A Survey on Data-Flow Testing

TING SU, KE WU, WEIKAI MIAO, GEGUANG PU, and JIFENG HE, School of Computer Science and Software Engineering, East China Normal University
YUTING CHEN, Department of Computer Science and Engineering, Shanghai Jiao Tong University
ZHENDONG SU, Department of Computer Science, University of California, Davis

Data-flow testing (DFT) is a family of testing strategies designed to verify the interactions between each program variable's definition and its uses. Such a test objective of interest is referred to as a *def-use pair*. DFT selects test data with respect to various test adequacy criteria (i.e., *data-flow coverage criteria*) to exercise each pair. The original conception of DFT was introduced by Herman in 1976. Since then, a number of studies have been conducted, both theoretically and empirically, to analyze DFT's complexity and effectiveness. In the past four decades, DFT has been continuously concerned, and various approaches from different aspects are proposed to pursue automatic and efficient data-flow testing. This survey presents a detailed overview of data-flow testing, including challenges and approaches in enforcing and automating it: (1) it introduces the data-flow analysis techniques that are used to identify def-use pairs; (2) it classifies and discusses techniques for data-flow-based test data generation, such as search-based testing, random testing, collateral-coverage-based testing, symbolic-execution-based testing, and model-checking-based testing; (3) it discusses techniques for tracking data-flow coverage; (4) it presents several DFT applications, including software fault localization, web security testing, and specification consistency checking; and (5) it summarizes recent advances and discusses future research directions toward more practical data-flow testing.

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, data-flow analysis, test data generation, coverage tracking

ACM Reference Format:
Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A survey on data-flow testing. ACM Comput. Surv. 50, 1, Article 5 (March 2017), 35 pages.
DOI: http://dx.doi.org/10.1145/3020266

5

1. INTRODUCTION

Data-flow testing (DFT) is a family of testing strategies that selects program paths to exercise the definition-use relations with respect to data objects. It fills the gaps

---

This research was sponsored in part by the National Nature Science Foundation of China (Grant No. 61572197, 61402178, 61361136002, 61572312), and United States NSF Grants 1117603, 1319187, and 1349528. Weikai Miao was partially supported by Shanghai STC Project grant 14YF1404300.
Authors' addresses: T. Su, K. Wu, W. Miao, G. Pu (corresponding author), and J. He, Shanghai Key Laboratory of Trustworthy Computing, School of Computer Science and Software Engineering, East China Normal University, Shanghai, China; emails: tsuletgo@gmail.com, sei_wk2009@126.com, wkmiao@sei.ecnu.edu.cn, ggpu@sei.ecnu.edu.cn, jifeng@sei.ecnu.edu.cn; Y. Chen (corresponding author), Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China; email: chenyt@cs.sjtu.edu.cn; Z. Su, Department of Computer Science, University of California, Davis, CA, USA; email: su@cs.ucdavis.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 0360-0300/2017/03-ART5 $15.00
DOI: http://dx.doi.org/10.1145/3020266

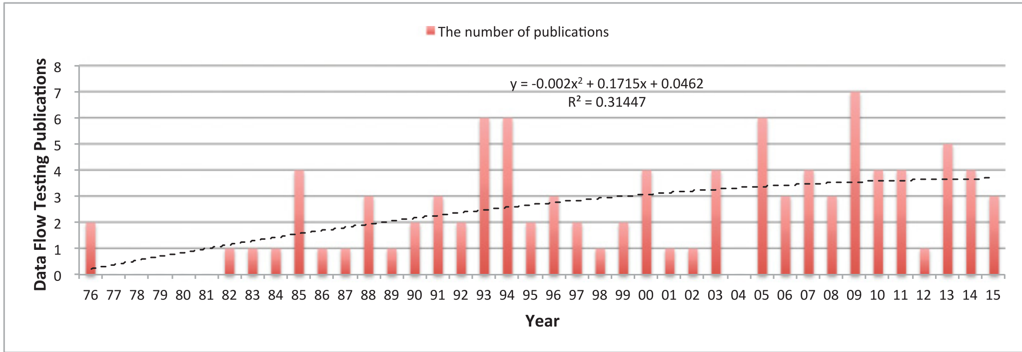


Fig. 1. Data-flow testing publications from 1976 to 2015 (the dotted curve indicates the trend of continuous research interests).

between all path testing and branch/statement testing with the aim to pinpoint the potential data-flow anomalies. When used as a test selection criterion, DFT can provide a more comprehensive testing method to ensure the test adequacy of a piece of software and detect bugs that would not necessarily be found by less demanding criteria.

The conception of data-flow testing grew out of data-flow analysis used in compiler optimizations [Allen and Cocke 1976] and was originally introduced by Herman [1976] in 1976. Since then, various slightly different notions of data-flow-based coverage criteria [Rapps and Weyuker 1982; Laski and Korel 1983; Rapps and Weyuker 1985; Frankl and Weyuker 1988; Clarke et al. 1989; 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 adaptations in procedural and object-oriented programming languages. Later, the effectiveness of DFT was justified by several empirical studies [Frankl and Weiss 1993; Foreman and Zweben 1993; Weyuker 1993; Hutchins et al. 1994; Frankl and Iakounenko 1998], which have shown that 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 [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 continuously researched (illustrated in Figure 1). Much research effort has been made to achieve practical and efficient DFT. However, little work in the literature gives a deep investigation or analysis on its state of the art, which leaves academic researchers and software practitioners unaware of the maturity of this field. For example, introductory chapters about data-flow testing can be found in many software testing tutorials, for example, the books by Beizer [1990], Pezzè and Young [2007], and Ammann and Offutt [2008]. They introduce the basic conceptions and identify the challenges but do not discuss its automation. Moreover, DFT provides a more intensive way of selecting test cases, which is among the most labor-intensive of tasks in its enforcement (this is true for other structural testing criteria as well) and has a strong impact on its testing effectiveness and efficiency. However, the automatic test data generation techniques for DFT have not been particularly investigated. Despite Edvardsson [1999] and Anand et al. [2013] surveying various techniques for automatic test data generation, they discuss them mainly in the context of control-flow-based coverage criteria.

In spite of the ability of DFT to detect data interaction faults, a big gap between real-world programs and the practicality of proposed DFT techniques still exists. Thus,

we believe that for both academic researchers and industrial practitioners, it is highly desirable to review the current research state, recognize the difficulties in its application, 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: (1) data-flow analysis, (2) data-flow test data generation, and (3) data-flow coverage computation. In this article, we mainly concentrate on the techniques used in the latter two steps and provide a relatively brief summary of data-flow analysis techniques applied in DFT because data-flow analysis itself has already been investigated in Kennedy [1979].

To this end, we present this first survey on data-flow testing: we set up a DFT publication repository, which contains a total of 97 papers from 1976 to 2015. Several popular online digital libraries (e.g., *ACM Digital Library*, *IEEE Xplore*, *Springer Online*, *Elsevier Online*, *Wiley Online,* and *ScienceDirect*) are searched to collect valid papers, which contain the following keywords (continuously refined during the search) in either their titles or abstracts: "def-use pairs," "data-flow relations," "data flow testing + analysis," "data flow testing + test generation," "data flow coverage," and "def-use testing." Then, following these 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, that is, 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
—***Application.*** Studies on applying data-flow testing to other research fields, for example, 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 category; 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 can still fairly represent the current research state in DFT.

The remainder 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 its application. Section 3 summarizes data-flow analysis techniques used for finding def-use pairs. Section 4 investigates the general approaches to DFT's test data generation and discusses their principles, strengths, and weaknesses. Section 5 surveys DFT's coverage tracking techniques and tools. Recent research advances are discussed in Section 6, and DFT's applications in Section 7. Section 8 presents our new insights and future research directions for data-flow testing. Section 9 makes a conclusion.

