algorithm to fulfill data flow testing against 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 [Baresi et al. 2010; Baresi and Miraz 2010], for structural testing on Java classes. This GA variant uses a multi-objective fitness function and works at class level as well as method level. The former

generates useful states for class objects, and the latter use them to reach the uncovered code in the class. In [Miraz 2010], 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 relate methods that cooperate with each other by exchanging data (*e.g.*, objects' fields). Other efforts include [Oster 2005; Deng et al. 2009], which also use GA to automate data flow testing but are only evaluated 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.* [Nayak and Mohapatra 2010] and Singla *et al.* [Singla et al. 2011a; Singla et al. 2011b] use particle swarm optimization in data flow testing. Ghiduk [Ghiduk 2010] uses ant colony optimization to fulfill data flow testing. Inspired by natural behaviors, these optimization algorithms simulate these behaviors to find optimal solutions in the context of DFT. However, these approaches have been only evaluated on toy programs. Their effectivenesses on large programs are still unknown.

Discussion The search-based techniques have already been successfully applied to enforce simple coverage criteria, *e.g.*, statement and branch testing [Anand et al. 2013]. For advanced coverage criteria, *e.g.*, logical coverage [Ammann et al. 2003; Inc 1992] and data flow coverage criteria, it has also shown great potential [Lakhotia et al. 2007; Awedikian et al. 2009; Ghani and Clark 2009]. It is easy to implement and efficient to execute. Moreover, this approach treats test data generation as a domain search problem, and thus, it is more competent at solving non-linear constraints and finding floating-point inputs [Lakhotia et al. 2009; Lakhotia et al. 2010; Bagnara et al. 2013] than those constraints solving-based approaches (*e.g.*, symbolic execution).

But the testing performance of search-based techniques heavily depends on the underlying fitness functions, and thus requires carefulness in its design and optimization. Moreover, although the multi-goal fitness functions can mitigate the impact of infeasible pairs, it cannot be employed to detect infeasible pairs.

3.4. Symbolic Execution-based Approach

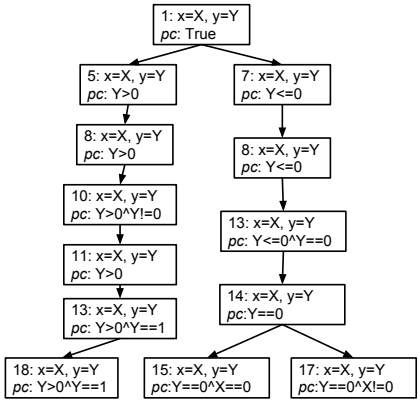
Symbolic execution is a classic program analysis technique, which was proposed by King [King 1976] in 1976. Recent impressive progress in constraint solvers (*e.g.*, Z3 [de Moura and Bjørner 2008], STP [Ganesh and Dill 2007], Yices [Dutertre 2014]) as well as the combination of both concrete and symbolic execution (referred as *dynamic symbolic execution* or *concolic testing* [Godefroid et al. 2005; Sen et al. 2005]) makes it possible to perform automated path-based testing on real-world programs. Many symbolic execution-based testing tools [Godefroid et al. 2005; Sen et al. 2005; Burnim and Sen 2008; Cadar et al. 2008; Tillmann and de Halleux 2008; Godefroid et al. 2012; Anand et al. 2007; Wang et al. 2009] bloomed up from both academic and commercial areas.

Principle of Symbolic Execution Symbolic execution is known as a classic program analysis technique, which uses symbolic values instead of concrete values as program inputs. As a result, the symbolic expressions composed of these inputs can be used to represent the values of program variables. During the process of symbolic execution, at any point, a program state includes: 1) the symbolic expressions (values) of program variables; 2) a *path constraint* (*pc*) over symbolic inputs in the form of a boolean formula that needs to be satisfied to reach this program point; and 3) a program counter which denotes the next program statement to execute.

The technique works as follows: During the execution, new constraints over inputs at each branch point are used to update *pc*. If the new *pc* is unsatisfiable, the exploration of the corresponding path will stop. Otherwise, the execution will continue along

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) the test inputs and the path constraints *w.r.t.* different program paths.



(b) the corresponding symbolic execution tree

Fig. 8: The symbolic execution process on Function *power* in Figure 3.

this branch point such that any solution of the *pc* will execute 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. A search strategy [Cadar et al. 2008; Burnim and Sen 2008; Cadar et al. 2006] will 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 3 (Section 2). Here, three program paths are explored and the test inputs are generated by solving the collected path constraints (as Figure 8(a) shows). The execution tree is given in Figure 8(b).

Static Symbolic Execution-based Approach Girgis [Girgis 1993] 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-subset* 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 cover def-use pairs of interest. In this system, a tester 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 feasible paths, this system can produce a test suite which fulfills the given data flow testing criterion.

For the example program in Figure 3, 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 $dua(l_{10}, l_{18}, res)$. The solution $(x \mapsto 0, y \mapsto 1)$ to its corresponding *pc*, (*i.e.*, $y == 1 \wedge y > 0$) can satisfy the pair.

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. The ability of symbolic execution to find more feasible paths is limited by the fact that solving the general class of constraints is undecidable [Anand et al. 2013].

To counter this problem, Godefroid and Koushik *et al.* [Godefroid et al. 2005; Sen et al. 2005] interleave the symbolic execution with concrete execution (*i.e.*, *dynamic* symbolic execution (DSE)) to systematically explore program paths. 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 becomes 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.

Su *et al.* [Su et al. 2015] first adapt this dynamic symbolic execution technique to conduct data flow testing on top of a DSE engine, called CAUT [Wang et al. 2009; Yu et al. 2011; Sun et al. 2009; Su et al. 2014]. 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 [Zamfir and Candea 2010; Ma et al. 2011]) and a redefinition path pruning technique (no redefinitions should appear on the sub-path between the *def* and the *use*).

For the example program in Figure 3, 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

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

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 [Burnim and Sen 2008]) 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

[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.

make the exploration very slow. In [Su et al. 2015], two techniques 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* 2.1). 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.

