Dynamic Data Flow Testing

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Mattia Vivanti

under the supervision of
Prof. Mauro Pezzè

March 2016

Dissertation Committee

Prof. Walter Binder	Università della Svizzera Italiana, Switzerland
Prof. Nate Nystrom	Università della Svizzera Italiana, Switzerland
Prof. Antonia Bertolino	Consiglio Nazionale delle Ricerche, Italy
Prof. Phil McMinn	University of Sheffield, United Kingdom

Dissertation accepted on 21 March 2016

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Walter Binder
PhD Program Director

Prof. Michael Bronstein
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Mattia Vivanti
Lugano, 21 March 2016

Abstract

Data flow testing is a particular form of testing that identifies data flow relations as test objectives. Data flow testing has recently attracted new interest in the context of testing object oriented systems, since data flow information is well suited to capture relations among the object states, and can thus provide useful information for testing method interactions. Unfortunately, classic data flow testing, which is based on static analysis of the source code, fails to identify many important data flow relations due to the dynamic nature of object oriented systems.

This thesis presents *Dynamic Data Flow Testing*, a technique which rethinks data flow testing to suit the testing of modern object oriented software. *Dynamic Data Flow Testing* stems from empirical evidence that we collect on the limits of classic data flow testing techniques. We investigate such limits by means of *Dynamic Data Flow Analysis*, a dynamic implementation of data flow analysis that computes sound data flow information on program traces. We compare data flow information collected with static analysis of the code with information observed dynamically on execution traces, and empirically observe that the data flow information computed with classic analysis of the source code misses a significant part of information that corresponds to relevant behaviors that shall be tested.

In view of these results, we propose *Dynamic Data Flow Testing*. The technique promotes the synergies between dynamic analysis, static reasoning and test case generation for automatically extending a test suite with test cases that execute the complex state based interactions between objects. *Dynamic Data Flow Testing* computes precise data flow information of the program with *Dynamic Data Flow Analysis*, processes the dynamic information to infer new test objectives, which *Dynamic Data Flow Testing* uses to generate new test cases. The test cases generated by *Dynamic Data Flow Testing* exercise relevant behaviors that are otherwise missed by both the original test suite and test suites that satisfy classic data flow criteria.

Contents

Figures

Tables

Listings

Chapter 1

Introduction

Software testing is the prevalent activity for software quality assurance in most industrial settings. It consists of executing the application with sample inputs and checking the correctness of the outputs to investigate the presence of faults and verify the behavior of the application.

The effectiveness of software testing depends on the thoroughness of the test cases executed on the application. Ideally, an application can be proven absolutely dependable through exhaustive testing, that is, executing and checking every possible behavior of the software. However, since software encodes an enormous amount of behavior, exhaustive testing is not practical, and test engineers have to sample the input space of the application to identify a finite set of test cases. The input space of an application can be sampled in many (typically infinite) ways, and finding a reasonably small set of test cases to execute all the important behaviors of the applications is a difficult task [PY07, Ber07].

Testing techniques help testers select test cases that adequately sample the application execution space by identifying a set of test objectives based either on the functional or the structural elements of the application. Testers can then decide which test cases add and which discard selecting only test cases that satisfy non yet executed test objectives. They can stop testing when all the objectives are executed by the suite.

Functional approaches identify functionalities and usage scenarios of the module under test as test objectives, approximating the thoroughness of the test suites as the relative amount of exercised functionality. Structural testing approaches consider code elements as test objectives, and measure the adequacy of a test suite as the amount of code elements exercised by the test suite with respect to the total amount of elements in the code.

Different structural approaches refer to different elements. Control flow criteria identify test objectives from the control flow graph of the program. For example, the statement adequacy criterion requires executing all program statements, and measures the thoroughness of a test suite as fraction of executed statements, while branch cov-

erage focuses on the coverage of the outcomes of the decision points in the program. Data flow testing criteria focus on data dependencies, and use the relations between variable definitions and uses as test objectives, by for example requiring the execution of all the pairs of definitions and uses of the same variable in the program.

One of the main reasons of success of structural testing approaches is that they can be computed automatically on the source code of the application without relying on other kinds of documents difficult to process and maintain such as software specifications. Control flow criteria have been largely studied and find wide use in industry, also thanks to an extensive tool support. Furthermore, in recent times researchers have proposed automated testing techniques to automatically generate test cases for satisfying control flow elements, which can reduce the high cost of test case writing. However, control flow criteria focus on the control relation between the code elements, ignoring the data relation that becomes prominent when the software behavior depends on the state-based relations between modules, as in the case of modern object-oriented systems.

Data flow testing techniques seems better suited than control flow approaches for testing object-oriented software, since object-oriented behavior is characterised by many data dependencies. By using data relations as test objectives, data flow testing criteria capture the state-based relations among objects which define and read the same class variables. Data flow testing is also particularly promising for the definition of automated testing techniques for object-oriented software, since it identifies test cases that suitably combine method calls that manipulate the objects state [HR94, SP03, DGP08, VMGF13]. Data flow testing criteria have been widely studied, but there is still a lack of evidence of their usefulness. Some empirical results confirm the usefulness of data flow testing, while other work questions their ability to scale to complex industrial size applications [Wey90, FW93, DGP08].

This thesis presents detailed results about the effectiveness of data flow approaches in the presence of complex data flow relations that stem from the interactions between objects; provides original results about the causes of the limited effectiveness of classic data flow testing approaches, and proposes a new technique based on dynamic (data flow) analysis to overcome the limitations of the current approaches.

Our research stems from the observation that classic data flow testing approaches do not cope well with systems characterized by complex data flow relations that depend on the dynamic behavior of the systems, due to the intrinsic limitations of the static analysis that is employed to compute the target data flow abstractions. Data flow analysis of the source code fails to scale to modern inter-procedural systems, whose behavior largely depend on the dynamic allocation of objects. In particular, conservative implementations of data flow analysis both compute too many test objectives to be practical, and include infeasible test objectives in the computation, i.e., infeasible paths from definitions to uses of the same variable. Therefore, existing data flow testing techniques use strong approximations in their analyses to scale to large complex systems, for example

excluding inter-procedural data relations and aliasing [HR94, SP03, DGP08, VMGF13].

This thesis provides empirical evidence that excluding important characteristics of programming languages, like inter procedural control flow and aliasing, leads to the computation of a set of test objectives of limited effectiveness. We show that current data flow testing approaches may miss a huge part of data flow information that is relevant to testing, and provide evidence that the main problem of data flow testing is not the presence of infeasible elements as commonly indicated in the literature, but the imprecision of approximations used to improve the scalability of the techniques.

In this thesis, we re-think data flow testing and propose a new dynamic structural testing approach that is both applicable to complex object-oriented applications and more effective than the existing approaches. We design a new strategy to identify test objectives that capture relevant state-based behavior of the application under test, and we used this strategy to automatically generate effective test cases for object-oriented system. Our technique, namely *Dynamic Data Flow Testing*, is built on top of a new data flow analysis that we refer to as *Dynamic Data Flow Analysis*, which consists of executing the program with one or more test cases and computing precise data flow information on the execution traces. The collected information represents data flow relations that derive from the interplay between the static structure of the program and the dynamic evolution of the system, and thus cannot be identified with classic data flow approaches. *Dynamic Data Flow Testing* infers new test objectives from this data flow information collected dynamically, and automatically generates test cases that satisfy them. These new test cases exercise complex state-based interactions of the application and can detect subtle failures and that would go otherwise undetected.

1.1 Research Hypothesis

The research hypothesis of this thesis is that the data flow information computed by means of a dynamic analysis can identify useful test objectives for guiding the generation of effective test cases for object-oriented systems.

We hypothesize that dynamically-computed data flow test objectives capture relevant state based interactions between objects that are missed by state-of-the-art techniques based on static analysis, and that the newly captured interactions are important to test to expose corner-case failures that depend on object interactions. We also hypothesize that the low applicability and effectiveness of existing data flow approaches depends on the underlying static data flow analysis, and that we can obtain a more effective data flow testing technique by exploiting dynamically computed data flow information.

1.2 Research Contributions

The main contribution of this thesis is the definition of *Dynamic Data Flow Testing*, a systematic testing technique that exploits dynamically computed data flow information to select test cases that elicit the complex state-based behavior of object-oriented systems.

In particular, our work on *Dynamic Data Flow Testing* contributes to the state of the research by:

- providing a precise quantitative analysis of the limits of the current data flow testing approaches based on static data flow analysis which under-approximates the dynamic behavior of software;
- defining *Dynamic Data Flow Analysis*, a dynamic implementation of data flow analysis for object-oriented systems that computes precise data flow information while executing a program;
- proposing *Dynamic Data Flow Testing*, a testing technique for systematically selecting effective test cases for object-oriented systems leveraging dynamic data flow information computed by *Dynamic Data Flow Analysis*. *Dynamic Data Flow Testing* iteratively improves an existing test suite: At each iteration (A) dynamically analyzes the program executed with the current test suite by means of *Dynamic Data Flow Analysis* to compute data flow information, (B) statically combines the data flow information to identify new test objectives, and (C) either selects or generates new test cases that satisfy the new objectives. The new test cases are added to the current test suite for the new iteration.
- providing experimental data about the effectiveness of the proposed technique, which confirm our research hypotheses.

1.3 Structure of the Dissertation

Chapters 2 and 3 focus on *static* data flow analysis and testing and on their limits. Chapter 2 describes the fundamental concepts of data flow analysis and its application to object-oriented systems. Chapter 3 introduces the classic data flow testing approach and describes its limits.

Chapters 4 and 5 focus on *Dynamic Data Flow Analysis* and its evaluation. Chapter 4 formalizes *Dynamic Data Flow Analysis* emphasising the differences with respect to static data flow analysis, and Chapter 5 presents an experiment that compares dynamic and static data flow analysis, which motivate our idea of using dynamic data flow analysis to select test objectives.

Chapters 6 and 7 present and evaluate *Dynamic Data Flow Testing*. Chapter 6 describes *Dynamic Data Flow Testing* in its three steps, the instantiation of *Dynamic Data Flow Analysis*, the inference of test objectives from the dynamically collected data flow

information and the selection of test cases to augment an existing test suite. Chapter 7 describes the prototype implementation of *Dynamic Data Flow Testing* and reports about the evaluation studies we conducted.

Chapter 8 surveys the related work on automated systematic testing techniques and test case generation approaches based on the software structure. Finally, Chapter 9 summarizes the results of the dissertation and outlines conclusions and future directions.

Part of the work on *Dynamic Data Flow Analysis* described in Chapters 4 and 5 appeared in the proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2014 [DPV14], while part of the work on *Dynamic Data Flow Testing* described in Chapters 6 and 7 appeared in the proceedings of the 37th International Conference on Software Engineering (ICSE) in 2015 [DMPV15].

Chapter 2

Data Flow Analysis

Data flow techniques analyze how data are propagated in a program, emphasizing how information is generated, flows and is used from a program point to another. Data flow analysis is applied in many domains ranging from compiler optimization, to security analysis and software testing. This chapter describes the terminology and fundamental concepts of data flow analysis, focusing on its application to object-oriented systems.

Data flow analysis refers to a body of software analysis techniques that compute how data propagate in a software application, emphasizing how information is generated, flows and is used from a program point to another. Data flow analysis constitutes the core of data flow testing approaches, where is used to identify test objectives.

This chapter introduces the fundamental concepts of data flow analysis. Section 2.1 introduces data flow analysis concepts and terminology: definitions, uses, definition use pairs and reaching analysis. Section 2.2 presents data flow analysis and abstractions for object-oriented systems. Section 2.3 discusses the limits of static data flow analyses in dealing with some programming languages characteristics.

2.1 Intra- and Inter-Procedural Data Flow Analysis

Data flow abstractions make it explicit how values are defined and used in a program, and how program points that define and use the same variables depends on each other. The fundamental data flow abstractions are definitions, uses and definition use pairs [ASU86].

A *definition* of a variable occurs when a value is assigned to the variable, and a *use* occurs when the value stored in the variable is read. For example, in Listing 2.1 the assignment at line 2 is a definition of the variable `fact`, and the return statement at line 6 is a use of the same variable `fact`. The same variable may be defined and used in the same statement, for instance, the increment operators `i++` at line 3 both uses and defines the variable `i`.