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

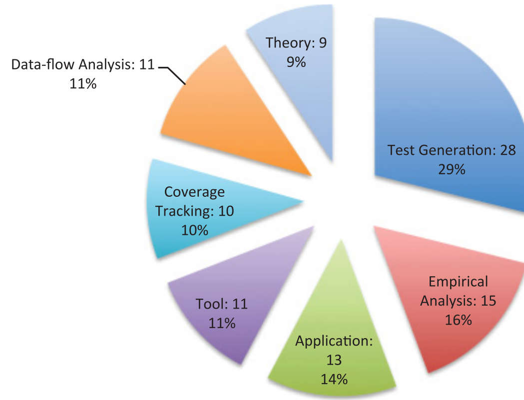


Fig. 2. Percentage of each research topic and its number of publications in the literature on data-flow testing.

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 it suffers from.

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 a program input.

Definition 2.1 (*Def-Use Pair*). Following the classic definition from 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* for short) of the 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, two types of variable uses are distinguished in data-flow testing [Rapps and Weyuker 1982, 1985]. If x is used in a computational or output statement, the use is referred to as a *computation use* (or *c-use*), and the pair is denoted as $dcu(l_d, l_u, x)$, where x is defined at l_d and used at l_u. If x is used in a conditional statement, its use is called as a *predicate use* (or *p-use*). At this time, two def-use pairs appear, denoted as $dpu(l_d, (l_u, l_t), x)$ and $dpu(l_d, (l_u, l_f), x)$, where x is defined at l_d, used at l_u, but has two opposite flow directions (l_u, l_t) and (l_u, l_f): the former denotes the *true* edge of the conditional statement in which x is used; the latter the *false* edge.[3]

In the literature, three types of data-flow testing [Badlaney et al. 2006] exist: *static*, *dynamic*, and *hybrid* data-flow testing. Static data-flow testing statically analyzes the program code and detects potential bugs with respect to the patterns of data anomalies [Huang 1979; Copeland 2003] without executing the code. However, it may fail in the situations where the state of a data object cannot be determined by only analyzing the code.[4] In contrast, dynamic data-flow testing detects data anomalies during

[2]We use line numbers to denote control points in a program.
[3]Without ambiguity, we use $du(l_d, l_u, x)$ to denote a def-use pair in this article.
[4]For example, static data-flow testing cannot determine which element is referenced when the value of an array index variable is only assigned at runtime.

```
 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 }
```

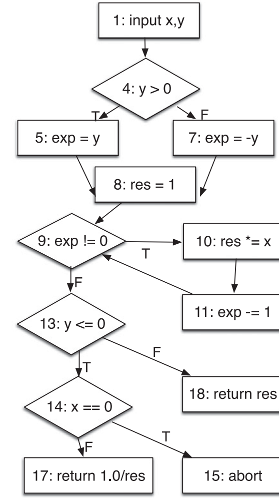


Fig. 3. An example: *power*.

program execution. Hybrid data-flow testing is the combination of them. Throughout the article, we focus on dynamic data-flow testing, which is the target problem in most research work. In the following, we give the definition of (dynamic) data-flow testing.

Definition 2.2 (*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 p 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.

Rapps and Weyuker [1982, 1985] first[5] define the requirement to cover all def-use pairs at least once as *all def-use coverage criterion* (or *all-uses coverage*), which means at least one def-clear path of each pair should be covered. In particular, for a c-use pair, p should cover l_d and l_u; for a p-use pair, p should cover l_d and the true or false edge, that is, (l_u, l_t) or (l_u, l_f), respectively.

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. Following the definitions from Rapps and Weyuker [1982], Figure 4 shows the definitions and uses of the variables in *power* and the corresponding def-use pairs. We can see that this example program has total 19 statements, eight branches, and 20 def-use pairs.

For example, the followings are two def-use pairs with respect to the variable *res*:

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

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

Here, du_1 is a def-use pair because the definition with respect to the variable *res* (at Line 8) can reach the corresponding use (at Line 17) through the control-flow

[5]We found almost all of the literature that followed uses or extends the definitions by Rapps and Weyuker [1982, 1985]. In addition, Frankl and Weyuker [1988] define *feasible* data-flow testing criteria, which emphasize the def-use pairs that are executable; Laski and Korel [1983] present testing strategies for use-definition chains. In this article, we focus on the most widely used criteria proposed by Rapps and Weyuker.

Line	Def	C-use	P-use	du
l_1	x, y			
l_4			y	$(l_1, (l_4, l_5), y)$, $(l_1, (l_4, l_7), y)$
l_5	exp	y		(l_1, l_5, y)
l_7	exp	y		(l_1, l_7, y)
l_8	res			
l_9			exp	$(l_5, (l_9, l_{10}), exp)$, $(l_5, (l_9, l_{13}), exp)$, $(l_7, (l_9, l_{10}), exp)$, $(l_7, (l_9, l_{13}), exp)$
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	$(l_1, (l_{13}, l_{14}), y)$, $(l_1, (l_{13}, l_{18}), y)$
l_{14}			x	$(l_1, (l_{14}, l_{15}), x)$, $(l_1, (l_{14}, l_{17}), x)$
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.

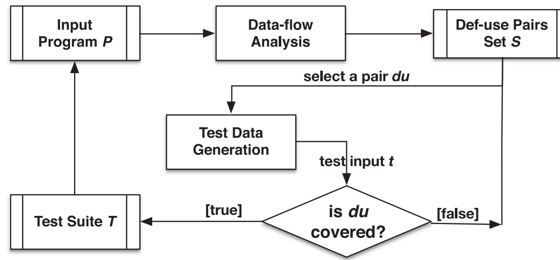


Fig. 5. The basic process of data-flow testing.

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 2.2). 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 inputs exist for this pair, which can both avoid redefinitions in the loop and reach the *use*.

2.3. Basic Testing Process

Data-flow testing consists of three basic phases: data-flow analysis, test data generation, and coverage tracking (illustrated in Figure 5), which totally occupy almost 50% of 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 and not redefined, t is incorporated into the 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.

2.4. Difficulties

Despite DFT being able to detect 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.

Complexity of Data-Flow Test Design. The number of test objectives with respect to data-flow criteria in a program is much larger than those of simple control-flow criteria.[6] In addition, more efforts are required to derive a data-flow test case: a tester has to cover a variable definition and its corresponding uses without variable redefinitions rather than just covering a statement or branch.

Infeasible Test Objectives. Due to the conservativeness of static data-flow analysis techniques when applied in identifying test objectives, def-use pairs may include *infeasible* ones. A pair is *feasible* if there exists an execution path that can pass through it. Otherwise, it is *infeasible* (e.g., the pair (l_8, l_{18}, res) in Section 2.2 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.

Here the problem of identifying infeasible test objectives is actually undecidable, and no techniques can reliably give definite conclusions on the feasibility. It is not unique in DFT but also exists in structural testing. In spite of the aforementioned difficulties, with the help of the existing techniques and recent advances, DFT can be automated and these challenges can be mitigated, as this survey will demonstrate.

3. CLASSIC DATA-FLOW ANALYSIS

To identify test objectives (i.e., def-use pairs) in DFT, a reaching definition procedure [Allen and Cocke 1976] (it also inspires the definitions of data-flow coverage criteria) is usually used, which actually answers such a question: for each variable use, which definitions can potentially supply the values to it?