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

Second, a *cut point-guided search strategy* [Su et al. 2015] 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 by T or F to represent the *true* or *false* branch direction). Because $9F$ is closer to cut point l_{13} than $4F$, $9F$ is flipped. As a result, a new test input $t = (x \mapsto 0, y \mapsto 0)$ can be generated which 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 classic symbolic execution is a path-based testing approach, which can systematically explore paths to cover target def-use pairs. In the early work [Girgis 1993], Girgis uses a control-flow criterion as a coverage metric to guide path exploration, which can mitigate the path explosion problem but may run the risk of failing to cover 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. Moreover, the classic symbolic execution has to make some approximations when symbolic reasoning, which may lose precision in data flow testing. For example, it cannot precisely reason about which concrete element is referred by $a[x]$ (a is an array and x is an index variable) when the concrete value of x is unknown. One way is to treat $a[x]$ as a use of the whole array a.

In contrast, the *dynamic* symbolic execution-based approach can be more efficient and precise. For example, variable redefinitions caused by aliases can be detected more easily and precisely with the dynamic execution information.

However, symbolic execution-based testing is still incapable of identifying infeasible pairs. Because the DSE-based or SSE-based approach is an explicit path-based testing approach, it cannot draw a conclusion on the feasibility of a pair until all program paths are explored. 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, in covering an infeasible pair.

3.5. Model Checking-based Approach

Principle of Model Checking Model checking [Clarke et al. 1999] is known as a classic formal verification approach, which includes various techniques (*e.g.*, *explicit model checking* [Lichtenstein and Pnueli 1985], *symbolic model checking* [McMillan 1992] and *bounded model checking* [Biere et al. 1999]). A model checker is able to construct witnesses or find counterexamples when property checking.

At a high level, a model checker takes as input the system specification. Meanwhile, a temporal logic formula, expressed in some specification languages, *e.g.*, LTL (Linear Temporal Logic [Pnueli 1977]) or CTL (Computation Tree Logic [Clarke and Emerson 1981]), is adopted to describe some safety properties of interest. Then the model checker searches the entire state space of the system and check whether the property is violated or not. If the property is violated at some state, then a counter-example will be constructed 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 [Fraser et al. 2009] especially when those counter-examples are interpreted as test cases, which can help a human analyst to identify and fix the fault.

WCTL-based Model Checking In classic model checking, the verification task usually works on an abstract model, the 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, *i.e.*, for each $s \in S$ there is an $s' \in S$ such that $(s, s') \in T$; 4) $L : S \to 2^{AP}$ is a labeling function, which maps s to a set of atomic propositions that hold in s.

Based on the Kripke structure, CTL formulas can be used to express temporal properties of interest. Here we give a simple introduction to CTL (see details in [Clarke and Emerson 1981]): CTL formulas are composed of path qualifiers (*e.g.*, **A** stands for *all paths*, **E** for *some path*), modal operators (*e.g.*, **X** stands for *next time*), **F** for *eventually*, **G** for *always*, and **U** for *until*), and logical operators. For a CTL formula f and a state s of Kripke structure M, $K, q \models f$ if q satisfies f (or briefly written as $q \models f$). A CTL formula f is called as a WCTL formula if 1) f only has temporal operators **EX**, **EF** and **EU**; and 2) in each sub-formula of f (f=$f_1 \wedge f_2 \wedge \cdots \wedge f_n$), at most one conjunct f_i is an atomic proposition.

Hong *et al.* [Hong et al. 2003; Hong and Ural 2005] first use such a Kripke structure-based model checking approach to perform data flow testing via a CTL-based model checker. The test obligations of def-use pairs are expressed in 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 identifying witnesses for a set of logical formulas. In particular, they denote the flow graph G of the program under test as $G = (V, v_s, v_f, A)$ where V is the vertices set, $v_s \in V$ is the start vertex, $v_f \in V$ is the final vertex, and A is a finite arcs set. Here, a vertex represents a statement and an arc denotes the control flow between two statements. $DEF(v)$ denotes the variables set that is defined at the vertex v, while $USE(v)$ denotes the variables set that is used at the vertex v. As a result, the flow graph G is viewed 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 3. We use d_l^x or u_l^x to represent the variable x 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 [Hong et al. 2003], it characterizes 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 ensures 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 only when the Kripke structure derived from the data flow graph satisfies the formula in (5). Note in this approach, it is not necessary to know in advance whether

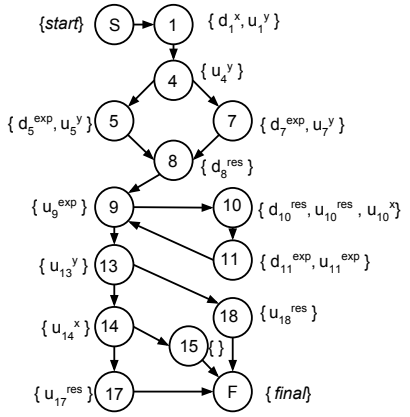


Fig. 9: The Data Flow Graph for the Function *power* in Figure 3.

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 [Hong et al. 2000; Ural et al. 2000]. 2) This approach casts the data flow testing problem into the model model checking problem, which can benefit from future 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 count of formulas can be quadratic *w.r.t.* 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 cannot 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 [Ball and Rajamani 2002; Henzinger et al. 2002; Chaki et al. 2003], was proposed in 2002. Since then, it was applied to automatically check safety properties of OS device drivers [Ball and Rajamani 2002; Beyer et al. 2007; Beyer and Keremoglu 2011] as well as generate structural test cases [Beyer et al. 2004] (*e.g.*, statement or branch coverage). Several state-of-the-art CEGAR-based model checkers, have also been implemented, *e.g.*, SLAM [Ball and Rajamani 2002], BLAST [Beyer et al. 2007] and

CPAchecker [Beyer and Keremoglu 2011]. 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. 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 the initial location, X a set of program variables, Ops a set of operations on X, and $\rightarrow \subseteq (Q \times Ops \times Q)$ a finite set of edges labeled with operations. The operation set Ops contains 1) *basic blocks* of instructions, 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 [Necula et al. 2002; Lattner 2002].