Listing 2.1. Code example

```
1  int n = read();
2  int fact = 1;
3  for (int i = 1; i <= n; i++) {
4     fact *= i;
5  }
6  return fact;
```

Definition use pairs capture the data dependencies between program points that define a value, and program points where that value may be accessed. They associate definitions with uses of the same variables when there is at least one program execution in which the value produced by the definition may be read by the use without being overwritten.

Definition use pairs can be defined in terms of paths of the control flow graph of the application. A definition use pair of a variable v is a pair (d,u) where d is a program point that contains a definition of v, u a program point that contains a use on v, and there exist at least one control flow path between d and u with no intervening definition of v. In this case we say that d *reaches* u, and that there is a *definition-clear path* between d and u. On the contrary, if a definition d' of a variable v overwrites a previous definition d of the same variable, then the former definition is *killed* by the latter on that path.

For instance, in Listing 2.1 the definition and use of the variable `fact` that occur at lines 2 and 4 form a definition use pairs. When the variable `n` is greater than 1, the definition of `fact` at line 4 *kills* the definition of `fact` at line 2. But since `n` could be less than 1, the definition of `fact` at line 2 *reaches* also its use at line 6, forming another definition use pair. Table 2.1 reports all the definition use pairs of the example.

The definition use pairs of a given program could be computed by analyzing the propagation of each individual definition along all the paths of the control flow graph of the program. This approach requires many traversals of the graph and is in general inefficient [ASU86].

Data flow analysis provides a more efficient algorithm for computing definition use pairs, which does not require to traverse every individual path, but summarizes data flow information at a program point over all the paths reaching that program point. In particular, the analysis that enables the computation of definition use pairs is called *reaching definition analysis*, and computes for each program point the set of definitions that *may* reach that program point.

Reaching definitions analysis defines a fixed point algorithm that exploits the way reaching definitions at a node are related to reaching definitions at an adjacent node to compute data flow information efficiently. Reaching definitions that reach the beginning of a node n can be evaluated as the union of all the definitions that reach the exit points of the predecessors of n. The reaching definitions that reach the exit of the

Table 2.1. Definition use pairs found in the code snippet in Listing 2.1

Variable Name	Definition		Use	
	Code	Line	Code	Line
n	`int n = read();`	1	`i <= n;`	3
fact	`int fact = 1;`	2	`fact *= i;`	4
fact	`int fact = 1;`	2	`return fact;`	6
fact	`fact *= i;`	4	`fact *= i;`	4
fact	`fact *= i;`	4	`return fact;`	6
i	`for (int i = 1;`	3	`i <= n;`	3
i	`for (int i = 1;`	3	`fact *= i;`	4
i	`for (int i = 1;`	3	`i++)`	3
i	`i++)`	3	`i <= n;`	3
i	`i++)`	3	`fact *= i;`	4
i	`i++)`	3	`i++)`	3

node n instead are evaluated as the difference between the definitions that reach the entry points of n and the definitions killed in n, plus the new definitions generated in n itself.

Formally, the algorithm works on four sets of information about each node of the control flow graph: GEN, $KILL$, $ReachOut$ and $ReachIn$. GEN and $KILL$ denote the sets of definitions that start and stop propagating in a node, they are computed simply inspecting the definitions of each node, and are passed as input to the algorithm. $ReachOut$ and $ReachIn$ denote the set of prior and next reaching definitions, respectively, and are produced by the algorithm, which computes them by repetitively applying the data flow Equations 2.1 and 2.2 on the graph until the result stabilizes.

$$ReachIn[n] = \bigcup_{p \in pred(n)} ReachOut[p] \tag{2.1}$$

$$ReachOut[n] = ReachIn[n] \setminus KILL[n] \cup GEN[n] \tag{2.2}$$

Figure 2.1 shows an example of the results of reaching definitions analysis performed over the program reported in listings 2.1. For each node of the control flow graph, the figure reports the initial sets *GEN* and *KILL*, and the sets *ReachIn* and *ReachOut* computed by the algorithm.

Once reaching definition information is computed, it is possible to evaluate definition use pairs by examining each node of the control flow graph and matching the definition that reach the node entry with the uses present in the node.

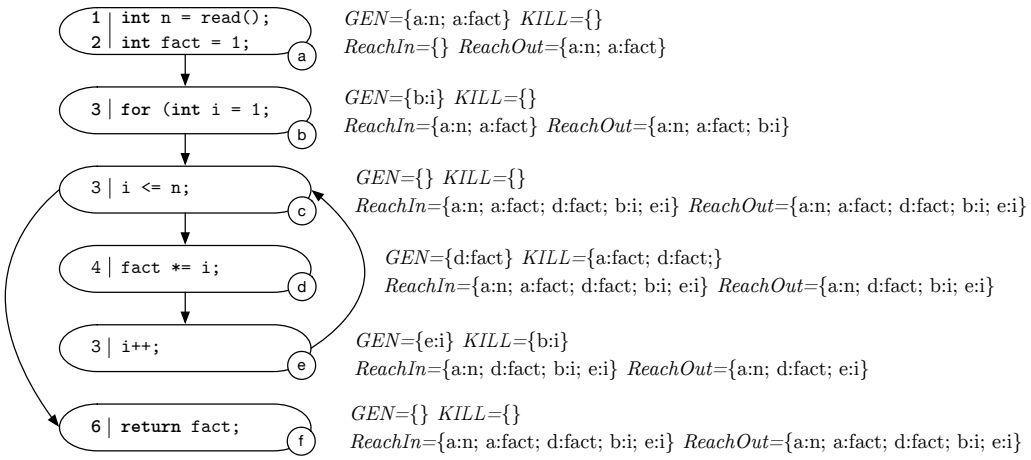


Figure 2.1. CFG of the code snippet in Listing 2.1 annotated with information on reaching definitions.

2.2 Data Flow Analysis of Object-Oriented Systems

The structure of object-oriented programs differs from that of the procedural programs originally targeted by data flow analysis techniques. The behavior of procedural programs mostly depends on procedures with a complex control flow. On the contrary, the distinctive characteristics of the object-oriented paradigm are classes and objects, and the behavior of object-oriented software depends on how objects interact with each other through their methods and variables.

This section describes how data flow abstractions and analysis can be extended to model the data relations between objects. Firstly, data flow analysis has to be extended to propagate definitions over the inter-procedural control flow that arises from the method interactions within and between classes. We discuss this in Section 2.2.1.

Secondly, object-oriented design principles encourage the encapsulation of object states: objects can refer to other objects as part of their state leading to nested state-based relationships between them. Section 2.2.2 describes how we can model the data dependencies that arises from object encapsulation by extending definition use pairs with the information of the invocation context.

2.2.1 Intra Class Data Flow Analysis

Data flow relations in a class occur both within and across methods that share information either passing values as parameters or operating on the same shared variables (class fields) [HR94].

Definition use pairs can involve local variables defined and read within a method (intra-method pairs). For example, in the code snippet in Listing 2.2 that includes a

Listing 2.2. `CoffeeMachine` Class Example

```
1  class CoffeeMachine{
2    Inventory i = new Inventory();
3    int price, amtPaid;
4
5    public void makeCoffee(boolean sugar){
6      boolean canMakeCoffee=true;
7      if(amtPaid<price || i.getCoffee()<=0
8        || i.getSugar()<=0)
9        canMakeCoffee=false;
10     if(canMakeCoffee){
11       Coffee c = new Coffee();
12       if(sugar)
13         addSugar(c);
14       amtPaid-=price;
15       i.consumeCoffee();
16       }
17   }
18
19 public void makeTea(boolean sugar){
20     boolean canMakeTea=true;
21     if(amtPaid<price || i.getTea()<=0
22       || i.getSugar()<=0)
23       canMakeTea=false;
24     if(canMakeTea){
25       Tea c = new Tea();
26       if(sugar)
27         addSugar(c);
28       amtPaid-=price;
29       i.consumeTea();
30       }
31   }
32
33
34    private void addSugar(Coffee c){
35      c.addSugar();
36      i.consumeSugar();
37    }
38
39   public void addCoins(int amt){
40     amtPaid+=amt;
41    }
42
43   public void addInventory(int coffee,
44    int sugar){
45     i.setSugar(sugar);
46     i.setCoffee(coffee);
47    }
48 }
49
50 class Inventory{
51 int sugar;
52 int coffee;
53 int tea;
54  public void setSugar(int s){
55    sugar=s;
56  }
57  public void consumeSugar(){
58    sugar--;
59  }
60  ...
61 }
```

simple Java program that models the behavior of a coffee machine, the definition and use of the variable `canMakeCoffee` at lines 6 and 10 form an intra-method definition use pair. Intra-method pairs can be detected on the control flow graph of a method applying the classic data flow analysis introduced in the previous section.

Definition use pairs arise also from the inter-procedural control flow of different methods that directly call each other, or that define and use a class field. We define an *inter-method* pair as a pair whose definition and use occur in two different methods that call each other, and that share the defined and used variable as parameter, for example the definition and use of variable `c` at lines 11 and 35 in Listing 2.2. We define an *intra-class* pair as a pair whose definition and use occur in two different public methods that define and use the same class field [HR94]. For example, the definition and use of the class field `CoffeeMachine.amtPaid` at lines 40 and 7 in Listing 2.2.

Inter-method and intra-class pairs can be computed with an inter-procedural extension of data flow analysis, addressing the inter-procedural control flow between the methods of one or multiple classes. Inter-procedural data flow analysis works by propagating data flow equation over an inter-procedural control flow graph called *class*

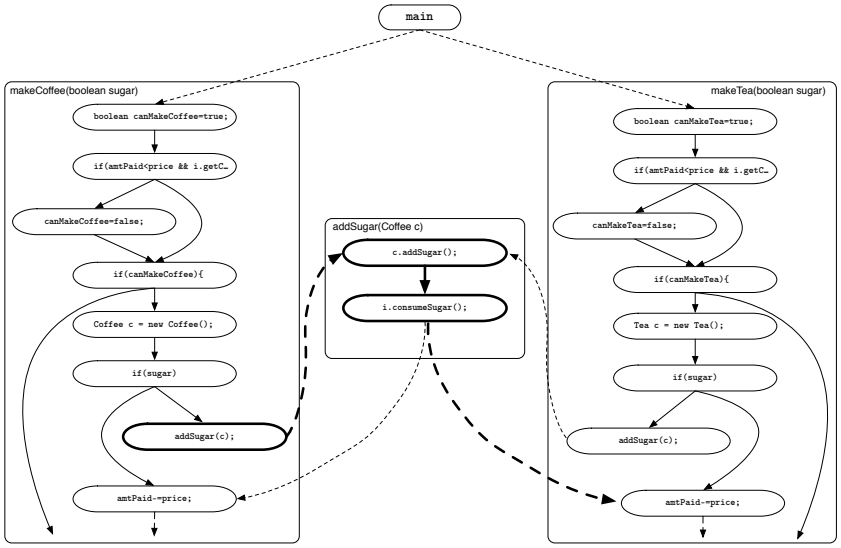


Figure 2.2. Extract of the CCFG of the class CoffeeMachine of Listing 2.2, that focus on the interaction between the methods makeCoffee, makeTea and addSugar.

control flow graph (CCFG). The class control flow graph connects control flow graphs of individual methods by adding special edges from nodes that correspond to method invocations to the entry nodes of the called method, and from the exit nodes to nodes that correspond to return from the invocation. A special entry node is used to represent the entry of an hypothetical "main" method that can call each method sequence in any arbitrary order [PY07, HR94]. An example of CCFG is reported in Figure 2.2, showing the CGFG of the methods `CoffeeMachine.makeCoffee()`, `CoffeeMachine.makeTea()` and `CoffeeMachine.addSugar()` of Listing 2.2.

CGFGs as defined above do not take into account the context of the different calls. Thus, a CGFG may contain paths that do not match the program call-return semantics, that is, paths in which a call does not match the return. Propagating data flow equations on these paths could produce spurious information.