Harrold and Soffa [1994] use a standard iterative data-flow analysis to compute definition-use relations for high-level languages. The intraprocedural definition and use information are abstracted for each procedure via control-flow graphs and then used to compute interprocedural def-use pairs that cross the boundaries of procedures. Pande et al. [1994] extend the reaching definition analysis to handle programs with single-level pointers for *C* language. The algorithm considers program-point-specific pointer-induced aliases and has polynomial-time complexity.

To counter the complexity of the traditional exhaustive and incremental data-flow analysis, Duesterwald et al. [1996, 1997] propose a demand-driven data-flow analysis technique to aid DFT when integration testing is used to validate program interfaces. The analysis is performed as a goal-oriented search instead of using exhaustive information propagation. It efficiently computes the newly established data-flow information during each bottom-up integration step and does not need to store the reaching definition solutions between each step.

Harrold and Rothermel [1994] extend data-flow analysis for object-oriented languages, which not only considers the definition-use relations within methods (i.e.,

[6]Note that some control-flow criteria can be more demanding than data-flow criteria, namely, any that explicitly require execution of all paths of a particular type (e.g., all loop-free paths) and all paths that go around loops at most 5 times.

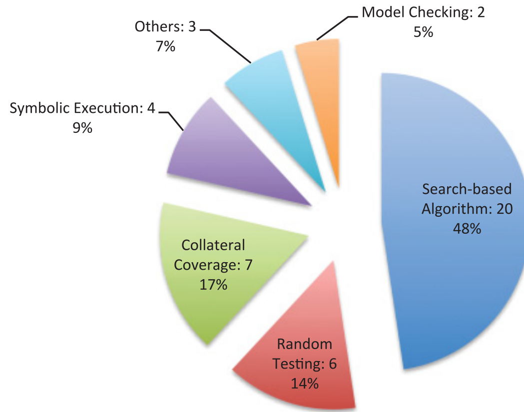


Fig. 6. Percentage of publications using each testing approach on data-flow-based test data generation.

intramethod def-use pairs) but also computes data-flow relations through instance variables (i.e., *intermethod* and *intraclass* def-use pairs). Chatterjee and Ryder [1999] propose a flow- and context-sensitive algorithm based on point-to analysis to compute def-use pairs for object-oriented libraries. The algorithm tackles the difficulties of unknown aliasing between parameters, unknown concrete types of the parameters, and dynamic dispatch and exceptions. Souter and Pollock [2003] and Denaro et al. [2008] also extend classic data-flow analysis to object-oriented programs and especially consider the construction of contextual def-use pairs that are created by class objects.

To improve the precision of static def-use pair analysis, Bodík et al. [1997] propose an approach to exclude parts of infeasible def-use pairs by utilizing the information of some infeasible paths that can be detected at compile time (note that the problem of identifying all infeasible paths is undecidable). This approach detects *static branch correlation* to identify infeasible program subpaths and then excludes def-use pairs that span these infeasible subpaths. The algorithm is implemented as demand driven at both intraprocedural and interprocedural levels and is suitable for DFT in regression testing and integration testing.

Classic data-flow analysis belongs to static analysis, which may have to be enhanced by alias analysis in practice at different precision levels (e.g., flow and/or context sensitive). In practice, a suitable approximation should be made to trade off between precision and scalability.

4. APPROACHES TO DATA-FLOW-BASED TEST DATA GENERATION

This section presents various approaches to automating data-flow-based test data generation, which have a strong impact on the effectiveness and efficiency of DFT. From the publication repository, we find that the test generation problem is the most active research topic in the study of data-flow testing and has been continuously concerned over the past 20 years. A total of 27 technical research papers are related to this topic. We classified them into five main groups according to their testing techniques: *search-based testing*, *random testing*, *collateral-coverage-based testing*, *symbolic-execution-based testing*, and *model-checking-based testing*.

We compute the percentages[7] of each testing approach used in these papers (shown in Figure 6). We find that the search-based testing approach (including genetic algorithm and optimization algorithm) is the most widely studied, which occupies almost 50% of

[7] If one paper uses more than one testing approach, we count all of them.

research efforts. The collateral-coverage-based approach and random testing are also popular. But more sophisticated testing approaches, for example, symbolic execution and model checking, are less investigated.

In the following, we will detail these approaches in independent sections starting with the most widely studied and going to the least studied, that is, search-based testing (Section 4.1), random testing (Section 4.2), collateral-coverage-based testing (Section 4.3), symbolic-execution-based testing (Section 4.4), and model-checking-based testing (Section 4.5). At last, some other approaches are discussed in Section 4.6.

4.1. Search-Based Testing Approach to Data-Flow Testing

Search-based software testing [Harman et al. 2015], as an instance of the general search-based software engineering area, has continuously received wide research interest in the past decades.

Principle of Search-Based Approach. The search-based approach includes various metaheuristic techniques [McMinn 2004] and utilizes them 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.

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 that share genetic material from parents, while mutation adds small changes to a proportion of the populations.

Several GA-based testing methods have been proposed to tackle the DFT problem [Girgis 2005; Ghiduk et al. 2007; Vivanti et al. 2013]. Girgis [2005] first uses GA for data flow testing w.r.t. all-uses coverage. In Girgis [2005], GA uses a binary string s of length m as a chromosome (i.e., a test case) to represent the values of input variables. Assume the program under test has k input variables (e.g., $v_1, \ldots, v_i, \ldots, v_k$, $1 \le i \le k$), the input range of v_i is $[a_i, b_i]$, and d_i is the desirable precision for the values of v_i. Then the mapping from the binary string s_i to the variable value v_i with the domain $[a_i, b_i]$ is established by the following formula:

$$v_i = a_i + v_i' * \frac{b_i - a_i}{2^{m_i} - 1},$$

where v_i' is the decimal value of the binary string s_i. Take the program in Figure 3 (Section 2) as an example and assume the input range of x and y is $[-2, 9]$ and $[-4, 13]$, respectively, and the chromosome is a binary string of length 9, where the first 4 bits from the left represent the value of x, and the next 5 bits denote the value of y. For example, the binary string $s = 010100110$ represents a test case with $x = 5$ and $y = 6$. Before test generation, GA uses this encoding method proposed by Michalewicz [1994] to generate the initial population of test cases.

In Girgis [2005], GA uses the ration between the number of covered def-use paths and the number of total def-use paths as the fitness function, which exclusively uses coverage information to determine the effectiveness of an individual test case. This GA-based method works as follows. First, it generates a set of test cases encoded in the binary string form. Then it uses a roulette wheel algorithm-based selection method [Michalewicz 1994] to pick promising individuals according to their fitness values. Next, GA uses search operators (i.e., crossover and mutation) to produce new

chromosomes from the selected parents. After a predefined count of iterations, GA could output a set of desired test cases that can cover the target def-use paths (because of the undecidability of this search problem, uncovered def-use paths may still exist).

Ghiduk et al. [2007] later find that there are some pitfalls inside the fitness function used by Girgis [2005], which could be too coarse to identify the closeness of the test cases in the following situations: (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 does 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 a similar procedure in Girgis [2005], they propose a new multiobjective 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, that is, the *def* and the *use*. To evaluate the closeness of a test case with respect to 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 but tries to cover the *def* or *use* is closer than a test case that misses and does not try to cover the *def* or *use* (a test case trying to cover a target statement means, during the following search, its mutants are reaching closer to the target). 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 that this GA approach costs less search time and requires fewer program iterations than random testing. However, it is still unclear how much this technique outperforms that of Girgis [2005] because no relevant results are provided.

Vivanti et al. [2013] use the genetic algorithm to handle 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: *intramethod pairs*, *intermethod pairs,* and *intraclass 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 the SF100 corpus of classes [Fraser and Arcuri 2012], they confirm that the test objectives of data-flow testing are much more than those of branch testing, but the resulting test suite is more effective in fault detection.