In [Beyer et al. 2004], Beyer *et al.* suggest a CEGAR-based two-phases approach, *i.e.*, *model checking* and *tests from counterexamples*, to automatically generating structural test cases. It 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 lazy abstraction is enforced [Henzinger et al. 2002]. It starts from a set of abstraction predicates in CFA. If a path that reaches 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 consult an underlying theorem prover to suggest additional abstraction predicates, which will be used to refine the original program state (*i.e.*, 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, and solve the constraint to get the corresponding test case. Subsequently, this approach can generate test cases without false positives.

Su *et al.* [Su et al. 2015] 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 proposed to directly encode the test requirement into the program under test. It instruments the original program P to P' and reduces 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 *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 budget, and reports the result as *unknown*).

For the example program in Figure 3 and the two pairs du_1 and du_2 in (1) and (2), the transformed program encoded with these two test requirements are shown in Figure 10a and Figure 10b, respectively. For the pair $du_1(l_8, l_{17}, res)$, Figure 10a 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

```
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 }
```

(a) $du_1(l_8, l_{17}, res)$

```
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 }
```

(b) $du_2(l_8, l_{18}, res)$

Fig. 10: The transformed function *power* for the def-use pair $du_1(l_8, l_{17}, res)$ in (a) and $du_2(l_8, l_{18}, res)$ in (b). The encoded test requirements are showed by the highlighted statements.

definitions on variable res at l_{10}. Before the *use*, a checkpoint is set to verify whether *cover_flag* can be *true* at l_{14}. If the checkpoint is unreachable, this pair can be proved as 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 10b, the model checker can quickly conclude no paths can reach the check point and witness the truth of *cover_flag*. Thus du_2 is infeasible.

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. Due to the conservativeness of static data-flow analysis, test objectives contain infeasible def-use pairs. This CEGAR-based approach can easily detect and eliminate infeasible pairs without false positives, which can save much testing effort as well. Moreover, this approach can also generate counter-examples (*i.e.*, test cases) for feasible def-use pairs. The technique itself can even benefit from the future 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 easier to be implemented than the other model checking-based approaches.

However, this CEGAR-based approach may also have performance limitations: This approach is essentially a *static* approach, its actual testing performance may not be as high as other *dynamic* testing approaches (*e.g.*, dynamic symbolic execution-based approach [Su et al. 2015]) when generating test cases for feasible pairs.

3.6. Other Approaches

Khamis *et al.* [Khamis et al. 2011] enhance the Dynamic Domain Reduction procedure [Offutt et al. 1999] (DRR) to perform data flow testing for Pascal programs. The DDR technique basically integrates the ideas of symbolic execution and constraint-based testing. 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 adopted to 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 [Khamis et al. 2011] 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.* [Buy et al. 2000] 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.* [Martena et al. 2002] extend this technique from single class to multiple classes; that is, testing the interclass interactions. It incrementally generates test cases from simple classes to more complicated classes.

Baluda *et al.* [Baluda et al. 2010; Baluda 2011; Baluda et al. 2011] 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 algorithm [Beckman et al. 2008; Gulavani et al. 2006], which aims to either prove a faulty statement is unreachable, or produce a test case that executes the statement. Baluda *et al.* adapt this algorithm for structural testing, and enhance it with "coarsening" to improve its scalability. In [Baluda 2011], 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.

Summary Despite several challenges in identifying data flow-based test data, researchers have developed various general approaches to automating this process. The search-based testing and collateral coverage-based testing are the two most popular techniques used to automate test data generation (as shown in Figure 6). A reasonable explanation is that these two approaches are easy to be applied in DFT. The symbolic execution-based and model checking-based techniques attracted attentions quite early, but until recently did they find practical adoptions. The fact that symbolic execution and model checking techniques are more difficult to implement than other approaches may explain this phenomenon. Additionally, for both procedural language (*e.g.*, C) and object-oriented language (*e.g.*, Java), several data flow testing tools exist. However, according to our investigation, none of these tools are commercial tools. More efforts are needed to develop efficient and easy-to-implement techniques.

4. APPROACHES TO COVERAGE TRACKING

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

4.1. 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 [Frankl 1987; Ostrand and Weyuker 1991a; Horgan and London 1992; Misurda et al. 2005b; Santelices and Harrold 2007; Harrold and Soffa 1994] have been developed to track the coverage of def-use pairs.

Table I: A Summarization of Data Flow Coverage Tools. "-" means *unknown*. "*"means the tool is publicly available.

Tool	Language	Coverage	Infrastructure	Technique
ATAC*	C/C++	Intra	A Yacc-based Parser	last definition
Coverlipse*	Java	Intra	Eclipse	path recording
DaTeC	Java	Intra/Inter	Soot	-
DuaF*	Java	Intra/Inter	Soot	coverage inference
ASSET	Pascal	Intra	-	automata
TACTIC	C	Intra	-	memory tracking
POKE-TOOL	C	Intra	-	automata
JaBUTi*	Bytecode	Intra/Inter	A Bytecode Tool	last definition
JMockit*	Java	Inter	ASM	-
Jazz	Java	Intra	Eclipse, Jikes RVM	demand-driven
DFC*	Java	Intra	Eclipse	-
BA-DUA*	Java	Intra	ASM	bitwise algorithm

Frankl [Frankl 1987] proposes a *deterministic finite automata*-based approach to track the coverage status of def-use pairs. In his approach, a pair du is related with a regular expression which describes the 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 has to do special handling when the procedure under test recursively calls itself. Ostrand *et al.* [Ostrand and Weyuker 1991a] use a *memory tracking* technique to precisely determine which pairs are covered, while Kamkar [Kamkar et al. 1993] use *dynamic slicing* to improve coverage precision.

Hogan and London [Horgan and London 1992] 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 that uses this defined variable is executed, the last definition of this variable is verified and the pair is set as covered.