For example in the graph in Figure 2.2, we highlighted in bold a spurious path that connects the node `addSugar(c)` of method `CoffeeMachine.makeCoffee()` to the node `amtPaid-=price` of method `CoffeeMachine.makeTea()`, passing through method `CoffeeMachine.addSugar()`. This path does not match the call-return semantics between the methods, since at runtime the method `CoffeeMachine.addSugar()`, when called from `CoffeeMachine.makeCoffee()`, will never return to the method `CoffeeMaker.makeTea()`.

An inter-procedural data flow analysis that propagates data flow equations ignoring the context of the method calls is referred to as *context-insensitive*, because it does not distinguish the calling context of a called procedure when jumping back to the original call site [ASU86]. Context-insensitive analysis propagates data flow equations also on

the spurious paths of the CCFG, potentially identifying invalid information. However, can be implemented efficiently, and allows the analysis to scale on large programs. On the contrary, a *context-sensitive* inter-procedural data flow analysis considers the calling context when analyzing the target of a function call, and correctly jumps back to the original call site of a called procedure. *Context-sensitive* analysis is more precise than context-insensitive one, however, the number of contexts can be exponential the number of procedures in the program. Although efficient algorithms exploits summaries to store information for each procedure and lazily re-analyze the context only if it has not been analyzed before in that specific context, context-sensitive analyses can be costly to apply on large complex systems; and many works that employed a data flow analysis relied on a context insensitive implementation that provides a better scalability [HR94, SP03, DGP08, VMGF13].

2.2.2 Inter Class Data Flow Analysis

In object-oriented programming, the class state is an assignment of the attributes (the fields) declared in a class, in the context of some objects that instantiate the class. A class state is structured if the class includes at least an attribute with a non-primitive value, that is, an attribute defined as a data structure, possibly declared with reference to the type of other classes. For example, the state of the class `CoffeeMachine` in Listing 2.2 includes an object of type `Inventory` (the field `CoffeeMaker.i`), which contains the three primitive fields `Inventory.sugar`, `Inventory.coffee` and `Inventory.tea`. At runtime, the state of an object `CoffeeMachine` depends on the internal state of the object `Inventory` referenced by `CoffeeMachine.i`, that is, on the values of the variables `Inventory.sugar`, `Inventory.coffee` and `Inventory.tea`. A definition or an use of these variables will affect not only the object `Inventory`, but also the (nested) state of the object `CoffeeMachine`.

To better express the nested-state relationships between classes, we can represent the class state as a set of *class state variables*, each corresponding to an assignable (possibly nested) attribute that comprises the state of the class under test. State variables are identified by the class they belong to and the chain of field signatures that characterize the attribute. In the example just discussed, the class state variables of `CoffeeMachine` are `CoffeeMachine.i.sugar`, `CoffeeMachine.i.coffee` and `CoffeeMachine.i.tea`.

We can model data dependencies that arise from the nested state of classes extending definitions and uses with the information on the *context* of the method invocations, where the context of methods invocation is the chain of (possibly nested) method invocations that, from the class, leads to the definition or the use of the class state variable [SP03, DGP08]. We formally define *class state variables* and *contextual* definitions, uses and definition use pairs in Definitions 2.1, 2.2 and 2.3. These definitions extend the original definitions proposed by Souter and Pollock and Denaro et al. [SP03, DGP08].

Definition 2.1 (*Class State Variables*)**.** A class state variable is a pair $\langle class\_id, field\_chain \rangle$, where $class\_id$ is a class identifier and $field\_chain$ is a chain of field signatures that navigates the data structure of the class up to an attribute declared therein.

Definition 2.2 (*Contextual definition (use)*)**.** A contextual definition (use) cd (cu) of a class c is a tuple $(var, \langle m_1, m_2, ..., m_n \rangle, l)$ where var is the defined (used) class state variable of c, l is the location of the actual store (load) instruction that modifies (references) the nested state of c, and $\langle m_1, m_2, ..., m_n \rangle$ is a call sequence where m_1 is the call site of the first method call leading to a modification (reference) of the state of c, and m_n is the method containing l.

Definition 2.3 (*Contextual definition use pairs*)**.** A contextual definition use association for a class state variable var is a pair (cd, cu) that associates a contextual definition cd of var, with a contextual use cu of var, which have at least one def-clear paths between them.

For example, just consider the following contextual definition use pair of the class state variable `CoffeeMachine.i.sugar` computed over the two classes `CoffeeMachine` and `Inventory` of Listing 2.2:

	Context	Line
Definition	CoffeeMachine::addInventory(int, int) [45] → Inventory::setSugar(int)	55
Use	CoffeeMachine::addSugar(Coffee) [36] → Inventory::consumeSugar()	58

Where → indicates method calls and the numbers surrounded by square brackets are the lines where the calls occur.

The invocation of the method `i.setSugar(int)` in the method `addInventory(int, int)` at line 45 leads to a definition of the variable `sugar` of `i`. Similarly, the invocation of `i.consumeSugar()` in the method `addSugar(Coffee)` at line 36 leads to a use event over the variable `sugar` of `i`. This definition use pair captures the data dependencies between the methods `addInventory(int, int)` and `addSugar(Coffee)` that define and read the internal state of the class `CoffeeMachine`, that is the variable `CoffeeMachine.i.sugar`.

Contextual data flow information is computed with an inter-procedural data flow analysis that propagate data flow equations over multi-class extension of the class control flow graph, which models the control flow between methods of different classes [SP03, DGP08]. The construction of the inter-procedural graph on object-oriented systems, which extensively use virtual call invocations instead of static ones (caused by polymorphism and in general dynamic binding), presents well-known challenges and impact on the applicability and on the precision of the analysis. We discuss the problems that arise from the presence of dynamic binding in the next section.

2.3 Dealing with Dynamic Features

Static program analysis techniques such as data flow analysis have precision problems when applied over programs which behavior cannot be statically known without executing the application.

Some programming language features, such as pointers and dynamic binding, define software behavior that depends on the runtime status of the program. This *dynamic* behavior of software cannot be encoded precisely by statically analyzing the source code, but can be deduced only observing the program execution. In these cases, static analyses have to abstract this dynamic behavior into decidable over- or under-approximations on which it is possible to reason statically [ASU86].

An analysis over-approximates the dynamic behavior of software when it conservatively includes all the (statically detectable) possible behavior in the application model. For example, analyses that over-approximate the application execution model considers as executable all the statically computed paths in a control flow graph, even if not all the paths in a CFG represents possible executions.

Analyses that under-approximates the dynamic behavior of software model the application abstracting out behavior. For example, an analysis could overlook the presence of reference aliases to reduce the size and the complexity of the model.

Both under- and over-approximations impact on the precision and the cost of the analysis. In particular, the approximations impact differently on the *soundness* and *completeness* of a data flow analysis, which we define below:

Soundness (Data Flow Analysis): A sound data flow analysis is an analysis that identifies only executable data flow abstractions, for instance, only definition use pairs that can be observe at runtime when executing the program. A sound data flow analysis may fail to detect some data flow abstractions.

Completeness (Data Flow Analysis): A complete data flow analysis is an analysis that identifies all the executable data flow abstractions without missing any. A complete data flow analysis may include non executable data flow abstractions in the results.

Analyses that over-approximate the dynamic behavior of software usually detect all data flow relations, but they may include non observable behavior in the application model, thus introducing unsoundness in the results. Over-approximations may also negatively affect the scalability of the analysis, because considering all the statically detectable behavior of systems with a complex structure can exponentially increase the analysis cost.

Under-approximations may negatively affect both the soundness and the completeness of the analysis, since abstracting the behavior of the application may produce both incomplete and unsound information. Under-approximations reduce the complexity of

the analysis, thus improving scalability and performance, and can therefore be cost-effective when applied in contexts where a certain degree of imprecision is tolerated.

Choosing the appropriate approximations to deal with the dynamic behavior of software is one of the main challenges when designing a static data flow analysis. Too strong approximations make the analysis too imprecise to be useful. On the other hand, handling some complex language features without under-approximations would make the analysis unscalable to the point of being useless.

Below we describe the main characteristics of programming languages that are approximated in static data flow analyses, focusing on the common over- and under-approximations that are implemented in existing approaches in literature, and on the impact that these approximations have on the soundness and completeness of the analysis.

2.3.1 Arrays

Array elements are accessed through indexes which values are either statically declared or evaluated at runtime. In this latter case, a static analysis cannot determine wether two indexes are always, never or sometime equal without executing the program, introducing imprecision in the results. For example, in the code snippet in Listing 2.3 the indexes i and j depend on the runtime of the application. They may point to the same value (triggering a definition use pair on the value referenced by the array at line 1 and 2), or never be equal in any execution.

Listing 2.3. Example of arrays

```
1 | a[i] = 0;
2 | x = a[j];
3 | a[5] = 5;
```

To limit the uncertainty introduced in the results, static data flow analysis treats arrays in different ways trading off conservativeness for scalability and soundness.

The most conservative strategy distinguishes single definitions and uses of each cell of the array. When the index is not statically known, like `i` and `j` in the example of Listing 2.3, the definition or access of a cell is considered a possible definition or access of *any* cell, over-approximating the possible behavior of the application. This is an example of complete but unsound analysis. The analysis conservatively includes all the feasible data flow abstractions of arrays elements, but it will also include a likely high number of infeasible abstractions. Intuitively, a single array with many elements could generate an exponential number of infeasible data flow relations, leading to scalability problems and weakening the usefulness of the result of the analysis.

A less conservative approach abstracts arrays as single entities that are defined every time any cell is modified, and used every time any cell is read. This abstraction makes

the results of a data flow analysis both unsound and incomplete. For example, the assignment at lines 3 in Listing 2.3 will *always* kill the definition at line 1, even if at runtime the write events could occur on different memory locations. Even if this approach introduces a strong under-approximation, it is scalable and often considered a fair trade off when the analysis is used in the context of testing.

Finally, arrays could also be overlooked by the analysis. In this case the only definition use pairs identified over arrays are between the creation of the array itself (a definition of the array reference) and any access to the array for both writing or reading a cell (an use of the array reference). This approach result in a (very) unsound and incomplete analysis.

2.3.2 Pointers

There is an alias relation whenever a memory location is referred using different names. This particular behavior, which depends from programming languages constructs such as pointers and object references, has a strong impact on the soundness of a data flow analysis.

Data flow analysis distinguishes definitions and uses in the source code identifying them according to their variable name, that is, it assumes that two accesses refer to the same memory location only when they occur though the same variable. In the presence of aliases, however, this assumption does not hold, and data flow events detected on different variables could in reality interest the same storage location.

Listing 2.4. Example of pointers

```
1 int i = 0;
2 int *intptr = &i;
3 *intptr = 10;
4 printf("\%d\n", i);
```

For example, the code snippet in Listing 2.4 illustrates a very simple case of aliasing due to pointers: the pointer `*intptr` (line 2) is an alias of variable `i`. Variable `i` is initially assigned to 0, and its value is later set to 10 through the pointer (line 3). A static data flow analysis is not able to decide whether two different variables point to the same memory location, and thus can share a data flow relation, or not. For instance, a data flow analysis might miss the fact that, in Listing 2.4, the definition through the pointer `*intptr` at line 3 kills the definition of `i` at line 1.

Another example are arrays of pointers that refer to the same element in multiple cell locations, for instance, given an array a we have an alias when we store the same element e in the cell $a[1]$ and $a[2]$.

To identify aliases and increase the soundness of the results, data flow analysis can be complemented with an alias analysis, that is, a static analysis that resolves the alias relations of a program.

Alias analyses compute either *may* or *must* alias information. May alias information report a *possible* alias relation between two or more pointers during program execution. A may alias analysis (or *point-to analysis*) conservatively computes a superset of all possible aliases over-approximating the heap locations that each program pointer may point to. Doing so, a may alias analysis produces complete information, but introduces unsoundness in the form of aliases that may be impossible to actually observe executing the program.

Must alias analysis identifies a necessary (*must*) alias relation between two or more pointers at some point during the program execution. A must alias analysis produces sound but potentially incomplete results, since it may be not powerful enough to identifying all the must aliases that exist at runtime, but statically computes only a subset of must aliases.