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. [2007] and Liaskos and Roper [2008] hybridize GA with the artificial immune systems (AIS) [Liaskos and Roper 2007] 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. develop 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 multiobjective fitness function and works at the class level as well as method level. The former generates useful states for class objects, and the latter uses them to reach the uncovered

code in the class. In Miraz [2010], Matteo applies this GA variant to cover def-use pairs. The author points out that 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] and Deng et al. [2009], who also use GA to automate data-flow testing but only evaluate small examples.

There also have been attempts at using optimization-based search techniques to tackle the DFT problem. Nayak and Mohapatra [2010] and Singla et al. [2011a, 2011b] use particle swarm optimization, while Ghiduk [2010] uses ant colony optimization. Inspired by natural behaviors, these optimization algorithms simulate these behaviors to find optimal solutions in the context of DFT. However, these approaches have been evaluated only on toy programs. Their effectiveness on large programs is still unclear.

Discussion. The search-based techniques have already been applied to enforce simple coverage criteria (e.g., statement and branch testing [Anand et al. 2013]), as well as some advanced coverage criteria [Ammann et al. 2003; Inc 1992] (e.g., logical coverage [Awedikian et al. 2009; Ghani and Clark 2009]) and data-flow coverage as discussed earlier. This approach treats test data generation as a domain search problem, and thus, it is more competent at solving nonlinear constraints and finding floating-point inputs [Lakhotia et al. 2009, 2010; Bagnara et al. 2013] than those constraint-based approaches (e.g., symbolic execution).

However, there still exist some problems that should be noted and investigated: First, the testing performance of search-based techniques heavily depends on the underlying fitness functions (it may take a very long time to find good solutions), and thus enough carefulness is required in its design and optimization. Second, compared with GA, some optimization-based algorithms (e.g., particle swarm optimization and colony optimization) are much less studied, and their scalability on real-world programs is still unclear. Third, although the multigoal fitness functions [Lakhotia et al. 2007; Fraser and Arcuri 2013] can mitigate the impact of infeasible pairs, it still cannot detect infeasible pairs.

4.2. Random-Testing-Based Approach to Data-Flow Testing

Random testing [Bird and Munoz 1983] is one of the most widely used and cost-effective testing approaches. In its classic implementation, test inputs are randomly picked from the value ranges with respect to program specifications and later executed against the program under test.

This classic random testing technique has been adopted as an easily implemented but quite efficient baseline approach for data-flow testing in several works [Girgis 2005; Ghiduk et al. 2007; Su et al. 2015]. For object-oriented systems, a test case is a sequence of class constructor invocations and method calls [Pacheco et al. 2007]. Random testing, adapted to randomly generate these sequences to exercise the classes under test, has also been used in the filed of data-flow testing [Alexander et al. 2010; Denaro et al. 2015]. In addition, other forms of random testing [Girgis et al. 2014], for example, randomly selecting test paths from program graphs to cover def-use pairs (and then using test generators to derive corresponding test cases from those test paths), are used to achieve data-flow testing.

Discussion. Random testing is cost-effective and easy to implement, but it can only distinguish limited sets 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, random testing could become a competitive test generation approach for DFT.

For example, researchers find that if previously selected tests cannot reveal program faults, the next new tests should be selected far away from the already executed ones

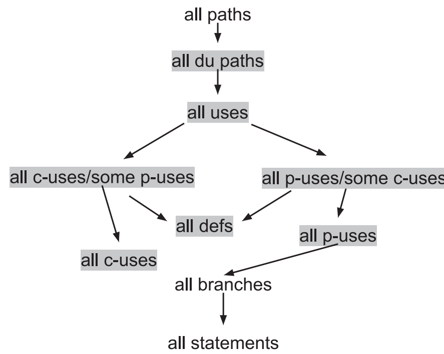


Fig. 7. Hierarchy of structural characteristics of a program.

so as to improve the chance of triggering faults. Adaptive random testing [Chen 2008; Ciupa et al. 2008; Lin et al. 2009; Arcuri and Briand 2011], such as enhancement of traditional random testing, which evenly spreads test cases across their input domains, could improve the efficiency of data-flow testing. In addition, feedback-directed random testing [Pacheco et al. 2007]), which improves random testing by incorporating the feedback information obtained from the execution of test cases when they are created, can also benefit DFT.

4.3. Collateral Coverage-Based Approach to Data-Flow Testing

In software testing, *collateral coverage* has been exploited to optimize test suite generation [Harman et al. 2010; Fraser and Arcuri 2013]. It is based on the following observation: the test case that satisfies a target test objective can "accidentally" cover other test objectives. Thus, if we exclude these covered test objectives and invest the testing budgets in the remaining uncovered objectives, the size of the resulting test suite and the cost of test execution and oracle checking can be reduced.

Similarly, when exercising a program to satisfy a given testing criterion (e.g., branch coverage), test objectives with respect to other coverage criteria (e.g., data-flow coverage) may also be accidentally covered, which is another form of collateral coverage [Malevris and Yates 2006]. Formally, a test criterion C_1 *subsumes* another criterion C_2 if, for all programs P, the test cases that satisfy all the test objectives of P with respect to C_1 also satisfy those with respect to C_2. For example, Figure 7 (given in Rapps and Weyuker [1982, 1985]) shows the subsumption relations between different testing coverage criteria.[8] The criterion at the arrow tail subsumes the criterion at the arrow head (e.g., the branch criterion subsumes the statement criterion). Since the subsumption relation is transitive, it actually defines the relations between various coverage criteria. The shadowed criteria are seven types of data-flow testing criteria (refer to Rapps and Weyuker [1982, 1985] for their detailed definitions), which emphasize different ways to exercise definition-use relations. *All-uses* coverage is *all def-use* coverage (*cf.* Definition 2.2 in Section 2.1), and it subsumes *all c-uses* and *all p-uses* coverage. Moreover, *all-uses* coverage also subsumes branch coverage.[9]

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;

[8]It should be noted that the relations in Figure 7 will change when *feasible* data-flow testing criteria, proposed by Frankl and Weyuker [1988], are taken into consideration.

[9]A *p-use* pair is composed of the definition statement and the conditional statement with the use. The *all p-uses* coverage requires that both the *true* and *false* edges of the conditional statements be exercised. It is this distinction that is responsible for the fact that *all-uses* coverage subsumes branch coverage.

Marré and Bertolino 1996, 2003; Santelices et al. 2006]. 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 with respect to seven data-flow criteria (the ones shadowed in Figure 7). In their experiment, on average, over 35% of all-du-paths coverage and over 40% of all-uses coverage can be achieved when enforcing branch testing and selecting a mean of 6.44 paths per unit. The study also reveals that the actual data-flow coverage can be modeled as a function that takes as parameters the number of selected paths with respect to branch testing and the number of feasible paths therein. In addition, they also find that (1) the data-flow coverage is independent of the language that a unit uses, (2) the level of collateral coverage can be predicated to estimate the possible testing budgets demanded by DFT, and (3) undertaking branch testing before data-flow testing can be more cost-effective because parts of def-use pairs will be covered during branch testing.

Merlo and Antoniol [1999] exploit the coverage implication between def-use pairs and nodes (i.e., statements) to achieve intraprocedural data-flow testing. Pre- and postdominator analysis is used to identify a set of nodes whose coverage could imply the coverage of a subset of def-use pairs. They evaluate the approach on a 16KLOC *Gnu find* tool and find that, on average, 75% of def-use pairs are definitely covered when covering the nodes of each routine.

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 that includes all these predicate conditions for each entity is constructed to create the subsumption relations of entities for efficient coverage tracking.