Misurda *et al.* [Misurda et al. 2005b] propose a *demand-driven* strategy to track def-use pair coverage, which is aimed to improve the performance of the static instrumentation approach [Horgan and London 1992]. The approach works as follows: First, all variable definitions in a test region are identified and seed probes are inserted at their locations; 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 [Santelices and Harrold 2007] develop an efficient *matrix-based* strategy to directly track data flow coverage. In this strategy, a *coverage matrix* is created, and initialized as zeros, in which each column represents a variable use (associated with a use ID), and each cell in a column records a definition (associated with a definition ID) for that use (*i.e.*, a cell corresponds to a def-use pair). In this structure, the matrix cell can be quickly accessed through the use ID and the definition ID, to reduce the runtime cost of probes. At runtime, the probes track the last definition of a variable. At each use, a probe is inserted, which uses the use ID and the last definition ID to update the coverage status of the pair in the matrix.

In [Santelices and Harrold 2007], a novel *coverage inference* strategy is designed, 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. Details can be referred from the collateral coverage-based testing approach in Section 3.2.

In order to make coverage tracking more scalable, Harrold [Harrold 1994] develops a technique on multi-processor systems to accept tests and produces parallelizable coverage tracking workload. The workload can be statically or dynamically scheduled onto different platforms. The evaluation on a multiprocessor system shows a good speedup over the uniprocessor system.

Some work exploits dynamic data flow analysis [Harrold and Malloy 1992; Su et al. 2015] to improve the precision of tracking data flow coverage.

4.2. Coverage Tools

There have been lots of robust coverage tools [Yang et al. 2009] at hand for statement and branch coverage, but only a few are available for data flow coverage. Table I summarizes the coverage tools for data flow testing, including ASSET [Frankl and Weyuker 1985; Frankl et al. 1985; Frankl 1987; Frankl and Weyuker 1988] (the first data flow coverage tool), ATAC [Horgan and London 1992], Coverlipse, DaTec [Denaro et al. 2008; Denaro et al. 2009], DuaF [Santelices and Harrold 2007], TACTIC [Ostrand and Weyuker 1991b], POKE-TOOL [Chaim 1991], JaBUTi [Vincenzi et al. 2005], JMockit, Jazz [Misurda et al. 2005a], DFC [Bluemke and Rembiszewski 2009; Bluemke and Rembiszewski 2012], BA-DUA [Chaim and de Araujo 2013a; de Araujo and Chaim 2014]. 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. Six out of total twelve tools are publicly available, but none of them are commercial tools, which is also reported by Hassan *et al.* [Hassan and Andrews 2013] and Araujo *et al.* [de Araujo and Chaim 2014] recently.

5. 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.* [Hassan and Andrews 2013] introduce a new family of coverage criteria, called *Multi-Point Stride Coverage* (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 branch coverage, can reach a similar or higher level of accuracy than all def-use coverage when measuring test effectiveness. And the instrumentation for MPSC coverage is also more efficient than that for data flow coverage.

Bardin *et al.* [Bardin et al. 2014] 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 a more general coverage criterion to express data flow criteria, such that a symbolic executor can be directly used as a black-box tool to facilitate testing automation.

Alexander *et al.* [Alexander et al. 2010] extend the classic data flow criteria to test and analyze the polymorphic relationships in object-oriented systems. The new cover-

age 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.* [Denaro et al. 2014; Vivanti 2014] investigate the limits of the traditional static data-flow analysis used in DFT. They use a *dynamic data-flow analysis* technique 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 are missed by the traditional static data flow analysis, which undermines the effectiveness of the previous DFT approaches. This dynamic data flow analysis technique sheds light on a new direction of data flow testing which can better encompass data flow-based test objectives.

In [Denaro et al. 2015], the authors adapt this dynamic data-flow analysis technique to test object-oriented systems. This approach does not compute all the pairs a priori, but runs some tests with dynamic analysis, merges the traces to infer never executed pairs, generates new tests to cover them and then iterates until it cannot find anything new. It results in an increment around 30% over the mutation score of existing branch coverage test suites.

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 [Misurda et al. 2005b; Santelices and Harrold 2007] have been proposed to tackle this problem, which are based on expensive computations and data structures.

Inspired by classic solution to data flow problems (*e.g.*, *reaching definition* [Aho et al. 1986]), Chaim and Araujo [Chaim and de Araujo 2013a; de Araujo and Chaim 2014] 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 (or block), which is used to determine the coverage of pairs. The BA code tracks three working sets, *i.e.*, the *alive* pairs, the *covered* pairs, and the current *sleepy* 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 [Chaim and de Araujo 2013b] and the theoretical analysis, which show their algorithm demands less memory and execution time than the previous demand-driven and matrix-based approaches [Misurda et al. 2005b; Santelices and Harrold 2007]. In their evaluation [Chaim and de Araujo 2013a], this conclusion is further corroborated by simulating these instrumentation strategies [Chaim et al. 2011]. In [de Araujo and Chaim 2014], this approach is 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.

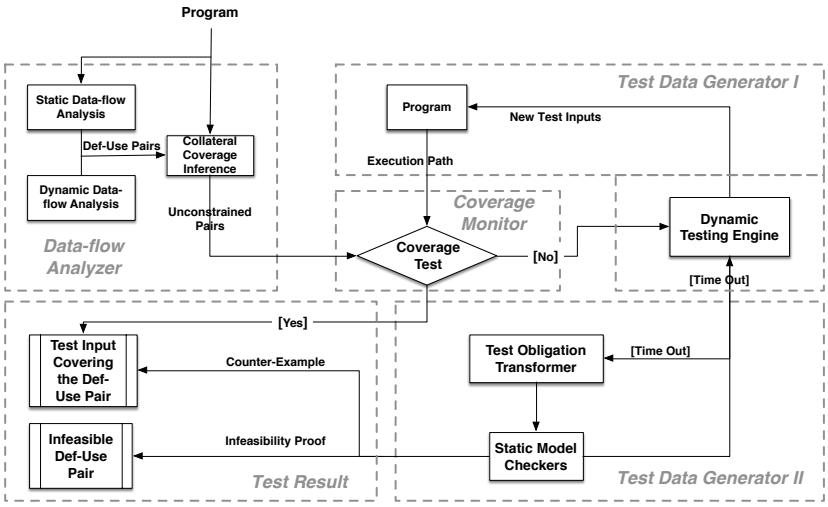


Fig. 11: The Hybrid Data Flow Testing Framework.