There are many may and must alias analyses, ranging from relatively simple abstractions to very complex techniques [LH99]. Most scalable alias analyses are *flow insensitive*, they assume that the statements can execute in any order and any number of time. Flow insensitive alias analyses are computationally efficient, at the cost of being imprecise. More precise but computationally expensive analyses are *flow* and *context sensitive* analyses that consider the correct execution order of both statements and procedures.

Surveying alias analyses is beyond the scope of this thesis, however it is important to note that also in the context of alias analyses a tradeoff between precision, scalability and usefulness of the analysis is present. Flow and context sensitive analyses may not scale to industrial-size program, while scalable approaches may be approximated to the point of being of limited usefulness.

When we complement a data flow analysis with an alias analysis, the completeness, soundness and scalability of the analysis depends on the completeness and soundness of the used alias information, and on the cost of the alias analysis. Incorporating may alias information in data flow analysis increases the completeness of the analysis, and can be computed using a flow insensitive analysis with a limited impact on the cost and scalability of the analysis. However, may alias analyses, especially flow insensitive ones, compute infeasible alias relations that may lead to an explosion of the number of infeasible data flow abstractions, reducing the soundness of the data flow information and its usefulness, for example, for testing.

Must alias information can increase the soundness of a data flow analysis, since it includes only feasible alias relations that increase the precision of the results. Must alias information can avoid computing infeasible data flow relations that depends on the propagation of definitions that at runtime are always killed using an alias, like the

infeasible pair between line 1 and line 4 in Listing 2.4. However, must alias analyses typically require flow and context sensibility, thus can introduce a significant overhead on the data flow analysis, and limit its scalability.

Incorporating *only* must alias information in data flow analysis could increase the analysis soundness avoiding an explosion of infeasible elements. However such analysis may be incomplete to a large extent, and it may miss a large number of feasible data flow abstractions that depend on may aliases. Whether this under-approximation could be considered acceptable depends on which use the analysis is intended for.

Despite the large amount of research, and the pervasiveness of aliases in most programming languages, most of the approaches that rely on data flow analysis do not include alias analysis [HFGO94, FWH97, OPTZ96, FWH97, DGP08, HA13, VMGF13]. Overlooking aliases while performing data flow analysis can have a huge impact on the soundness and completeness of the analysis, and can directly affect the effectiveness of a technique that relies over data flow information.

2.3.3 Paths

Data flow analysis typically over-approximates the behavior of the application by assuming that every conditional can always evaluate as both true and false, introducing a degree of unsoundness in the results. For example, it may be the case that a program path identified on a control flow graph traverses some conditionals that at runtime always evaluate as false, for instance in the case of defensive programming. In this case, the identified path is spurious, or infeasible, it is statically identified but impossible to be executed at runtime by any input to the program.

Including spurious paths in the analysis may lead to computing infeasible data flow abstractions, which make the analysis unsound [Wey90, FW93]. For instance, if a reaching definitions analysis propagates values over infeasible paths, it may compute an *infeasible definition use pair* that identifies as data dependent two program points that at runtime share no dependency.

Consider again, for example, the code snippet in Listing 2.1, and its CFG reported in Figure 2.1. Assume that the function `read()` always returns a number greater than 0, therefore the program will *always* enter the `for` loop and will define `fact` at line 4 that kills the defintion of `fact` at line 2. In this case, the path that traverses the nodes a, b, c and f is infeasible, as it is infeasible the definition use pair between the definition and the use of `fact` at line 2 and 6.

Although, in general, the problem of deciding whether a path is feasible or not is undecidable, some techniques can detect subsets of infeasible paths. These techniques are based either on symbolic execution, model checking or on pattern matching [DBG10, BHMR07, NT07, NT08]. Techniques based on symbolic execution and model checking submit path conditions to theorem provers to verify the satisfiability

of the path. Techniques based on pattern matching analyse the source code for common programming patterns that lead to infeasible paths. These techniques are however particularly expensive to apply and to integrate in a data flow analysis.

The state of practice in data flow analysis is to tolerate the unsoundness that depends on the presence of infeasible paths. Some attempts have been done to integrate techniques for infeasible path detection and data flow analysis, but the applicability of these techniques is strongly limited by the cost of the analysis [SFP+15].

The infeasibility problem has important implications when using data flow analysis for testing, since infeasible definition use pairs identify test objectives that are impossible to execute and may divert the testing effort, as discussed in detail in Section 3.3.

2.3.4 Dynamic Binding

Dynamic binding is a programming mechanism that allows to select which implementation of a polymorphic operation to call at runtime, differently from static binding that fixes all types of variables and expressions in the compilation phase.

Dynamic binding is a characteristic feature of object-oriented software: the object-oriented paradigm encourages declaring variables and parameters using superclasses and interfaces, and invoking on them methods that are overridden in their subclasses, which concrete type is determined at runtime. Advanced programming languages features such as virtual method invocations and reflection, which allows the developer to perform runtime actions given the descriptions of the objects involved, and which are implemented in popular object-oriented languages like Java, also define dynamic binding relationship between calling objects that are undecidable to resolve statically.

Software systems written exploiting programming languages features for dynamic binding have, therefore, an inter-procedural control flow which largely depends on the dynamic behavior of the application. In this case, the source code does not contain enough information for statically resolving the binding relations between call points and called elements, and for statically identifying which parameters and variables are passed between procedures. As a result, the precise static construction of the inter-procedural control flow graphs is generally undecidable, affecting the soundness of a static analysis based on that inter-procedural control flow graph.

The problem of statically resolving calls to procedures or methods that depends on dynamic binding translates to an aliasing problem: we have to identify to which memory locations some variables and references could point to. It requires an alias analysis to over- or under- approximate the possible states of the application, and then constructing the inter-procedural control flow graph according to the chosen approximation.

It is important to note that dynamic binding is pervasive in many applications, and combined with other forms of aliasing causes an explosion of the complexity of an inter-procedural analysis.

Pairing data flow analysis with alias analysis is necessary to produce a sound representation of the inter-procedural control flow of software, but implementing the correct level of approximation for balancing soundness and scalability is particularly challenging when dynamic binding is pervasive in the application, and with no perfect solution [Ryd03]. Over-approximations can lead to the identification of an exponential number of infeasible paths on the inter-procedural control flow graph, while under-approximations may lead to missing significative parts of the application behavior.

In the following we illustrate the problems involved with the static representation of the inter-procedural control flow in the presence of dynamic binding with the Java example of Listing 2.5.

We use the example to illustrate an inter-procedural reaching definition analysis when the expected results depend on the possible dynamic bindings of some method calls and on the occurrence of aliases between program variables and objects in memory.

In the Java program of Listing 2.5, the field `nest.i` is defined by invocations of the methods of the classes `Nest` (lines 27 and 28) and `NestA` (line 32). These definitions propagate intra-procedurally to the end of the respective methods. We denoted these definitions as D_0, D_1 and D_2, respectively. In Listing 2.5, we have annotated the exit of the methods with a comment that indicates the definitions propagated in each method, that is, the definitions that are possibly executed within the execution flow of that method and not yet overridden by any subsequent assignment until the exit of that method. For example, executing method `Nest.n()` and `NestA.n()` would propagate D_1 or D_2, respectively, while executing method `NestB.n()` would propagate the definition D_0 that is active at that point because of the execution of the constructor of class `Nest`.

The definitions D_0, D_1 and D_2 also propagate to the methods `m1`..`m3` through the calls to the methods `n`. In these cases, the propagation of the definitions depends on the dynamic binding of the method calls. For example, the call to the method `n` in method `m1` at line 7 can be dynamically bound to any of the methods of the classes `Nest`, `NestA` or `NestB`, and can thus result in the execution of the definitions at lines 27, 28 or 32, and all the three definitions can propagate to the exit point of method `m1`. Similarly, the calls to the method `n` in the methods `m2`..`m3` can be dynamically bound to different sets of methods in `Nest`, `NestA` or `NestB`, and can thus propagate different definitions to the exit of the methods, discussed in details in the examples below.

The methods `m1`, `m2` and `m3` exemplify the different aspects of the impact of the dynamic binding of method calls on static analysis.

Method `m1` may call `nest.n()` at line 7 that, depending on the runtime type of the object `nest`, can result in executing any method out of `Nest.n()`, `NestA.n()` or `NestB.n()`. Accordingly, the execution of method `m1` can propagate any definition out of D_0, D_1 and D_2 to the exit of the method. When considering only the *static type* `Nest` declared for the reference `nest`, we may erroneously conclude that only D_1 propagates

Listing 2.5. A sample Java program

```
1  class ClassUT {
2      private Nest nest;
3      ClassUT(Nest n){
4          nest = n;
5      }
6      void m1(int p){
7          if (p<0) nest.n();
8      }//D0, D1, D2
9      void m2(){
10         if (!(nest instanceof NestA)) return;
11         nest.n();
12     } //D2
13     void m3(int p){
14         if (!(nest instanceof NestB)) return;
15         m1(p);
16     }//D0
17
18      //...
19
20      void doSomething(){
21         int v = nest.i;
22          /do some computation with v/
23     }
24 }
25 class Nest{
26     protected int i;
27     Nest(){i = 0;} //D0
28     void n(){i = 1;} //D1
29     Nest f(){/ lot of code; / return this;}
30 }
31 class NestA extends Nest{
32     void n(){i = 2;}//D2
33 }
34 class NestB extends Nest{
35     void n(){}//D0
36 }
```

until the exit of `m1`, because of the call to `Nest.n()`. To correctly identify the possible flows of data related to method `m1`, a static data technique must know the possible dynamic bindings of the call `nest.n()`. In cases like this, we can statically identify the correct bindings with a simple and efficient *flow-insensitive may-alias analysis*, since in this case `nest` could be of any subtype of `Nest`.

Method `m2` shows that the use of flow-insensitive information may not be sufficient in general, since it can produce over-approximated results. Executing method `m2` may lead to calling method `nest.n()` at line 11, if the condition at line 10 holds. But this condition restricts the dynamic type of `nest` to `NestA`, and thus only definition D_2 can propagate to the exit of method `m2`. In cases like this, flow-insensitive may-alias analysis over-approximates the behavior of `m2`, identifying the propagation of the (infeasible) definitions D_0 and D_1, because their propagation is constrained by flow-sensitive infor-

mation. *Flow-sensitive alias analysis* can provide the information needed to correctly identify the dynamic propagation of the definitions (that is, D_2 only), but *flow-sensitive alias analysis* is even more computationally expensive than flow-insensitive analysis.

Method `m3` exemplifies the need for further extending the flow sensitive analysis, to handle the invocation context of the method calls. Method `m3` may call `nest.n()` indirectly, as a result of calling method `m1` at line 15. However, while the direct call of `m1` can propagate all the definitions D_0, D_1 and D_2, the call of `m1` in this context propagates only $D0$. In method `m3`, the condition at line 14 restricts the type of `nest` at line 15 to `NestB`, and thus only the definition D_0 can propagate to the exit of method `m3`. Thus, the already computational expensive *flow-sensitive* analysis must be made invocation *context-sensitive*, at the price of further increased complexity [Ryd03].

In general, *program paths and aliases through methods depend on the context in which they are invoked*. The more complex the inter-procedural data flow of an application is, the more there is a need for context sensitive alias analysis for detecting sound information and avoid infeasibility. However, context sensitive analyses struggle on applications with a complex inter-procedural control flow, for their computational cost.

Data flow approaches must trade-off between precision and affordability on several design decisions related to data flow analysis, alias analysis and their combination. Many approaches often end up with embracing a mix of (different) under and over-approximations that make unclear the degree of approximation of the final technique. For example, most of the implementations of data flow analysis used for testing do not include *any* alias analysis, introducing strong approximations to increase the applicability of the technique.

Chapter 3

Data Flow Testing

Data flow testing refers to a body of techniques that use data flow abstractions as test objectives, following the idea that the effectiveness (i.e., the probability of revealing faults) of a test suite is related to its ability to exercise the data relationships of the program under test. In recent times, data flow testing has attracted new interest in the context of testing object-oriented systems, since data flow information is well suited for capturing relations among the object states, and can provide useful information for selecting interesting method interactions to test.