Later, Santelices and Harrold [2007] proposed 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: *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 *noninferable* (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: 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, and 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 coverage tracking can be greatly mitigated since the program is instrumented at branch coverage level instead of data-flow coverage level.

Marré and Bertolino [1996, 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 therein. This set is called a *spanning set* and the pairs therein are called *unconstrained pairs*. The cardinality of this set is actually the upper bound on the size of the test suite required to achieve all-uses coverage. As a result, it can help estimate the cost of data-flow testing. Harrold et al. [1993] propose a technique to generate a representative set of test cases from a test suite that achieves the same coverage rate as the original entire test suite. This technique minimizes the test suite size by utilizing collateral coverage and is independent of the testing methodology. It only requires an association between a test requirement and the test cases that satisfy

this requirement. The evaluation on data-flow testing demonstrates that the technique is effective in removing redundant test cases but without sacrificing the coverage rate, which is particularly useful in regression testing to reduce testing cost.

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.

4.4. Symbolic Execution-Based Approach to Data-Flow Testing

Symbolic execution, first proposed by King [1976], is a classic program analysis technique and has been widely applied in software testing [Cadar and Sen 2013].

Principle of Symbolic Execution. Symbolic execution 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 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 that denotes the next program statement to execute.

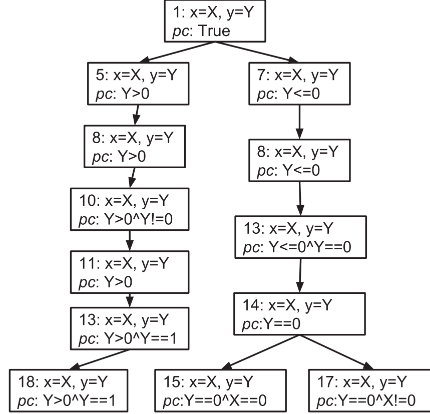
The technique works as follows: During the execution, new constraints over inputs at each branch point are used to update *pc*. If the new *pc* is unsatisfiable, the exploration of the corresponding path will stop. Otherwise, the execution will continue along this branch point such that any solution of the *pc* will execute the corresponding path. In particular, when both 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 [1993] first used a similar *static* symbolic execution system to generate data-flow-based test data. This approach first generates a set of program paths from the CFG of the program under test with respect to 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 *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 that 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 that fulfills the given data-flow testing criterion.

For the example program in Figure 3, this approach first statically explores as many paths as possible with respect to a control-flow criterion (e.g., branch coverage). Assume

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.

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 (located 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. Godefroid et al. [2005] and Sen et al. [2005] interleave the symbolic execution with concrete execution to improve the scalability of static symbolic execution. This hybrid technique (referred to as *dynamic symbolic execution* or *concolic testing* [Godefroid et al. 2005; Sen et al. 2005]) 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 complicated and is out of the reach of the constraint solver, these concrete values can be used to simplify it by value substitution.

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* that 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 that 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 can appear on the subpath 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, for example, $t = (x \mapsto 0, y \mapsto 42)$. This

test input triggers an execution path p

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

which already covers the *def* of du_1 at l_8. To cover its *use*, the classical DSE approach (e.g., with depth-first or random path search [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 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* 2.2). To illustrate, the branching nodes that will not be flipped are crossed out on p and the rest are highlighted in Equation (4). As a result, a large number of *invalid* branching nodes are pruned:

$$p = \boxed{l_4}, l_5, l_8, \underbrace{\boxed{l_9}, l_{10}, l_{11},\ \not{l_9}, l_{10}, l_{11}, \ldots}_{\text{repeated 42 times}}, \not{l_9}, \not{l_{13}}, l_{18}. \quad (4)$$

Second, a *cut-point-guided search strategy* [Su et al. 2015] is used to decide which branching node to select first. 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 Equation (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 that 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, that is, $4F$, $13F$, 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 that covers all cut points and $du_1(l_8, l_{17}, res)$ itself.

Discussion. The classic symbolic execution is a path-based testing approach that can systematically explore paths to cover target def-use pairs. In early work [Girgis 1993], Girgis used 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 is used, which may lose precision in data-flow testing. For example, it cannot precisely reason about which concrete element is referred to 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 precise and efficient. For example, variable redefinitions caused by aliases can be detected more easily and precisely with the dynamic execution information, and dynamic program execution is much faster than static program execution.

However, neither the SSE-based nor DSE-based approach is capable of identifying infeasible pairs, because symbolic-execution-based testing is an explicit path-based approach, which 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 may spend a large amount of time, in vain, covering an infeasible pair.

4.5. Model-Checking-Based Approach to Data-Flow Testing

Principle of Model Checking. Model checking [Clarke et al. 1999] is a classic formal verification approach. 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 and a property of interest; it then checks whether the property is violated or not. If the property is violated, a counterexample will be generated 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 counterexamples 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—for each $s \in S$, there is an $s' \in S$ such that $(s, s') \in T$; and (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 a WCTL (weighted CTL) formula if (1) f only has temporal operators **EX**, **EF,** and **EU**, and (2) in each subformula 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. [2003] and Hong and Ural [2005] first used 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. 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 Equation (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 with respect to 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

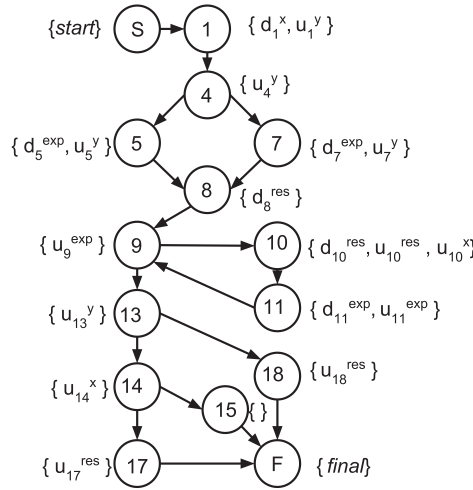


Fig. 9. The data-flow graph for the function *power* in Figure 3.

formula in Equation (5). Note that in this approach, it is not necessary to know in advance whether there exists a control-flow path between the location l and l' because the formula itself implicitly imposes this constraint:

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

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

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

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

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

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-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. So the count of formulas can be quadratic with respect to G. If it is applied into interprocedural 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. Given the program source code and a temporal safety specification, CEGAR either statically proves that the program satisfies the specification or produces a

```
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 shown by the highlighted statements.

counterexample path to demonstrate the violation. Since then, it has been 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 test cases [Beyer et al. 2004] with respect to statement or branch coverage.

Beyer et al. [2004] suggest a CEGAR-based two-phase approach, that is, *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 that exhibits p at q, a CEGAR-based model checker can generate a test case that 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.

Su et al. [2015] further adapt this CEGAR-based model-checking approach to achieve data-flow testing with respect to the all def-use criterion. 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 order 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 Equations (1) and (2), the transformed program encoded with these two test requirements is shown in Figure 10(a) and Figure 10(b), respectively. For the pair $du_1(l_8, l_{17}, res)$, Figure 10(a) shows the transformed function *power* and the encoded test requirement of du_1 in highlighted statements. The variable *cover_flag* is introduced at l_2. It is initialized to

false and set as *true* immediately after the *def* at l_7, and set to *false* immediately after the other definitions on variable *res* at l_{10}. Before the *use*, a checkpoint is set to verify whether *cover_flag* can be *true* at l_{14}. If the checkpoint is unreachable, this pair can be proved as infeasible. Otherwise, a counterexample (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 10(b), the model checker can quickly conclude that no paths can reach the checkpoint 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 path reachability checking problem. CEGAR can generate counterexamples (i.e., test cases) for feasible def-use pairs, and can also detect infeasible pairs without false positives. This technique itself can even benefit from the future advances in CEGAR-based model checkers. (2) In CEGAR, 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 implement than the other model-checking-based approaches.

However, in general, CEGAR cannot decide the feasibility of all def-use pairs, since the problem of checking path feasibility itself is undecidable. Under this circumstance, CEGAR may not terminate, and only conclude *unknown*. In addition, CEGAR is essentially a *static* approach; its 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.

4.6. Other Approaches

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 also enhance this technique with some methods for handling loops and arrays. But it only illustrates the idea with some proof-of-concept examples, and its practicality is unclear.

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, Martena et al. [2002] extended this technique from a 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. [2010], Baluda [2011], and 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 that 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. Baluda [2011] 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 widely studied techniques used to automate test data generation (as shown in Figure 6). A reasonable explanation is that these two techniques are relatively easy to implement for DFT. The symbolic-execution-based and model-checking-based techniques attracted attention quite early, but until recently they were not applied on large real-world programs, only on laboratory programs. The fact that these techniques are more difficult to implement than other approaches, and also heavily rely on the advances of other techniques (e.g., constraint solving), may explain this phenomenon. Additionally, for both procedural language (e.g., C) and object-oriented language (e.g., Java), several academic tools exist. However, according to our investigation, there are still no commercial tools supporting DFT. More efforts are needed to develop efficient and easy-to-implement techniques.

We also need to note that no existing testing techniques can reliably identify infeasible pairs due to the fundamental undecidability issue (and this is true for other structural testing as well). When applying any of them, one may always fail to cover some pairs and cannot know whether it is because no sufficient testing has been done or because they can never be covered.

5. 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.

5.1. 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.

Frankl [1987] proposes a *deterministic finite automata*-based approach to track the coverage status of def-use pairs. In her approach, a pair *du* is related with a regular expression that describes the control-flow paths covering it. Each automaton associated with *du* is checked against all execution paths. Once one path is accepted by some automaton, the pair *du* is set as covered. But this approach has to do special handling when the procedure under test recursively calls itself. Ostrand and Weyuker [1991a] use a *memory tracking* technique to precisely determine which pairs are covered, while Kamkar et al. [1993] use *dynamic slicing* to improve coverage precision. Some work [Harrold and Malloy 1992; Su et al. 2015] exploits dynamic data-flow analysis to improve the precision of tracking data-flow coverage.

Horgan and London [1992] exploit code instrumentation to track DFT coverage, which is later called 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 where 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. [2005b] propose a *demand-driven* strategy to track def-use pair coverage, which aims to improve the performance of the static instrumentation approach (e.g., 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 [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, that is, *inferable*, *conditionally inferable,* and *noninferable* 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 4.3.

In order to make coverage tracking more scalable, Harrold [1994] develops a technique on multiprocessor 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.

Discussion. Data-flow coverage imposes high overhead on tracking its coverage. The main reasons are that (1) data-flow-based test objectives are usually much more than statements or branches, and (2) data-flow coverage puts constraints (i.e., satisfying def-clear paths) on program paths instead of simply program entities, which makes tracking more expensive. The existing approaches mainly resort to efficient data structures or exploit coverage inference to mitigate the overhead.

5.2. 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, 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, 2012], and 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 interprocedure pairs or both, the analysis infrastructure it is based on, the coverage tracking technique it uses, and its availability. Six out of a total of 12 tools are publicly available, but none of them are commercial tools, which has also been reported by Hassan and Andrews [2013] and de Araujo and Chaim [2014] recently.

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

6. RECENT ADVANCES

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

New Coverage Criteria. Hassan and Andrews [2013] introduce a new family of coverage criteria, called *Multipoint 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 that 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.

Alexander et al. [2010] extend the classic data-flow criteria to test and analyze the polymorphic relationships in object-oriented systems. The new coverage criteria consider definitions and uses between state variables of classes, particularly in the presence of inheritance, dynamic binding, and polymorphic overriding of state variables and methods. The aim is to increase the fault detection ability of DFT in object-oriented programs.

Dynamic Data-Flow Analysis. Denaro et al. [2014] and 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 more precise than considering statically computed alias relations, which is the typical overapproximation when integrating alias information in static data-flow analysis.

The evaluation on five Java projects reveals that a large set of data-flow relations is missed by the traditional static data-flow analysis, which undermines the effectiveness of the previous DFT approaches. This dynamic technique sheds light on a new direction of data-flow testing that can better encompass data-flow-based test objectives.

Denaro et al. [2015] 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 is attributed 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 the 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] invented 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, that is, 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.

7. 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 and time-consuming activity in program debugging to locate program errors and bugs. Agrawal et al. [1995] propose a novel method that 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 that fails on execution instead of succeeding on execution. As a result, testers can focus the statements on the failed slice. A data-flow testing tool called 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. They found data-flow tests could effectively detect those seeded errors and the dice could notably improve the fault localization performance.

Santelices et al. [2009] propose a lightweight fault localization technique that 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 have enriched people's daily lives. But testing web applications becomes a tough job when the architecture and implementation become more and more complicated. Several efforts have been devoted to apply 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. [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 so forth. Each of these models 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 intraobject, interobject, and interclient aspects. In this way, DFT is extended to test web applications.