6. A HYBRID DATA FLOW TESTING FRAMEWORK

Data flow testing exercises programs more thoroughly, but also brings several challenges in its enforcement. Inspired by various approaches, techniques and recent advancement which have been surveyed above, we propose a new hybrid data flow testing framework (shown in Figure 11). This framework intends to combine the strengths of existing approaches to counter their respective weaknesses or limitations, which hopefully can achieve more practical data flow testing. Following the basic testing process of DFT, the framework consists of three basic components, *i.e.*, a data-flow analyzer, a test data generator, and a coverage monitor, as explained as follows:

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 [Denaro et al. 2014; Denaro et al. 2015] 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 can employ 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 is that these dynamic testing approaches can only identify feasible test objectives but incapable of dealing with infeasible ones; 2) The model checking-based approaches are used to handle the remaining pairs (including infeasible ones) when the allocated testing budgets (*e.g.*, testing time) are used up. 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 detected by this verification-based approach. For infeasible pairs, the infeasibility

proofs are provided, which prevent testing time from being wasted. If there were still uncovered pairs after 1) and 2), more testing budgets could be invested. It can restart another round of the hybrid 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 [de Araujo and Chaim 2014] can be employed.

Given a program as input, this hybrid framework: 1) *outputs test data for feasible test objectives* and 2) *eliminates infeasible test objectives*. It can achieve better performance by combining the strengths from its component approaches, and benefit from the future advancements in data-flow analysis, test data generation, and coverage tracking.

7. APPLICATIONS

Data Flow Testing has not only empirically demonstrated its effectiveness [Frankl and Weiss 1993; Hutchins et al. 1994; Khannur 2011] in revealing software bugs, but 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.

7.1. Software Fault Localization

Software fault localization is a tedious as well as time-consuming activity in program debugging to locate program errors and bugs. Agrawal *et al.* [Agrawal et al. 1995] propose a novel method which combines DFT and execution slices together to achieve more efficient fault localization. Their work is based on 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, testers can focus the statements in the failed slice. A data flow testing tool ATAC [Horgan and London 1992] 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.* [Santelices et al. 2009] 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 against 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 by different coverage types, but the combination of these different coverage types can achieve the overall best performance.

7.2. 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 complicated. Several efforts has been devoted to data flow testing against web applications.

Since the data in web applications can be stored in HTML documents, it could affect the data interactions between the server and the client. Liu *et al.* [Liu et al. 2000] 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 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 mod-

els is composed of attributes and operations to store the fundamental information. The DFT structural model uses four flow graphs to capture the relevant data flow information. After obtaining the data flow information, the 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.* [Qi et al. 2006] 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. The whole task of data-flow testing can be divided into subtasks and performed by these test agents.

Mei *et al.* [Mei et al. 2008] 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 the *XPath Rewriting Graph* as a data structure to model the XPath in WS-BPEL. And then they conceptually determine the def-use pairs in the XRG and propose a set of data flow testing criteria to test WS-BPEL applications.

7.3. 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.* [Wang and Cavarra 2009] propose a DFT-based approach to check requirement model consistency. 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, and 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) [Ural et al. 2000] and statecharts [Hong et al. 2000]. The resulting test suites provide the capability to test whether the implementation is in accordance with high-level specification models.

7.4. Other Applications

DFT has also been applied to test other programs or applications. Zhao [Zhao 2003] use DFT to test aspect-oriented programs. Harrold *et al.* [Harrold and Malloy 1992] use DFT to check parallelized code. DFT has also been applied to test object-oriented libraries [Chatterjee and Ryder 1999] and service choreography [Mei et al. 2009].

Additionally, we investigate the percentage of each language which data flow testing has been applied to (showed in Figure 12). We get several observations: 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 many subtle faults when checking object states.

8. CONCLUSION AND FUTURE WORK

In the last forty years, data flow testing has been increasingly and extensively studied. Various approaches and techniques have been developed to pursue efficient and

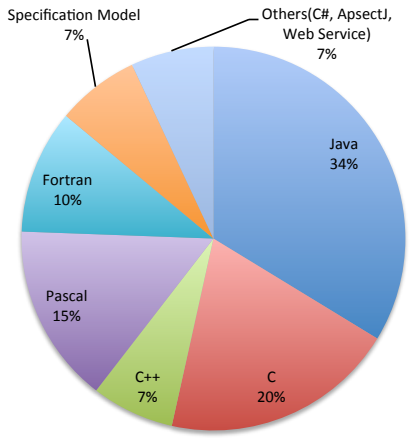


Fig. 12: Percentage of each language used in data flow testing.

automated DFT, given its ability of checking data interactions. The research effort has been endeavored in many directions, including data-flow analysis, test data generation, coverage tracking and empirical analysis. Many data flow testing tools have also been built and evaluated.

This survey makes four main contributions: 1) To our knowledge, this is the first systematic survey for data flow testing. We have constructed a publication repository with 93 research papers, demonstrated the current state of research, and provided comprehensive analysis in this field. 2) We have classified data flow-based test generation approaches into five categories. For each category, we have explained its technical principle, and have discussed the strengths and weaknesses. 3) Based on the above investigation, we have proposed a conceptional DFT framework, which combines the strengths of existing approaches. This framework is hopefully to be more efficient and practical. 4) According to the survey and other knowledge, we recommend the following research directions for future work:

— New data-flow analysis techniques can be developed to efficiently capture def-use relations. Specially, the static data-flow analysis can be complemented with the dynamic data-flow analysis to benefit each other in precision and scalability.
— Since dynamic test data generation techniques usually cannot identify infeasible test objectives, it is beneficial to complement them with some verification-based techniques, in that they can weed out infeasible def-use pairs. By sharing some runtime information, the symbolic execution and model checking techniques can be deeply combined to tackle the path explosion problem.
— Efficient coverage tracking algorithms can be proposed, which can improve the usability of data flow testing on large real-world systems.
— New cost-effective coverage criteria can be proposed to complement data flow coverage criteria. The new criterion should be easy to enforce, as well as has comparable fault detection ability against DFT, *e.g.*, [Hassan and Andrews 2013; Li et al. 2013]. Data flow coverage criteria can also be extended to various testing scenarios, *e.g.*, object-oriented systems, web applications and mobile apps, to check the correctness of data manipulations.
— A robust data flow testing framework can be built, which can facilitate DFT research from two aspects. One is the evaluation and comparison between different testing

techniques on a more fair basis. The other is the enforcement of data flow coverage testing on more real-world programs to gain deeper understanding of its effectiveness and complexities [Namin and Andrews 2009; Inozemtseva and Holmes 2014].