Data flow testing is a form of testing that identifies data flow abstractions as test objectives to both evaluate the quality of test suites and guide the generation of test cases. In this chapter, we describe the main data flow testing approaches and discuss their effectiveness and limits.

Section 3.1 introduces data flow testing concepts and criteria as originally defined for testing procedural programs. Section 3.2 focuses on the application of data flow techniques for testing object-oriented systems, where data flow testing has been proposed to effectively identify combinations of methods that elicit the state based behavior of classes. Section 3.3 discusses the problems and limitations of data flow testing techniques, limitations that we address in this thesis.

3.1 Data Flow Criteria

Data flow testing techniques identify data relations as test objectives, and set coverage requirements on definitions, uses and definition use pairs identified by means of data flow analysis.

Data flow testing builds on the idea that if executing a line of code produces a wrong value, to effectively reveal the fault we need to execute not only the faulty line of code, but also a line of code that uses the erroneous value in some computation that will

Listing 3.1. Division by Zero Example

```
1  if(a==1){
2    c = 0; //bug
3  }
4  ...
5  if(b==1){
6    a = b/c;
7  }
```

lead to an observable failure. Consider for example the code snippet in Listing 3.1, and assume that the assignment `c = 0` at line 2 is faulty, since it causes a division by zero at line 6. To reveal the fault, we need not only to execute the statement at line 2, but also to pair it with the use at line 6, which causes an exception.

Building on this, researchers have proposed several data flow adequacy criteria, pairing definitions and uses in different ways, and using more or less thorough coverage requirements.

In the following we discuss the main data flow criteria and techniques described in literature, and we show how they relate to each other and to other structural coverage criteria.

Criteria

Data flow testing criteria consider data dependencies between definitions and uses in various ways. A common data flow testing criterion requires a test suite to exercise every definition use pair identified over an application under test in at least one program execution [Her76, LK83]. This criterion is known as the *all-pairs* or *reach coverage* criterion, and follows the original intuition that an erroneous value computed in a statement can be revealed only when used in another statement.

The all-pairs criterion can be extended to enforce the coverage of branches and statements as well, since satisfying definition-use pairs does not guarantee the execution of all the control elements of the application, and statement coverage is usually considered a minimum requirement while testing an application.

The *all-uses* criterion, proposed by Rapps and Weyuker, combines data flow and branch coverage, by requiring a test suite to execute at least one kill-free subpath from each definition to each use reached by the definition *and each direct successor statement of the use* [RW85, CPRZ89]. By requiring the execution of all the successor statements of the uses in predicates, the all-uses criterion forces all branches of the application to be taken. We can also say that the all-uses criterion *subsumes* both the all-pairs and the branch criterion, that is, a test suite that satisfies the all-uses always satisfies both all-pairs and branch as well.

A more thorough data flow criterion, the *all-du-path* criterion, requires the exe-

cution of every path between every definition and use of the same variable [RW85, CPRZ89]. By covering all definition use pairs, the all-pairs criterion requires to execute at least a kill-free path between the definition and the use, but there can be several paths between the definition and the use which elicit different behaviours of the application. The *all-du-path* criterion requires to execute all kill-free paths between each definition use pair, and thus subsumes the all-pairs criterion, but can be difficult to apply, as the number of paths between definitions and uses can be exponential in the size of the program unit — in particular, we are not aware of any work that applied the all-du-path criterion on industrial-size software.

All the criteria discussed above have a relatively high cost, for they identify numerous test requirements, and require complex tests to satisfy them. A simpler, but less powerful criterion is the *all-defs* adequacy criterion, which requires a test suite to execute at least one use for each definition in the program under test [RW85, CPRZ89].

Effectiveness

Despite the amount of work on data flow criteria [RW85, CPRZ89, Nta84, SFP+15], experimental results on their effectiveness are still limited and inconclusive. Some studies indicate that data flow information is useful for exercising corner cases and that test suites with high data flow coverage performs better than test suite obtained targeting structural criteria [FW93, SJYH09, VMGF13]. However, the work presented so far does not provide enough evidence of the existence of a causal relation between the execution of data flow abstractions and test suites effectiveness. Test suites that satisfy data flow criteria are generally bigger in size than suites for structural criteria, and thus the increase of effectiveness may depends simply on the number of test cases and not on the effectiveness of the criterion.

The few studies that experimented on suites with a fixed size do not support the thesis arguing that satisfying data flow relations is more effective than satisfying branches or statements. Frankl and Weiss studied the effectiveness of the all-uses criteria [FW93]; they investigated whether all-uses adequate test suites are more likely to expose bugs than branch coverage adequate ones. They compared the fault detection ratio obtained by test suites with different all-uses and branch coverage values of different size, on each and every 9 small-size Pascal program seeded with errors. They found that test suites with high all-uses coverage are more effective than test suites with high branch coverage, but also bigger in size. This is an intuitive result, since all-uses subsumes branch coverage, but their data do not support the thesis that the probability of spotting a failure increases as the percentage of def-use pairs or branch covered increases, nor give conclusive results in the case of equal-size test suites.

Hassan and Andrews reached similar results [HA13]. They compared the effectiveness of different coverage metrics on fifteen programs of different languages (C, C++ and Java) with seeded faults. In their experiment, they controlled the size of the suites, generating different suites of size between 3 and 50 tests. They report that when test

suite size is taken into account all-uses coverage performs similarly to branch coverage. They do not investigate the effectiveness of suites of different size obtained to maximise the different criteria though.

Hutchins et al. investigated the effectiveness of all-pairs with respect to branch coverage and random testing [HFGO94]. The main difference between this work and Frankl and Weiss' work is that the all-pairs criterion does not subsume branch coverage. They report that both all-pairs and branch test suites are more effective than suites selected randomly, but neither all-pairs nor branch coverage can be considered better than the other criterion, since they frequently detect different faults and thus are complementary in their effectiveness.

Santelices et al. [SJYH09] studied the effectiveness of coverage metrics when used in fault-localization techniques, which are techniques used in debugging to assist testers in finding the faults that caused an execution to produce incorrect outputs. Their study shows that neither structural nor data flow coverage perform best for all faults, but that different kinds of faults are best localized by different coverage criteria.

Mathur and Wong, Offut et al. and Frankl et al. compared data flow testing with mutation testing [MW94, OPTZ96, FWH97], a testing technique that mutate program by seeding faults, and selects test cases that distinguish the mutated from the original program. These studies suggest that mutation testing in practice often subsumes all-uses coverage, and that mutation testing performs slightly better than all-uses. For example, Offutt et al. observed an increase of effectiveness of 16%. On the other hand, test suites generated for mutation testing are more costly to obtain and execute: They are much bigger in size, requiring on the average from 200% to 350% more tests than data flow suites, and the seeding of faults necessary to use the technique is particularly expensive in terms of execution overhead.

It is important to note that all the experiments but Hassan and Andrews' one were conducted on limited sets of small procedural programs, thus is not clear whether the results could be generalised to industrial size software.

3.2 Data Flow Testing of Object-Oriented Software

Data flow testing has attracted new research interest in the context of testing object-oriented systems. Testing object-oriented systems differs from testing procedural programs, due to the different structure and state based behaviour of object-oriented programs. To test a class is not sufficient to test its single methods in isolation as we test procedures, but we have to invoke sequences of different methods (of one or different classes) that set the object state and that verify the behavior of the object while it is in that particular state. This is because a given method can produce different results depending on the state of the object when the method is invoked.

Standard structural testing approaches based on branches and statements give little guidance for selecting a reasonable combinations of methods and objects to test, be-

Listing 3.2. Code excerpt from Apache Commons Lang

```
1  public class MutableFloat {
2    private float value;
3    public MutableFloat(float value) {
4      this.value = value;
5    }
6    public void add(Number operand) {
7      this.value += operand.floatValue();
8    }
9    public boolean isNaN() {
10     return Float.isNaN(this.value);
11   }
12   public boolean isInfinite() {
13     return Float.isInfinite(this.value);
14   }
15   ...
```

cause they focus on the control flow of single methods ignoring the state based nature of objects. On the contrary, data flow information can capture the state based dependencies between methods that interact on the same state variables, and thus can be used to identify relevant state based interactions to test. Below we discuss how data flow testing has been extended for testing methods and classes.

3.2.1 Intra-Class Data Flow Testing

Data flow testing identifies relevant combinations of methods that interact on the same variables, and which properly exercise the intended behavior of a class under test. Consider for example the program reported in Listing 3.2 taken from a popular Java open source project. The class `MutableFloat` encapsulates a float `value`, and provides some methods to manipulate it. The methods `MutableFloat.add()` and `Mutable.isInfinite()` interact through the variable `value`. The variable `value` is defined (assigned) in method `MutableFloat.add()` and used (read) in method `Mutable.isInfinite()`, thus forming a definition use pairs between the two methods on `value`. A typical data-flow testing criterion would require a test case that calls the method `MutableFloat.add()` and eventually calls `Mutable.isInfinite()` in such a way that the result of the latter method is computed based on the value assigned by the former method, while common control flow criteria that targets statements and branches require only the individual elements to be executed, but not necessarily their combination. This test requirement increases the chances that the effects of a possible faulty value assigned to variable `value` in method `MutableFloat.add()` propagates to the use in method `Mutable.isInfinite()` and manifests as a failure, and better exercises the intended behavior of the class.

Harrold and Rothermel where the first to use data flow abstractions as test objectives

for testing classes [HR94]. They proposed a framework to compute definition use pairs over object-oriented systems and to adapt existing criteria (with a particular attention to all uses criterion). They distinguish between intra-method, inter-method and intra-class definition use pairs as discussed in Section 2.2.1, emphasizing the importance of using intra-class definition use pairs to identify the relevant sequences of methods to execute.

Buy et al. combine data-flow analysis, symbolic execution and automated reasoning to generate test cases for classes [BOP00]. They exploit symbolic execution to obtain the method pre-conditions that must be satisfied in order to traverse a feasible definition-clear path for each definition use pair, and use automated deduction to determine the order of method invocations that satisfy the preconditions of interest.

Other techniques employs meta-heuristic search algorithms to generate test cases that exercise definition use pairs. Liaskos and Roper [LRW07], Ghiduk et al. [GHG07], Miraz [Mir10] and Vivanti et al. [VMGF13] defined approaches that support the automated generation of tests cases for meeting data flow criteria for testing classes. However, these approaches implement many approximations in the data flow analysis for increasing the scalability of the techniques, of the type we discussed in Section 2.3, potentially reducing the effectiveness of the approaches.

3.2.2 Inter-Class Data Flow Testing

Inter-class data flow testing techniques extend object-oriented data flow testing approaches to identify combination of methods that interact on the nested state of classes. As discussed in Section 2.2.2, object-oriented programs define class states as sets of class fields that may be of primitive or structured types. In the latter case, the fields refer to objects that in turn include other fields, which define nested state dependencies between methods. Inter-class data flow testing techniques use inter-procedural data flow abstractions, such as contextual definition use pairs of class state variables, as test objectives to select test cases that exercise the complex nested state based behavior of classes.

Consider for example the classes `Robot` and `Armor` in Listing 3.3. The class `Robot` contains the class `Armor` as part of its state (field `Robot.armor` at line 2), and the behavior of the robot depends on the internal state of its armor (that is on the value of the class state variables `Robot.armor.damage` and `Robot.armor.value`). When the robot moves, through the invocation of method `Robot.move()`, its speed depends on the value of the class state variable `Robot.armor.damage`. A robot moves slowly if its armor is damaged, faster if not (method `Robot.computeSpeed() at line 13`, which invokes the method `Armor.isDamaged()` at line 14). The value of `Robot.armor.damage` depends on the armor status, which is modified by multiple invocations of the method `Robot.collides()` at line 5.