Qi et al. [2006] develop a multiple agent-based DFT method to test web applications. They split the testing task into three levels: 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. [2008] exploit DFT to test service-oriented workflow applications such as WS-BPEL applications. They find that 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.

Alshahwan and Harman [2012] propose a state-based DFT technique for web applications, which generates new sequences of HTTP requests to enhance the existing test suites in terms of coverage and fault detection. The new tests are designed to execute the definitions of state variables (e.g., session variables) and to ensure that these values reach the corresponding uses unchanged. They find that the resultant test suite can indeed improve the quality of test suites.

7.3. Specification Consistency Checking

Various specification models are widely used in software development to build reliable systems, which help automatically generate a conforming implementation. As a result, checking model consistency is an important vehicle to ensure implementation correctness. 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 intermethod 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 is also some work that generates data-flow-based test suites from specification models like SDL (Specification and Description Language) [Ural et al. 2000] and

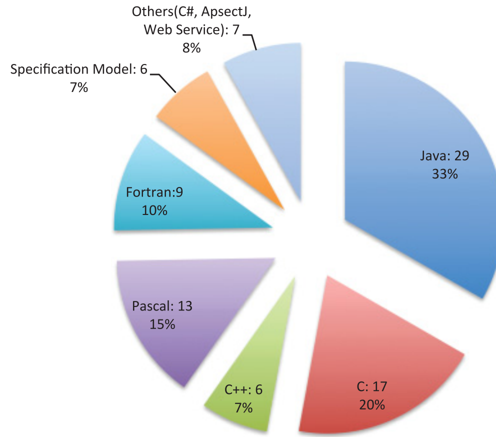


Fig. 11. Percentage of each language to which data-flow testing is applied.

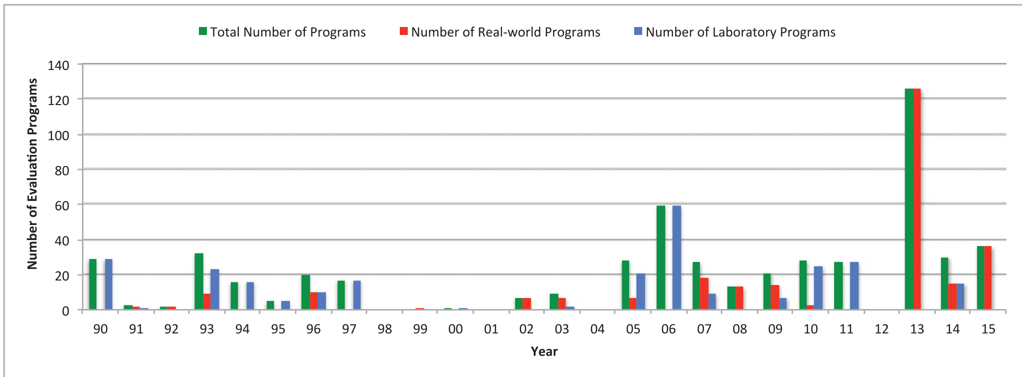


Fig. 12. Number of evaluation programs that data-flow testing is applied to since 1990, including the total number, the number of real-world, and the number of laboratory programs on each year.

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 [2003] use DFT to test aspect-oriented programs. 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].

In addition, we investigate the percentage of each language that DFT has been applied to (shown in Figure 11). We can observe the following: First, DFT is originally applied to procedural languages (e.g., Fortran, Pascal, C), but in recent years, object-oriented programs have gained more emphasis because DFT can help find more subtle faults when checking object states. Second, the object-oriented languages (e.g., C++ and Java) are the most popular language in enforcing data-flow testing. Third, specification languages and web services also attract research interests.

Figure 12 shows the number of evaluation programs in each year reported by the publications since 1990 (we omit the papers before 1990 because they mainly focus on theory and formal analysis). We can observe that (1) DFT has become more practical in

the past 10 years since the number of evaluated programs is increasing, and (2) DFT is gradually being applied to real-world programs other than laboratory programs. Moreover, we find that the techniques of data-flow analysis and coverage tracking (at the largest scale of 200KLOC) are more scalable than those of test generation (at the largest scale of 15KLOC). It indicates that more research efforts are required to narrow this gap.

8. NEW INSIGHTS AND FUTURE WORK

This section presents the new insights that we have gained during this survey. Following the three basic testing process of DFT, that is, data-flow analysis, test data generation, and coverage tracking, we recommend the following future research directions.

Data-Flow Analysis. Data-flow analysis is responsible for identifying def-use pairs.

(1) To better encompass test objectives, one can combine known static data-flow analysis techniques with *dynamic* data-flow analysis techniques [Denaro et al. 2014, 2015] and strike a balance between scalability and precision.
(2) The data-flow analysis procedure can leverage the idea of collateral coverage to infer the coverage relation between def-use pairs themselves or between def-use pairs and other program structs (e.g., statements and branches). The intention is to identify a minimal set of pairs whose coverage implies the coverage of others so that data-flow testing can focus on these *critical* pairs, and the testing cost can be reduced.
(3) The type of data-flow analysis techniques should be determined according to the testing scenarios. For example, for unit testing, traditional exhaustive data-flow analysis can be used. But for regression testing or integration testing, demand-driven data-flow analysis techniques are more suitable, which can avoid unnecessary overhead.
(4) New data-flow analysis techniques can be developed and adapted for different programming languages. For example, procedural language and object-oriented language are much different in the construction of def-use pairs.

Test Data Generation. Test data generation aims to efficiently generate suitable test cases for def-use pairs.

(1) Various *dynamic* testing approaches can be combined, including random testing, genetic/optimization-based testing, and symbolic-execution-based testing, to satisfy def-use pairs. They have different strengths in test generation, despite the fact that they are incapable of dealing with infeasible pairs.
(2) Model-checking-based approaches can be used to complement the dynamic testing approaches. It can generate tests for feasible def-use pairs as well as handle parts of infeasible pairs. The test requirements imposed by the pairs can be transformed into acceptable forms of model checkers, and then model checkers can be used to check path feasibility and output counterexamples as test cases.
(3) The symbolic-execution-based approach is effective in path-based test generation but faces the path explosion problem, while the model-checking approach (CEGAR) is effective in checking path feasibility. As a result, the symbolic execution can be informed by the runtime information of CEGAR to avoid unnecessary path explorations and improve its performance in DFT.
(4) The combination of search-based and symbolic-execution-based testing techniques has already been applied to achieve more efficient branch testing [Inkumsah and Xie 2008; Baars et al. 2011]. Their combination may also be able to improve DFT.

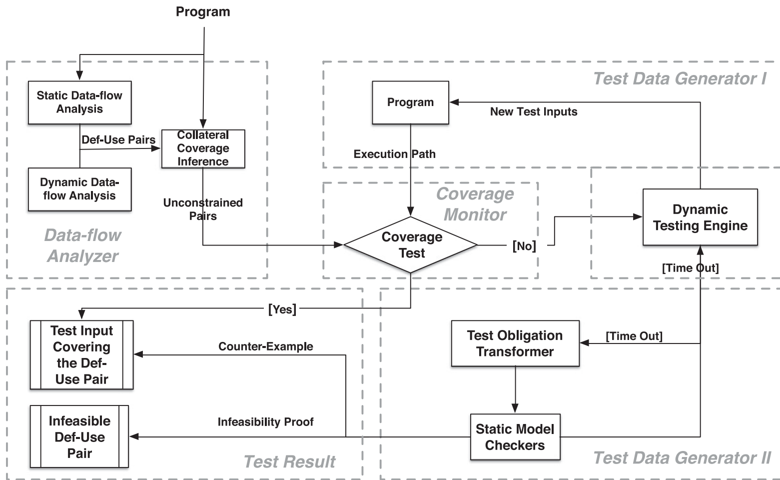


Fig. 13. The hybrid data-flow testing framework.

Coverage Tracking. Efficient coverage tracking algorithms can be proposed to improve the usability of data-flow testing on large real-world systems. Such instrumentation approaches as the BA algorithm [de Araujo and Chaim 2014] can be employed.

Based on the aforementioned new insights and research directions, a novel hybrid data-flow testing framework (shown in Figure 13) could be proposed to achieve more practical data-flow testing. It is composed of three basic components: a data-flow analyzer, a test data generator, and a coverage monitor. Given a program as input, this hybrid framework (1) *outputs test data for feasible test objectives* and (2) *eliminates infeasible test objectives*. It interleaves between dynamic testing approaches and static model checkers to maximize the data-flow coverage. It hopefully can achieve better performance by combining the strengths from its component approaches and benefit from the future advances in data-flow analysis, test data generation, and coverage tracking. Moreover, this framework 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].

Additionally, future research efforts can be endeavored to develop new cost-effective coverage criteria to complement data-flow coverage criteria. The new criterion should be easy to enforce, and it has comparable fault detection ability against DFT (e.g., Hassan and Andrews [2013] and 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.

9. CONCLUSION

In the last 40 years, data-flow testing has been increasingly and extensively studied. Various approaches and techniques have been developed to pursue efficient and automated data-flow testing, given its ability to check data interactions. To our knowledge, this is the first systematic survey for data-flow testing. We have constructed a publication repository with 97 research papers, demonstrated the current state of research, and provided comprehensive analysis in this field. We have classified data-flow-based test generation approaches into five categories. For each category, we have explained its technical principle and have discussed its strengths and weaknesses. The techniques

of coverage tracking and data-flow analysis are also summarized. Based on this investigation, we have proposed the new insights and future research directions, which aim to make DFT more efficient and practical.

ACKNOWLEDGMENTS

We would like to thank several researchers and practitioners in the area of 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 for their time and expertise. We also would like to thank the anonymous reviewers for their insightful and valuable feedback.

REFERENCES

Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering (ISSRE'95)*. 143–151.

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

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.

Nadia Alshahwan and Mark Harman. 2012. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'12)*. 45–55.

Paul Ammann, A. Jefferson Offutt, and Hong Huang. 2003. Coverage criteria for logical expressions. In *Proceedings of the International Symposium on Software Reliability Engineering*. 99–107.

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press, New York, NY.

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.

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'11)*. 265–275.

Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. 2009. MC/DC automatic test input data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'09)*. 1657–1664.

Arthur I. Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja E. J. Vos. 2011. Symbolic search-based testing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. 53–62.

Janvi Badlaney, Rohit Ghatol, and Romit Jadhwani. 2006. *An Introduction to Data-Flow Testing*. Technical Report NCSU CSC TR-2006-22. Department of Computer Science, North Carolina State University, Raleigh, NC 27695. Retrieved from http://people.eecs.ku.edu/~aiedian/Teaching/Fa09/814/Lectures/intro-df-testing-1.pdf.

Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. Symbolic path-oriented test data generation for floating-point programs. In *2013 IEEE 6th International Conference on Software Testing, Verification and Validation*. 1–10.

Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. 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)*. 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, 59–66.

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

Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. TestFul: An evolutionary test approach for java. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. 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'10)*. 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'08)*. 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, 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.

David L. Bird and Carlos Urias Munoz. 1983. Automatic generation of random self-checking test cases. *IBM Syst. J.* 22, 3 (1983), 229–245.

Ilona Bluemke and Artur Rembiszewski. 2009. Dataflow approach to testing java programs. In *Proceedings of the 4th International Conference on Dependability of Computer Systems, 2009 (DepCos-RELCOMEX'09)*. IEEE, 69–76.

Ilona Bluemke and Artur Rembiszewski. 2012. Dataflow testing of java programs with DFC. *Adv. Softw. Eng. Techniques, LNCS* 7054, 3 (2012), 215–228.

Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. 1997. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes* 22, 6 (Nov. 1997), 361–377.

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 *Proceedings of the 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'06)*. 322–335.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

M. L. Chaim. 1991. POKE-TOOL-A tool to support structural program testing based on data flow analysis. *School of Electrical and Computer Engineering, University of Campinas, Campinas, SP, Brazil*.

Marcos L. Chaim, Anthony Accioly, Delano Medeiros Beder, and Marcelo Morandini. 2011. Evaluating instrumentation strategies by program simulation. *IADIS Applied Computing*.

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 8th International Conference on Quality Software (QSIC'08)*. 443.

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. 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*. 52–71.

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

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.

Lee Copeland. 2003. *A Practitioner's Guide to Software Test Design*. Artech House, Norwood, MA.

Roberto Paulo Andrioli de Araujo and Marcos Lordello Chaim. 2014. Data-flow testing in the large. In *Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation (ICST'14)*. 81–90.

Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. 2008. Contextual integration testing of classes. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE'08), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'08)*. 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'09). Companion Volume*. 421–422.

Giovanni Denaro, Alessandro Margara, Mauro Pezzè, and Mattia Vivanti. 2015. Dynamic data flow testing of object oriented systems. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE'15), Volume 1*. 947–958.

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 *Proceedings of the 2009 Joint Conferences on Pervasive Computing (JCPC'09)*. 429–434.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1996. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*. IEEE Computer Society, 575–584.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 992–1030.

Jon Edvardsson. 1999. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*. 21–28.

Lynn M. Foreman and Stuart H. Zweben. 1993. A study of the effectiveness of control and data flow testing strategies. *J. Syst. Softw.* 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 *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'98)*. 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 2nd Conference on Software Development Tools, Techniques, and Alternatives*. IEEE Computer Society Press, Los Alamitos, CA, 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 *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 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.

Kamran Ghani and John A. Clark. 2009. Automatic test data generation for multiple condition and MCDC coverage. In *The Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA'09)*. 152–157.

Ahmed S. Ghiduk. 2010. A new software data-flow testing approach via ant colony algorithms. *Universal J. Comput. Sci. Eng. Technol.* 1, 1 (Oct. 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.

Moheb R. Girgis, Ahmed S. Ghiduk, and Eman H. Abd-elkawy. 2014. Automatic generation of data flow test paths using a genetic algorithm. *Int. J. Comput. Appl.* 89, 12 (March 2014), 29–36.

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, 213–223.

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, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15)*. 1–12.

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 *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10), Workshop Proceedings*. 182–191.

Mary Jean Harrold. 1994. Performing data flow testing in parallel. In *Proceedings of the 8th International Symposium on Parallel Processing*. 200–207.

M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (July 1993), 270–285.

Mary Jean Harrold and Brian A Malloy. 1992. Data flow testing of parallelized code. In *Proceedings of the Conference on Software Maintenance, 1992.* 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 *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 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 Comput. J.* 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.

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 *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom'05)*. 23–39.

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

J. C. Huang. 1979. Detection of data flow anomaly through program instrumentation. *IEEE Trans. Softw. Eng.* 5, 3 (1979), 226–236.

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. 1992. DO-178b: Software considerations in airborne systems and equipment certification. *Requirements and Technical Concepts for Aviation* (December 1992).

Kobi Inkumsah and Tao Xie. 2008. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. 297–306.

Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 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 Univ. J.* 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 *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*. 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, 95–104.

Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. 2010. FloPSy - Search-based floating point constraint solving for symbolic execution. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'10)*. 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.

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'13), part of SPLASH 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)*. 77–81.

Konstantinos Liaskos and Marc Roper. 2008. Hybridizing evolutionary testing with artificial immune systems and local search. In *Proceedings of the 1st International Conference on Software Testing Verification and Validation (ICST'08), Workshops Proceedings*. 211–220.

Konstantinos Liaskos, Marc Roper, and Murray Wood. 2007. Investigating data-flow coverage of classes using evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*. 1140.

Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. 2009. A divergence-oriented approach to adaptive random testing of java programs. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. 221–232.

Chien-Hung Liu, David Chenho Kung, and Pei Hsia. 2000. Object-based data flow testing of web applications. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software, 2000.* 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. *Inf. Softw. Technol.* 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, 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 *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS'02)*. 135–144.

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

Lijun Mei, W. K. Chan, and T. H. Tse. 2008. Data flow testing of service-oriented workflow applications. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering, 2008 (ICSE'08).* IEEE, 371–380.

Lijun Mei, W. K. Chan, and T. H. 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, New York, NY.

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 *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. 156–165.

Akbar Siami Namin and James H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. 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.

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 *Proceedings of the 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 *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 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.

Yu Qi, David Kung, and Eric Wong. 2006. An agent-based data-flow testing approach for Web applications. *Inf. Softw. Technol.* 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, 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, 478–488.

Raul Santelices and Mary Jean Harrold. 2007. Efficiently monitoring data-flow test coverage. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. ACM, New York, NY, 343–352.

Raúl A. Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering, (ICSE'09)*. 56–66.

Raúl A. Santelices, Saurabh Sinha, and Mary Jean Harrold. 2006. Subsumption of program entities for efficient coverage and monitoring. In *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA'06)*. 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, 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. *J. Adv. Sci. Technol.* 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 *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE'15), Volume 1*. 654–665.

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 *Proceedings of the 8th International Conference on Software Security and Reliability (SERE'14)*. 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.

Paolo Tonella. 2004. Evolutionary testing of classes. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. 119–128.

Hasan Ural, Kassem Saleh, and Alan W. Williams. 2000. Test generation based on control and data dependencies within system specifications in SDL. *Comput. Commun.* 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. *Sci. Comput. Program.* 56, 1 (2005), 211–230.

Mattia Vivanti. 2014. Dynamic data-flow testing. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14), Companion Proceedings*. 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 *Proceedings of the Asia-Pacific Software Engineering Conference, 2009 (APSEC'09).* 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. *Inf. Softw. Technol.* 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.

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 *Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003 (COMPSAC'03).* IEEE, 188–197.

Received April 2015; revised September 2016; accepted November 2016