ACKNOWLEDGMENTS

The authors would like to thank several researchers and practitioners in data flow testing, who kindly provided very helpful comments on an earlier draft of this survey. They are Ilona Bluemke, Mattia Vivanti, Giovanni Denaro, Phyllis Frankl, Jamie Andrews, and Roberto Araujo. We are grateful to their time and expertise. We also thank for Kaixiang Chen help us do proof-reading.

REFERENCES

Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault Localization using Execution Slices and Dataflow Tests. (1995).

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Roger T. Alexander, Jeff Offutt, and Andreas Stefik. 2010. Testing coupling relationships in object-oriented programs. *Softw. Test., Verif. Reliab.* 20, 4 (2010), 291–327.

Frances E. Allen and John Cocke. 1976. A Program Data Flow Analysis Procedure. *Commun. ACM* 19, 3 (1976), 137–147.

Paul Ammann, A. Jefferson Offutt, and Hong Huang. 2003. Coverage Criteria for Logical Expressions. In *International Symposium on Software Reliability Engineering*. 99–107.

Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2007. 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*. 134–138.

Andrea Arcuri and Lionel C. Briand. 2011. Adaptive random testing: an illusion of effectiveness?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 265–275.

Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. 2009. MC/DC automatic test input data generation. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*. 1657–1664.

Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. 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*. 1–10.

Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: Debugging system software via static analysis. In *POPL*. 1–3.

Mauro Baluda. 2011. 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*. 400–403.

Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2010. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test (AST '10)*. ACM, New York, NY, USA, 59–66.

Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2011. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal* 19, 4 (2011), 725–751.

Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. 2014. Efficient Leverage of Symbolic ATG Tools to Advanced Coverage Criteria. *ICST2014* (2014).

Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. 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*. 185–194.

Luciano Baresi and Matteo Miraz. 2010. 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*. 281–284.

Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. 2008. 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*. 3–14.

B. Beizer. 1990. *Software Testing Techniques*. International Thomson Computer Press.

Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating Tests from Counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 326–335.

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.* 9, 5 (Oct. 2007), 505–525.

Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV*. 184–190.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. 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*. 193–207.

David L. Bird and Carlos Urias Munoz. 1983. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal* 22, 3 (1983), 229–245.

Ilona Bluemke and Artur Rembiszewski. 2009. Dataflow approach to testing Java programs. In *Dependability of Computer Systems, 2009. DepCos-RELCOMEX'09. Fourth International Conference on*. IEEE, 69–76.

Ilona Bluemke and Artur Rembiszewski. 2012. Dataflow Testing of Java Programs with DFC. *Advances in Software Engineering Techniques, LNCS* 7054, 3 (2012), 215–228.

Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *ASE*. 443–446.

Ugo A. Buy, Alessandro Orso, and Mauro Pezzè. 2000. Automated Testing of Classes.. In *ISSTA*. 39–48.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*. 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. 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*. 322–335.

ML Chaim. 1991. 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).

Marcos L. Chaim, Anthony Accioly, Delano Medeiros Beder, and Marcelo Morandini. 2011. Evaluating instrumentation strategies by program simulation. (2011).

Marcos Lordello Chaim and Roberto Paulo Andrioli de Araujo. 2013a. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Inf. Process. Lett.* 113, 8 (2013), 293–300.

Marcos Lordello Chaim and Roberto Paulo Andrioli de Araujo. 2013b. *Proof of Correctness of the Bitwise Algorithm for Intra-procedural Data-flow Testing Coverage*. Technical Report PPgSI-001/2013. School of Arts, Sciences and Humanities, University of Sao Paulo. http://ppgsi.each.usp.br/arquivos/RelTec/PPgSI-001_2013.pdf

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2003. Modular Verification of Software Components in C. In *ICSE*. 385–395.

BRamkrishna Chatterjee and Barbara G. Ryder. 1999. *Data-flow-based testing of object-oriented libraries*. Technical Report DCS-TR-382. Rutgers University.

T. Y. Chen. 2008. Adaptive Random Testing. In *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK*. 443.

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 71–80.

Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. 52–71.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. 1989. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Trans. Software Eng.* 15, 11 (1989), 1318–1332.

Roberto Paulo Andrioli de Araujo and Marcos Lordello Chaim. 2014. 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*. 81–90.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 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*. 337–340.

Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. 2008. 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*. 246–260.

Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. 2009. 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*. 421–422.

Giovanni Denaro, Alessandro Margara, Mauro Pezze, and Mattia Vivanti. 2015. Dynamic Data Flow Testing of Object Oriented Systems. In *37th International Conference on Software Engineering, ICSE '15*.

Giovanni Denaro, Mauro Pezzè, and Mattia Vivanti. 2013. Quantifying the complexity of dataflow testing. In *AST*. 132–138.

Giovanni Denaro, Mauro Pezzè, and Mattia Vivanti. 2014. On the Right Objectives of Data Flow Testing. In *ICST*. 71–80.

Mingjie Deng, Rong Chen, and Zhenjun Du. 2009. Automatic test data generation model by combining dataflow analysis with genetic algorithm. In *Pervasive Computing (JCPC), 2009 Joint Conferences on*. 429 – 434.

Bruno Dutertre. 2014. Yices 2.2. In *Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 737–744.

Jon Edvardsson. 1999. A Survey on Automatic Test Data Generation. (1999).

Lynn M. Foreman and Stuart H. Zweben. 1993. A study of the effectiveness of control and data flow testing strategies. *Journal of Systems and Software* 21, 3 (1993), 215–228.

P. G. Frankl. 1987. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. Ph.D. Dissertation. University of New York, NY.