Therefore there is a nested state based interaction between the methods `Robot.collides()` and `Robot.move()`: the method `Robot.collides()` modifies the internal

Listing 3.3. Class `Robot`

```
1  public class Robot {
2    private Armor armor;
3    private double weight;
4
5    public void collides(Robot r){
6     armor.inflictDamage(weight*r.speed());
7    }
8
9    public void move(double dir) {
10      setPosition(computeSpeed(), dir);
11   }
12
13   public double computeSpeed(){
14    if(armor.isDamaged())
15      return 0.5*weight;
16    else return 1.2 * weight;
17   }
18 }
19
20
21 class Armor{
22   double value;
23   boolean damaged;
24
25   public void inflictDamage(double mass){
26     value = value - 0.3*mass;
27     if(value < 1)
28       damaged = true;
29   }
30
31   public boolean isDamaged(){
32     return damaged;
33   }
34
35 }
```

state of the robot armor, which changes the behavior of the method `Robot.move()`. This interaction is captured by the contextual definition use pair of the class state variable `Robot.armor.damaged` reported in Figure 3.1 that can be exercised only by a test case that invokes the method `Robot.collides()` multiple times until the the amor is damaged, and then the `Robot.move()`. Control-flow criteria and intra class data flow testing techniques do not identify any element that characterize this complex interactions, being limited to control structures and intra-class interactions, respectively.

Figure 3.1. A contextual definition use pair of variable `Robot.armor.damaged`

	Context	Point
Definition	Robot::collides(Robot) [6] → Armor::inflictDamage(double)	28
Use	Robot::move(double) [10] → Robot::computeSpeed() [14] → Armor::isDamaged()	32

Matena et al. proposed and experimented inter-class testing based on inter procedural data flow analysis, thus extending the Buy et al.'s work [MOP02]. While Buy et al. generate test sequences targeting intra-procedural definition use pairs, Martena et al. use the same combination of techniques targeting inter-procedural definition use pairs.

Souter and Pollock, and Denaro et al. [SP03, DGP08] have formalized the core object-oriented data flow testing concepts. Souter and Pollock noticed that to properly capture the inter procedural control flow relations between methods and classes, it is necessary to distinguish definitions and uses according to their invocation context, formalised the concept of contextual definition-use pair reported in Section 2.2.2, and proposed to use contextual definition use pairs to define thorough data flow testing techniques for object-oriented systems.

Denaro et al. extended Harrold and Rothermel's framework with the concepts introduced by Souter and Pollock. They formalised the notion of contextual intra method, inter method and intra class definition use pairs and testing, emphasising that contextual definition use pairs computed over subsystems can identify interesting state based relations between a class and the classes that compose its state. Denaro et al. also propose an efficient algorithm to compute contextual definition use pairs, which they implemented in DaTeC, a protoype tool for computing contextual data flow coverage [DGP09].

Souter and Pollock and Denaro et al. have focused on the formalization of contextual data flow analysis and on contextual data flow testing criteria, respectively, leaving open the problem of automatically generating test cases that exercise feasible contextual definition use pairs.

In this thesis we share the vision of Souter and Pollock and of Denaro et al. that contextual definition use pairs are well suited to select relevant method sequences to test, and introduce the *Dynamic Data Flow Testing* technique that uses dynamically identified contextual definition use pairs to automatically generate relevant test cases.

3.3 Open Challenges of Data Flow Testing

Despite the enormous amount of work on data flow testing, there are still unresolved problems that derive from the imprecision of static data flow analysis and the complexity of efficiently generating test suites that achieve a stable data flow coverage.

Classic data flow testing relies on static data flow analysis to compute the set of definition use pairs to use as test objectives. Data flow analysis can be imprecise when analyzing programs with a complex structure, and may both identify data flow elements that result infeasible and miss data flow elements that depend on the dynamic behavior of the application. The presence of infeasible elements complicates the identification of the right set of test objectives. Missing feasible data relations may reduce the efficacy of data flow test suites.

The large amount and high complexity of the test objectives identified with data flow testing criteria can be hardly managed without proper automated tools. Automating data flow testing is therefore necessary, but as the present time there is a lack of techniques and tools for both computing data flow coverage and automatically generating test cases that satisfy data flow abstractions.

Below we discuss more in detail these two problems, focusing on their impact on the effectiveness and applicability of data flow testing, and on the proposed solutions.

3.3.1 Handling the Imprecision of Data Flow Analysis

The static nature of data flow analysis may under or over-approximate the results, thus leading to a discrepancy between the data flow abstractions identified by the data flow

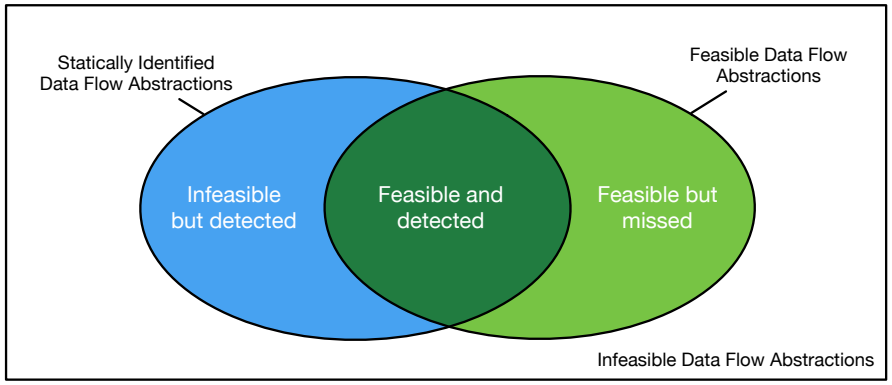


Figure 3.2. Relation between data flow abstractions statically identified with data flow analysis and feasible data flow abstractions.

analysis, and the data flow abstractions that are executable in the program under test. Figure 3.2 visualises the relation between statically identified and feasible data flow abstractions: classic static data flow analysis may identify both a set of feasible data flow abstractions (*Feasible and detected*) and a set of data flow relations that are not executable (*Infeasible but detected*), and may miss a set of feasible data flow elements (*Feasible but missed*). The size of the *Infeasible but detected* and the *Feasible but missed* sets determines the quality of data flow testing.

The presence of *Infeasible but detected* elements complicates the generation of test cases that target data flow test objectives, and reduces the relevance of data flow coverage metrics. In the presence of infeasible test objectives, both testers and automated tools may waste effort in trying to satisfy test objectives that are impossible to execute, diverting the testing activity. Testers may also find difficult to interpret low coverage measures, because of being unable to determine if low values depend on missed feasible objectives or infeasible ones.

The presence of *feasible but missed* data flow elements impact negatively on the ability of data flow testing of revealing faults. The rationale of data flow testing is that to detect faults it is necessary to stress *all* the data dependencies of the application, but when a data flow analysis misses many feasible elements, the data flow testing technique works on a subset of test objectives that badly approximate the set of executable ones, and consequently can leave untested important data relations of the application. To maximise fault finding, it is necessary for a data flow testing technique to consider the majority of the executable definition use pairs, to guarantee an appropriate coverage of the application execution space.

The size of the *infeasible but detected* and the *feasible but missed* sets depends on both the complexity of the analyzed application and the accuracy of the implemented data flow analysis.

Since the imprecision of data flow analysis stems mostly from the dynamic features

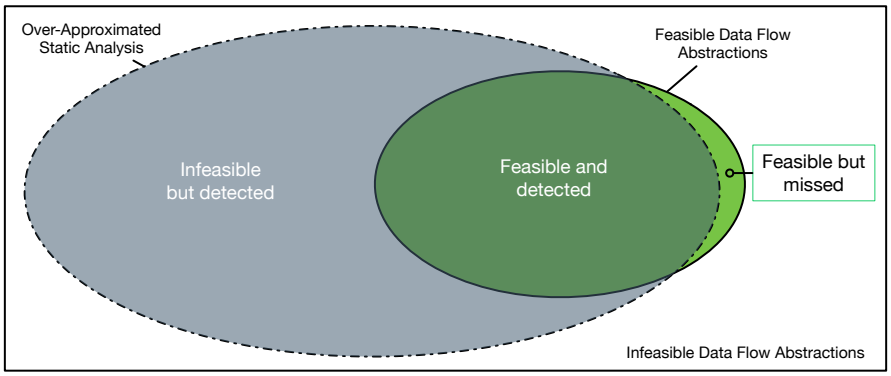


Figure 3.3. Interplay between feasible and infeasible data flow elements in the case of strongly over-approximated analysis

of programs, when targeting simple programs that do not use extensively dynamic features such as pointer aliases and dynamic binding, a standard data flow analysis is both simple to implement and apply, and precise in computing the right set of test objectives.

However, when targeting complex programs that heavily rely on dynamic constructs, it becomes difficult to design a static analysis with a good tradeoff between precision and scalability. Data flow analysis abstracts the dynamic behaviors of the program, which cannot be modelled precisely statically, and produces decidable either over or under-approximations. Over-approximating the dynamic behavior of the program improves the completeness of the analysis (reduces the amount of *feasible and detected* referring to Figure 3.2) at the cost of increasing the amount of *infeasible but detected* elements. Under-approximating the dynamic behavior of the program limits the amount of infeasible elements at the cost of increasing the amount of *feasible but missed* elements.

Precise implementation of data flow analysis are complemented with some flow and context-sensitive alias analyses to capture pointer aliases and dynamic binding, but they hardly scale to large applications. Implementations that lay down precision for scalability to apply data flow analysis to large and complex systems, rely either on strong over-approximations of the dynamic behavior of the application, such as modelling aliases and dynamic binding using a flow-insensitive alias analysis, or on strong under-approximations of the dynamic behavior, for instance considering only the binding of procedures declared statically in the program.

Strong over and under-approximations have a big impact on the precision of the results. Implementations based on strong over-approximations potentially lead to the identification of the majority of feasible elements, but they also identify a number of *infeasible but detected* elements that can be exponential the number of feasible ones (Figure 3.3). The high number of infeasible test objectives strongly reduces the applicability of the technique.

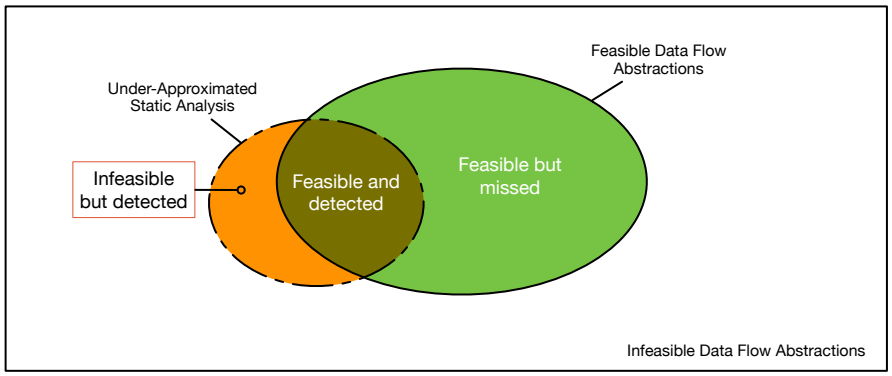


Figure 3.4. Interplay between feasible and infeasible data flow elements in the case of strongly under-approximated analysis

Most current data-flow testing prototype tools rely on data flow analyses based on strong under-approximations of the dynamic features of programs, that simplify the implementation, by reducing the number of identified definition use pairs (feasible and infeasible) at the cost of increasing the number of *feasible but missed* elements. These discrepancies between the data flow abstractions identified by a strongly under-approximated data flow analysis and the data flow abstractions that are executable in the program under test are pictured in Figure 3.4.

Most of techniques surveyed in this section focus on data flow abstractions computed within single procedures or methods, and exclude the inter procedural control flow, the dynamic binding of procedures, the representation of arrays and pointer aliases to a certain extent [FW93, HFGO94, FWH97, HA13]. Also the most advanced approaches for testing object-oriented systems do not take in account aliases and polymorphism, and under-approximate dynamic bindings, in their implementation [MOP02, DGP08, VMGF13].

Most of the work surveyed in this section does not evaluate the impact of the tolerated under-approximation on the effectiveness of the approaches. The few extensive studies on the infeasibility problem of data flow analysis discuss the problem but do not deeply investigate the impact of *feasible but missed* elements on data flow testing [Wey90, FW93, DGP08].

This thesis starts from the hypothesis that data flow analyses that strongly under-approximate the dynamic behavior of the application can miss many data flow abstractions relevant to testing, thus compromising the effectiveness of the technique. We hypothesise that this problem is pervasive in current data flow testing techniques, especially in the context of object-oriented systems that heavily rely on dynamic features. We further hypothesise that neither strong under nor over-approximation produce analysis suitable for computing test objectives for programs with a complex dynamic behavior, because they either miss too many feasible elements, or identify too many infeasible

ones, and thus we need a more precise approach to capture the right objectives for data flow testing.

We further discuss the unsuitability of strongly approximated data flow analyses for data flow testing, through the example reported in Listing 3.4 that highlight the impact of over and under-approximated data flow analyses on the amount of *feasible but missed* and *infeasible but detected* elements. In the example, we discuss the testing of a simple object-oriented application using two different analyses. One analysis strongly over-approximates dynamic biding by resolving every possible bind relying on the class hierarchy of the program, and the other strongly under-approximates it by resolving only static binding of procedures.

The program shown in Listing 3.4 comprises six Java classes: the class-under-test `Cut`, the class `Level` that is specialized in `L1`, `L2`, `L3`, and a factory class `LevelFactory`.

`Cut` implements the methods `met1()` and `met2()`, which modify and use the class state variables `Cut.v0` and `Cut.lev`. The variable `Cut.v0` (line 2) is of type boolean, while `Cut.lev` (line 3) is statically declared as of type `Level`, but at runtime is always of concrete type `L1`.

Class `L1` (line 22), which extends the abstract class `Level`, includes the state variable `L1.sub1` (line 24) of static type `Level`, that is always of concrete type `L2` (line 38). Therefore, at runtime the classes `L1` and `L2` compose the (nested) state of the class `Cut`. The class `L3` (line 49) extends `Level` as well, but is unrelated with the class under test `Cut`.

In classes `Cut`, `L1` and `L3` the state variables of type Level are instantiated using the class `LevelFactory` (line 63). `LevelFactory` implements the common design pattern factory: It returns new instances of the type `Level` by instantiating some of the subtypes according to a selector value passed as parameter.

Table 3.1 reports the feasible contextual definition use pairs of class `Cut` that shall be used as test objective and that we computed manually. Each row lists a definition use pair composed of an identifier (column *Pair*), the variable name (column *Variable*), the line of code that corresponds to the variable definition within the call chain that may lead to the definition (column *Def*) and the line of code that corresponds to the use within the corresponding call chain (column *Use*). The variable name indicates also the associations between class fields and objects according to their alias relations. For example the pair $p3$ refers to the variable `Cut.lev.v1` because the field `Cut.lev` is alias of an object of type `L1`.

These feasible pairs capture the state based interactions between the class `Cut` and the classes `L1` and `L2` that comprise its state. In particular, the pair p4 capture the non-trivial combination of method calls elicited by `test3()`, which leads to the failure of the assertion at line 11. Listings 3.5 reports a test suite composed of the test cases `test1()` and `test3()` that executes all the feasible pairs listed in Table 3.1.

Listing 3.4. Class `Cut` Example

```
1  class Cut {
2      boolean v0 = false;
3      Level lev = LevelFactory.makeLevel(1);
4      void met1(){
5          lev.doA();
6          v0 = true;
7      }
8      boolean met2(){
9          boolean ok = true;
10         if(v0) ok = lev.doB();
11         assert ok;
12         //Bug: this assertion can fail !
13         return ok;
14     }
15 }
16
17 abstract class Level{
18     abstract void doA();
19     abstract boolean doB();
20 }
21
22 class L1 extends Level{
23     boolean v1 = false;
24     Level sub1 = LevelFactory.makeLevel(2);
25     void doA() {
26         if(v1) sub1.doA();
27     }
28     boolean doB() {
29         v1 = true;
30         boolean ret = sub1.doB();
31         return ret;
32     }
33 }
34
35
36
37
38 class L2 extends Level{
39     boolean v2 = false;
40     void doA() {
41         v2 = true;
42     }
43     boolean doB() {
44         if(v2) return false;
45         return true;
46     }
47 }
48
49 class L3 extends Level{
50     boolean v3 = false;
51     Level sub3 = LevelFactory.makeLevel(4);
52     void doA() {
53         if(v3) sub3.doA();
54         v3 = sub3.doB();
55     }
56     boolean doB() {
57         if(v3) sub3.doA();
58         v3 = sub3.doB();
59         return true;
60     }
61 }
62
63 class LevelFactory{
64     static Level makeLevel(int selector){
65         switch(selector){
66             case 1: return new L1();
67             case 2: return new L2();
68             case 3: return new L3();
69                 //case 4, case 5, ...
70             default: return null;
71         }
72     }
73 }
```

Listing 3.5. Test Cases for Class `Cut`, reported in Listing 3.4

```
1  //executes pair p1
2  @Test
3  public void test1() {
4      Cut cut = new Cut();
5      cut.met2();
6  }
7
8
9
10
11 //executes pair p2
12 @Test
13 public void test2() {
14     Cut cut = new Cut();
15     cut.met1();
16     cut.met2();
17 }
18
19
20
21 //executes pairs p2 p3 p4
22 @Test
23 public void test3() {
24     Cut cut = new Cut();
25     cut.met1();
26     cut.met2();
27     cut.met1();
28     cut.met2();
29 }
```

Table 3.1. Feasible definition use pairs of class `Cut`

Pair	Variable	Def at	Through call chain	Use at	Through call chain
p1	Cut.v0	2	Cut.<init>	10	Cut.met2
p2	Cut.v0	6	Cut.met1	10	Cut.met2
p3	Cut.lev[L1].v1	29	Cut.met2→L1.doB	26	Cut.met1→L1.doA
p4	Cut.lev[L1].sub1[L2].v2	41	Cut.met1→L1.doA→L2.doA	44	Cut.met2→L1.doB→L2.doB

Table 3.2 reports a subset of the definition use pairs identified with a data flow analysis that strongly over approximate the possible dynamic behavior of the application (hereafter *OADF*, over-approximated data flow analysis), which integrates a may-alias static analysis to resolve alias relations [OW91, LH03]. OADF computes a conservative over approximation of the set of possible definition use pairs, with awareness of polymorphism and dynamic binding, relying on the static type hierarchy to resolve polymorphic types.

We refer to the set of objects that the over-approximated may-alias analysis infers, and that may bind to the variables used throughout the program, as the *may-alias set* of a variable at a program point. For instance, the may-alias set of variable `Cut.lev` used at the call-point `lev.doA()` (line 5 in Listing 3.4) consists of the objects instantiated within the method `LevelFactory.makeLevel()` at lines 66, 67 and 68. The OADF may-alias set binds the variable `Cut.lev` to the objects of type `L1`, `L2` and `L3`, ignoring the fact that the call to `LevelFactory.makeLevel()` at line 3 always returns an object of type `L1`, and thus identifies of many infeasible pairs.

Only the first four pairs, p1, p2, p3 and p4, that appears in both Tables 3.2 and 3.1 are feasible, all the other pairs that appear in Table 3.2 and many more that we do not report in the table for the sake of brevity are *infeasible but detected* pairs. They derive from alias relations that are inferred statically by the conservative may alias analysis and that do not occur at runtime. In this example, OADF captures all the important useful data flow information, but also a lot of false positives that cause the divergence of the testing effort, complicating data flow testing.

Under-approximate data flow analysis (hereafter *UADF*) binds variables to objects by relying on the static declarations of types in the source code, without any mechanism to resolve reference aliasing, with lower computational costs and better scalability than OADF.

UADF pays the lower cost and better scalability than OADF with lower precision that derives from less insights on the behavior of a class than OADF. For instance, in this example of Listing 3.4, UADF identifies only the pairs p1 and p2 of Table 3.1, and does not identify the pairs related to the calls of methods `sub.doA()` (line 5) and `sub.doB()` (line 10), because the static binding of these methods through the variable `sub` of class `Level` does not link to any concrete implementation of the methods. Thus, the only

Table 3.2. Definition use pairs computed with OADF for the methods of class `Cut`, reported in Listing 3.4

Pair	Variable [may-alias object type]	Def at	Through call chain	Use at	Through call chain
p1	Cut.v0	2	Cut.<init>	10	Cut.met2
p2	Cut.v0	6	Cut.met1	10	Cut.met2
p3	Cut.lev[L1].v1	29	Cut.met2→L1.doB	26	Cut.met1→L1.doA
p4	Cut.lev[L1].sub1[L2].v2	41	Cut.met1→L1.doA→L2.doA	44	Cut.met2→L1.doB→L2.doB
p5	Cut.lev[L1].sub1[L3].v3	54	Cut.met1→L1.doA→L3.doA	53	Cut.met1→L1.doA→L3.doA
p6	Cut.lev[L1].sub1[L3].v3	54	Cut.met1→L1.doA→L3.doA	57	Cut.met2→L1.doB→L3.doB
p7	Cut.lev[L1].sub1[L3].v3	58	Cut.met2→L1.doB→L3.doB	53	Cut.met1→L1.doA→L3.doA
p8	Cut.lev[L1].sub1[L3].v3	58	Cut.met2→L1.doB→L3.doB	57	Cut.met2→L1.doB→L3.doB
p9	Cut.lev[L1].sub1[L3].sub3[L2].v2	41	Cut.met1→L1.doA→L3.doA →L2.doA	44	Cut.met2→L1.doB→L3.doB →L2.doB
p10	Cut.lev[L2].v2	41	Cut.met1→L2.doA	44	Cut.met2→L2.doB
p11	Cut.lev[L3].v3	54	Cut.met1→L3.doA	53	Cut.met1→L3.doA
p12	Cut.lev[L3].v3	54	Cut.met1→L3.doA	57	Cut.met2→L3.doB
p13	Cut.lev[L3].v3	58	Cut.met2→L3.doB	53	Cut.met1→L3.doA
p14	Cut.lev[L3].v3	58	Cut.met2→L3.doB	57	Cut.met2→L3.doB
p15	Cut.lev[L3].sub3[L1].v1	29	Cut.met2→L3.doB→L1.doB	26	Cut.met1→L3.doA→L1.doA
p16	Cut.lev[L3].sub3[L1].sub1[L2].v2	41	Cut.met1→L3.doA→L1.doA →L2.doA	44	Cut.met2→L3.doB→L1.doB →L2.doB
p17	Cut.lev[L3].sub3[L2].v2	41	Cut.met1→L3.doA→L2.doA	44	Cut.met2→L3.doB→L2.doB
...	...	...		...	

information captured with UADF is the set-get behavior between method `met1()` and `met2()` that define and read the variable `Cut.v0`. These pairs are exerecised with a test suite composed of the test cases `test0()` and `test1()` of Listings 3.5, which does not exercise the state based interactions between `Cut`, `L1` and `L2` that are exercised by `test3()` that would expose the fault. In this example UADF does not identify any infeasible pair, but misses the critical pairs that would expose the failure.

The example witnesses the impact that data flow analysis have on the applicability and effectiveness of data flow testing. Analyses that strongly over-approximate the application behavior may hide useful information in a lot of false positives, undermining the usefulness of the approach, while analyses based on data flow information computed without alias information may miss important test objectives, reducing the effectiveness of data flow testing.

3.3.2 Automating Data Flow Testing

As in the case of all structural testing techniques, data flow testing cannot be applied without a reasonable automation support. Both computing data flow coverage and generating test cases with data flow objectives is intuitively more difficult that comput-

ing targeting simple structural objectives, for the target involves pairs of and not just single program locations. The large amount of data flow objectives further increased the cost of automating data flow testing, since the number of definition use pairs can be exponential in the size of the program, in particular when distinguishing definition use pairs using contextual information [SP03, DGP08].

Despite the needs of automation, the many problems of automating data flow testing hinder the production of tools, and most tools for structural testing target classic structural elements like statements, branches and conditions, and do not support data flow elements [YLW06].

Automating the computation of data flow coverage requires runtime tracking the active and killed definitions, and pairing them with the executed uses. This dynamic tracking have to be implemented either by instrumenting the original code or modifying the execution environment, and can generate a significant overhead in terms of execution time [SH07]. Despite the many proposed data flow coverage techniques, there are only few research tools for data flow testing, which support a limited set of programming languages and features, as opposed to classic structural coverage, like statement and branch coverage, that are often supported directly both by the compiler and commercial tools [YLW06].