Phyllis G. Frankl and Oleg Iakounenko. 1998. 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*. 153–162.

P. G. Frankl and S. N. Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Trans. Softw. Eng.* 19, 8 (Aug. 1993), 774–787.

Phyllis G Frankl, Stewart N Weiss, and Elaine J Weyuker. 1985. *Asset: A system to select and evaluate tests*. Courant Institute of Mathematical Sciences, New York University.

Phyllis G. Frankl and Elaine J. Weyuker. 1985. A Data Flow Testing Tool. In *Proceedings of the Second Conference on Software Development Tools, Techniques, and Alternatives*. IEEE Computer Society Press, Los Alamitos, CA, USA, 46–53.

P. G. Frankl and E. J. Weyuker. 1988. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. Softw. Eng.* 14, 10 (Oct. 1988), 1483–1498.

Gordon Fraser and Andrea Arcuri. 2012. Sound empirical evidence in software testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 178–188.

Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291.

Gordon Fraser, Franz Wotawa, and Paul Ammann. 2009. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.* 19, 3 (2009), 215–261.

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 519–531.

Kamran Ghani and John A. Clark. 2009. 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*. 152–157.

Ahmed S. Ghiduk. 2010. A New Software Data-Flow Testing Approach via Ant Colony Algorithms. *Universal Journal of Computer Science and Engineering Technology* 1, 1 (October 2010), 64–72.

Ahmed S. Ghiduk, Mary Jean Harrold, and Moheb R. Girgis. 2007. Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage. In *APSEC*. 41–48.

Moheb R. Girgis. 1993. Using Symbolic Execution and Data Flow Criteria to Aid Test Data Selection. *Softw. Test., Verif. Reliab.* 3, 2 (1993), 101–112.

Moheb R. Girgis. 2005. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *J. UCS* 11, 6 (2005), 898–915.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 213–223.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (Jan. 2012), 20:20–20:27.

Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. SYNERGY: a new algorithm for property checking.. In *SIGSOFT FSE*. 117–127.

Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. 2010. 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*. 182–191.

Mary Jean Harrold. 1994. Perforining Data Flow Testing in Parallel. In *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*. 200–207.

Mary Jean Harrold and Brian A Malloy. 1992. Data flow testing of parallelized code. In *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 272–281.

Mary Jean Harrold and Gregg Rothermel. 1994. Performing Data Flow Testing on Classes. In *SIGSOFT FSE*. 154–163.

Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (March 1994), 175–204.

Mohammad Mahdi Hassan and James H. Andrews. 2013. 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*. 172–181.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *POPL*. 58–70.

P. M. Herman. 1976. A Data Flow Analysis Approach to Program Testing. *Australian Computer Journal* 8, 3 (1976), 92–96.

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

Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. 2003. Data Flow Testing as Model Checking. In *ICSE*. 232–243.

Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo-Hwan Bae, and Hasan Ural. 2000. A test sequence selection method for statecharts. *Softw. Test., Verif. Reliab.* 10, 4 (2000), 203–227.

Hyoung Seok Hong and Hasan Ural. 2005. Dependence Testing: Extending Data Flow Testing with Control Dependence. In *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings*. 23–39.

J. R. Horgan and London. 1992. ATAC: A data flow coverage testing tool for C. In *Proceedings of Symposium on Assessment of Quality Software Development Tools*. 2–10.

Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *ICSE*. 191–200.

RTCA Inc. December 1992. DO-178b: Software Considerations in Airborne Systems and Equipment Certification. *Requirements and Technical Concepts for Aviation* (December 1992).

Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 435–445.

Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. 1993. Interprocedural Dynamic Slicing Applied to Interprocedural Data Flow Testing. In *ICSM*. 386–395.

Ken Kennedy. 1979. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division.

Abdelaziz Khamis, Reem Bahgat, and Rana Abdelaziz. 2011. Automatic Test Data Generation using Data Flow Information. *Dogus University Journal* 2 (2011), 140–153.

Arunkumar Khannur. 2011. *Software Testing - Techniques and Applications*. Pearson Publications.

James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. 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*. 1098–1105.

Kiran Lakhotia, Phil McMinn, and Mark Harman. 2009. 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*. IEEE Computer Society, Washington, DC, USA, 95–104.

Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. 2010. 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*. 142–157.

Janusz W. Laski and Bogdan Korel. 1983. A Data Flow Oriented Program Testing Strategy. *IEEE Trans. Software Eng.* 9, 3 (1983), 347–354.

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

Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141.

You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 19–32.

Konstantinos Liaskos and Marc Roper. 2007. Automatic Test-Data Generation: An Immunological Approach. In *Proceedings of Testing: Accademic and Industrial Conference - Practice and Research Techniques (TAIC PART '07)*. Windsor, UK, 77–81.

Konstantinos Liaskos and Marc Roper. 2008. 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*. 211–220.

Konstantinos Liaskos, Marc Roper, and Murray Wood. 2007. 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*. 1140.

Orna Lichtenstein and Amir Pnueli. 1985. 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*. 97–107.

Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. 2009. 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*. 221–232.

Chien-Hung Liu, David Chenho Kung, and Pei Hsia. 2000. Object-based data flow testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*. IEEE, 7–16.

Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *SAS*. 95–111.

Nicos Malevris and Derek F. Yates. 2006. The collateral coverage of data flow criteria when branch testing. *Information & Software Technology* 48, 8 (2006), 676–686.

Martina Marré and Antonia Bertolino. 1996. Unconstrained Duas and Their Use in Achieving All-uses Coverage. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*. ACM, New York, NY, USA, 147–157.

Martina Marré and Antonia Bertolino. 2003. Using Spanning Sets for Coverage Testing. *IEEE Trans. Softw. Eng.* 29, 11 (Nov. 2003), 974–984.

Vincenzo Martena, Alessandro Orso, and Mauro Pezzè. 2002. 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*. 135–144.

Kenneth Lauchlin McMillan. 1992. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX92-24209.

Phil McMinn. 2004. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.* 14, 2 (2004), 105–156.

Lijun Mei, WK Chan, and TH Tse. 2008. Data flow testing of service-oriented workflow applications. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 371–380.

Lijun Mei, WK Chan, and TH Tse. 2009. 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, 151–160.

Ettore Merlo and Giuliano Antoniol. 1999. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage.. In *CASCON*. 7.

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

Matteo Miraz. 2010. *Evolutionary Testing of Stateful Systems: a Holistic Approach*. Ph.D. Dissertation. Politecnico di Milano.

Jonathan Misurda, Jim Clause, Juliya Reed, Bruce R Childers, and Mary Lou Soffa. 2005a. Jazz: A tool for demand-driven structural testing. In *Compiler Construction*. Springer, 242–245.

Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. 2005b. Demand-driven structural testing with dynamic instrumentation. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 156–165.

Akbar Siami Namin and James H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 57–68.

Narmada Nayak and Durga Prasad Mohapatra. 2010. Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization. In *IC3 (2)*. 1–12.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 213–228.

A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. 1999. The Dynamic Domain Reduction Procedure for Test Data Generation. *Softw., Pract. Exper.* 29, 2 (1999), 167–193.

Norbert Oster. 2005. Automated Generation and Evaluation of Dataflow-Based Test Data for Object-Oriented Software.. In *QoSA/SOQUA*. 212–226.

Thomas J. Ostrand and Elaine J. Weyuker. 1991a. Data Flow-Based Test Adequacy Analysis for Languages with Pointers. In *Symposium on Testing, Analysis, and Verification*. 74–86.

Thomas J Ostrand and Elaine J Weyuker. 1991b. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the symposium on Testing, analysis, and verification*. ACM, 74–86.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. 75–84.

H. D. Pande, W. A. Landi, and B. G. Ryder. 1994. Interprocedural Def-Use Associations for C Systems with Single Level Pointers. *IEEE Trans. Softw. Eng.* 20, 5 (May 1994), 385–403.

Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, Washington, DC, USA, 46–57.

Yu Qi, David Kung, and Eric Wong. 2006. An agent-based data-flow testing approach for Web applications. *Information and software technology* 48, 12 (2006), 1159–1171.

Sandra Rapps and Elaine J. Weyuker. 1982. Data Flow Analysis Techniques for Test Data Selection. In *Proceedings of the 6th International Conference on Software Engineering (ICSE '82)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 272–278.

Sandra Rapps and Elaine J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Software Eng.* 11, 4 (1985), 367–375.

Torsten Robschink and Gregor Snelting. 2002. Efficient Path Conditions in Dependence Graphs. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 478–488.

Raul Santelices and Mary Jean Harrold. 2007. Efficiently monitoring data-flow test coverage. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07)*. ACM, New York, NY, USA, 343–352.

Raúl A. Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 56–66.

Raúl A. Santelices, Saurabh Sinha, and Mary Jean Harrold. 2006. 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*. 2–5.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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*. ACM, New York, NY, USA, 263–272.

Sanjay Singla, Dharminder Kumar, H M Rai, and Priti Singla. 2011a. A Hybrid PSO Approach to Automate Test Data Generation for Data Flow Coverage with Dominance Concepts. *Journal of Advanced Science and Technology* 37 (2011), 15–26.

Sanjay Singla, Priti Singla, and H M Rai. 2011b. An Automatic Test Data Generation for Data Flow Coverage Using Soft Computing Approach. *IJRRCS* 2, 2 (2011), 265–270.

Amie L. Souter and Lori L. Pollock. 2003. The Construction of Contextual Def-Use Associations for Object-Oriented Systems. *IEEE Trans. Software Eng.* 29, 11 (2003), 1005–1018.

Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining Symbolic Execution and Model Checking for Data Flow Testing. In *37th International Conference on Software Engineering, ICSE '15*.

Ting Su, Geguang Pu, Bin Fang, Jifeng He, Jun Yan, Siyuan Jiang, and Jianjun Zhao. 2014. 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*. 98–107.

Tao Sun, Zheng Wang, Geguang Pu, Xiao Yu, Zongyan Qiu, and Bin Gu. 2009. Towards Scalable Compositional Test Generation. In *QSIC*. 353–358.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *TAP*. 134–153.

Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*. 17–31.

Paolo Tonella. 2004. 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*. 119–128.

Hasan Ural, Kassem Saleh, and Alan W. Williams. 2000. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications* 23, 7 (2000), 609–627.

Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, W Eric Wong, and Márcio Eduardo Delamaro. 2005. Coverage testing of Java programs and components. *Science of Computer Programming* 56, 1 (2005), 211–230.

Mattia Vivanti. 2014. Dynamic data-flow testing. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*. 682–685.

Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. 2013. Search-based data-flow test generation. In *ISSRE*. 370–379.

Chen-Wei Wang and Alessandra Cavarra. 2009. Checking model consistency using data-flow testing. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*. IEEE, 414–421.

Zheng Wang, Xiao Yu, Tao Sun, Geguang Pu, Zuohua Ding, and Jueliang Hu. 2009. Test Data Generation for Derived Types in C Program. In *TASE*. 155–162.

Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information & Software Technology* 43, 14 (2001), 841–854.

Elaine J. Weyuker. 1990. The Cost of Data Flow Testing: An Empirical Study. *IEEE Trans. Software Eng.* 16, 2 (1990), 121–128.

Elaine J. Weyuker. 1993. More Experience with Data Flow Testing. *IEEE Trans. Software Eng.* 19, 9 (1993), 912–919.

Wolfgang Wögerer. 2005. A survey of static program analysis techniques. *Vienna University of Technology* (2005).

Qian Yang, J. Jenny Li, and David M. Weiss. 2009. A Survey of Coverage-Based Testing Tools. *Comput. J.* 52, 5 (2009), 589–597.

Xiao Yu, Shuai Sun, Geguang Pu, Siyuan Jiang, and Zheng Wang. 2011. A Parallel Approach to Concolic Testing with Low-cost Synchronization. *Electr. Notes Theor. Comput. Sci.* 274 (2011), 83–96.

Cristian Zamfir and George Candea. 2010. Execution synthesis: A technique for automated software debugging. In *EuroSys*. 321–334.

Jianjun Zhao. 2003. Data-flow-based unit testing of aspect-oriented programs. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*. IEEE, 188–197.