The scarce availability of tools for computing data flow coverage complicates also the research on data flow test generation that requires the tracking data flow coverage. The early attempts to completely automate data flow testing, from coverage to test case generation, exploit either model checking or search based algorithms [Gir93, HCL$^+$03, LRW07, GHG07, Mir10]. These tools focused either on procedural programs or on sample classes, stripped out of their dynamic features.

The current data flow testing tools work with strongly under approximated data flow analyses that compute only a subset of the executable definition use pairs. There is no evidence on how automated test generators would perform in the presence of test objectives identified with more precise analyses that limit the number of missed but feasible elements.

Chapter 4

Dynamic Data Flow Analysis

This section describes Dynamic Data Flow Analysis (DReaDs), a new technique to perform a sound inter-procedural data flow analysis by means of dynamic analysis. Differently from static analyses that compute data flow abstractions over the source code, DReaDs works by monitoring the program execution, identifying data flow information on the observed execution traces. By analysing the memory while executing the program, DReaDs overcomes many limitations of static approaches in dealing with the dynamic behavior of software, reducing the number of both the infeasible but detected, and the feasible but missed data flow elements.

This section describes *Dynamic Data Flow Analysis* (*DReaDs*[1]), a dynamic software analysis technique for computing sound inter-procedural data flow information.

DReaDs works by means of dynamic analysis: it monitors the execution of an application intercepting memory write and read events, and maps these events to the corresponding data flow abstractions of class state variables. The information computed by *Dynamic Data Flow Analysis* is sound, and can complement, or substitute, the unsound data flow information obtained by state of practice static data flow analyses.

Section 4.1 introduces the idea behind *DReaDs*. Section 4.2 describes the workflow of *DReaDs* and its architecture that is composed of three main elements that (i) build and maintain a model of the objects in memory during the execution of the program (Section 4.3), (ii) exploit the memory model to map memory events to data flow abstraction during the execution (Section 4.4), and (iii) store the data flow information for each class merging the information computed on different instances and traces (Section 4.5). Section 4.6 reports on the design of a prototype implementation of *DReaDs* for Java programs.

[1] *DReaDs* stands for **D**ynamic **Rea**ching **D**efinitions Analysi**s**

4.1 Objectives

The dynamic characteristics of programming languages, such as arrays, aliases and dynamic binding, complicate the computation of sound data flow information. Current implementations of data flow analysis often approximate the dynamic features of programs, reducing the soundness and completeness of the results to ease the analysis and improve its scalability, as discussed in Sections 2.3 and 3.3.

In this chapter we define *DReaDs*, a *dynamic* inter-procedural data flow analysis technique that computes precise data flow information, reducing the amount of *infeasible but detected* and *feasible but missed* data flow elements that hinder current data flow testing approaches.

The distinctive characteristic of *DReaDs* is to analyze the program while executing it, using the information available at runtime about aliases and references in memory to compute *sound* data flow information. *DReaDs* exploits a model of the object references that are in memory to resolve reference aliases and compute data flow abstractions precisely.

With respect to static approaches, *DReaDs* computes only feasible data flow abstractions, because the computed information derives from information collected at runtime, and identifies the data flow abstractions that depends on the dynamic behavior of the application, which static analyses either miss or over-approximate. The completeness of the *DReaDs* data flow analysis depends on the thoroughness of the analyzed executions: the more an application is executed and analysed with *DReaDs*, the more complete is the identified data flow information.

Figure 4.1 visualizes the relation between the different data flow analysis techniques with respect to the feasibility of the computed data flow information. Over-approximated analyses exceed in the number of infeasible but detected elements, while under-approximated analyses miss many feasible elements that depends on dynamic constructs. *DReaDs* aims to compute a representative subset of the feasible elements, excluding infeasible ones and identifying elements that depends on the dynamic behavior of the application.

4.2 *DReaDs* Architecture

Although *Dynamic Data Flow Analysis* (*DReaDs*) could be generalised to work on general programs, we propose it for inter procedural analysis of object-oriented systems, that is, for the computation of data flow information of class state variables as defined in Section 2.2.2. In details, given an object-oriented program, *DReaDs* computes the propagation of contextual definitions of class state variables, the reachability of contextual uses of class state variables, and the set of (observed) contextual definition use pairs.

The distinctive characteristic of *DReaDs* is to work on the observed behavior of the

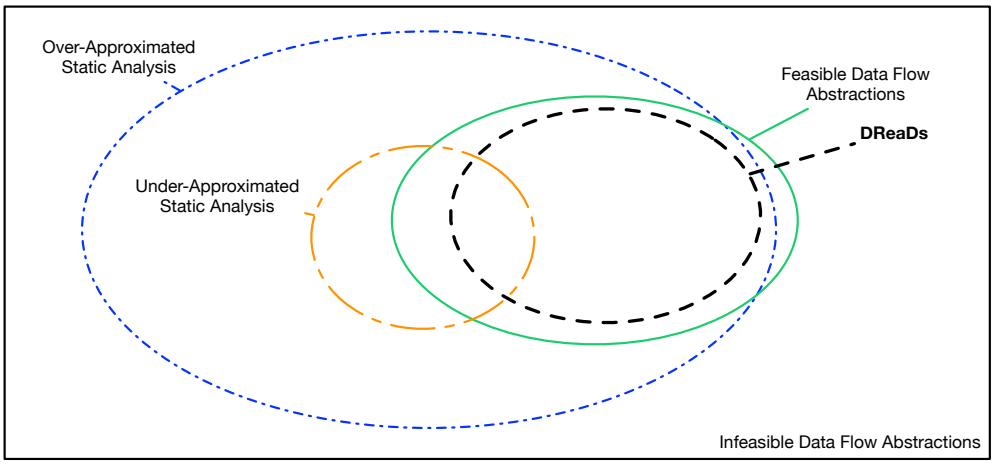


Figure 4.1. Comparison of different data flow analyses with respect to feasibility

program and not on its static structure. *DReaDs* analyzes the memory write and read events observed while executing the program, and computes definitions and uses of class state variables by identifying the objects states impacted by the memory events. *DReaDs* exploits a model of the references between objects present in memory, which it creates and updates while processing the observed memory events, to map memory accesses to data flow abstractions. The information collected on multiple objects and traces is merged to capture the general data flow information of the application.

DReaDs is composed of (1) a component that maintains a memory model to map both memory-load and memory-store events to the involved class state variables, (2) a component that exploits the memory model to monitor the propagation of definitions and the reachability of uses of the class state variables along the monitored execution traces, and (3) a component that merges the reaching definitions and uses computed across multiple instances and traces to produce a complete model of the dynamically identified data flow relations.

Algorithm 1 describes the workflow of *DReaDs*. *DReaDs* takes as input a program and a test suite, and executes all the test cases in the test suite (Algorithm 1, lines 2 and 3). For each test case, *DReaDs* executes the program (lines 6 and 7) until the execution of the test case terminates (line 11). For each execution step, *DReaDs*:

1. invokes the *maintain memory model* `maintainMModel` component (line 8) that updates the memory model to represent the objects instantiated in memory and the references between them in the current execution step. The `MModel` is initialised to empty at the beginning of an execution trace (line 4).

2. invokes the *identify data flow events* `identifyDFEvents` component (line 9) to identify on the memory model which objects where impacted by a definition or

Algorithm 1 DReaDs(Program, TestSuite)

```
Require: Program: The program under analysis
Require: TestSuite: A test suite for the program under analysis
 1: GeneralDFInfo = EMPTY
 2: while hasNextTestCase(TestSuite) do
 3:     TestCase = nextTestCase(TestSuite)
 4:     MModel = EMPTY
 5:     TraceDFInfo = EMPTY
 6:     repeat
 7:         State = executeStep(Program, TestCase)
 8:         MModel = maintainMModel(MModel, State)                    ▷ section 4.3
 9:         TraceDFInfo = identifyDFEvents(TraceDFInfo, MModel, State)  ▷ section 4.4
10:         GeneralDFInfo = mergeDFInfo(GeneralDFInfo, TraceDFInfo, State)  ▷ section 4.5
11:     until ¬atEndOfProgram(State, Program)
12: end while
13: reportReachingDefs(GeneralDFInfo, Program)                        ▷ section 4.5
```

use event, and updates the set of active definitions, observed uses and executed pairs accordingly (variable `TraceDFInfo`, line 9). *DReaDs* updates the active definitions following the kill-gen logic of classic data flow analysis.

3. invokes the *merge data flow information* `mergeDFInfo` component (line 10) to update the information on reaching definitions and reachable uses of the basic block of the class (variable `GeneralDFInfo`), using the information on active definitions and observed uses of the current execution trace stored in `TraceDFInfo`. The data flow information per basic block is initially empty (line 1), is incrementally updated by the merging mechanism during the execution of the different test cases, and is eventually exploited to report the computed information at the end of the analysis (line 13).

4.3 *DReaDs* Memory Model

DReaDs maintains a runtime model of the relations between the object instances in memory to identify the class state variables involved in assignments along an execution trace. The memory model is a directed graph, where the nodes represent the distinct object instances or primitive values in memory identified by their identity, and the edges represent references between instances identified by the name of the field that set the reference. For instance an edge from a node n_1 to a node n_2 with label l represents a field l in n_1 that refers to n_2.

Consider, for example, the classes in Listing 4.1, the test case in Listing 4.2 and the memory model reported in Figure 4.2. Listing 4.1 comprises three classes under test `Z`, `A` and `B`; Listing 4.2 reports a test case t that execute the classes, and Figure 4.2 shows

```
1  class Z{
2    private A a;
3    Z(A a){
4      this.a=a;
5    }
6    void doSmt(){
7      print(a.getB());
8    }
9  }
10
11
12 class A{
13   private B b;
14   void setB(B b){
15     this.b=b;
16   }
17   B getB(){
18     return b;
19   }
20 }
21 class B{
22 }
```

Listing 4.1. Classes Under Test

```
1  @Test
2  public void t() {
3    A a1 = new A();
4    A a2 = new A();
5    B b = new B();
6    a1.setB(b);
7    a2.setB(b);
8    Z z = new Z(a1);
9    z.doSmt();
10 }
```

Listing 4.2. Test *t* for classes `Z A B`

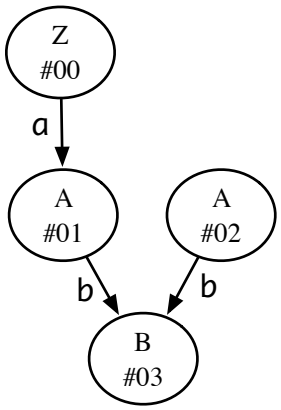


Figure 4.2. Memory Model obtained at the end of the execution of *t*

the memory model created by *DReaDs* when executing the test case *t*.

The memory model in the figure represents the state of the objects in memory after executing the test case *t*. The node *A*#01 represents the object of type `A` instantiated at line 3 of *t* (where #01 is its identity hashcode). The object is part of the state of *Z*#00 as captured by the edge *a*, reflecting the fact that the state of class `Z` includes the field `a` that refers to the object *A*#01. The memory model includes also an object of type *B* (*B*#03) and another object of type *A* (*A*#02) and the relative relations.

DReaDs builds and maintains the memory model incrementally, while monitoring the execution of the program under analysis. It initializes the model to an empty graph for each test case (Algorithm 1, line 4), and updates the model after each execution step (line 8) to correctly represent the status of the objects that exist in memory at that point of the execution. The updating mechanism adds nodes and either adds or removes edges, according to the memory related operations observed during the execution.

DReaDs adds a node to the model whenever it observes a memory reference related to an object instance that is not represented yet. To this end, it relies on a runtime monitoring framework that detects each referenced object, and augments the model with a new node for each object that is not already represented in the model. In this way, *DReaDs* lazily enforces the memory model to include a node for each object instances that has been accessed at least once at runtime.

DReaDs adds and removes edges to the model when observing assignments to in-

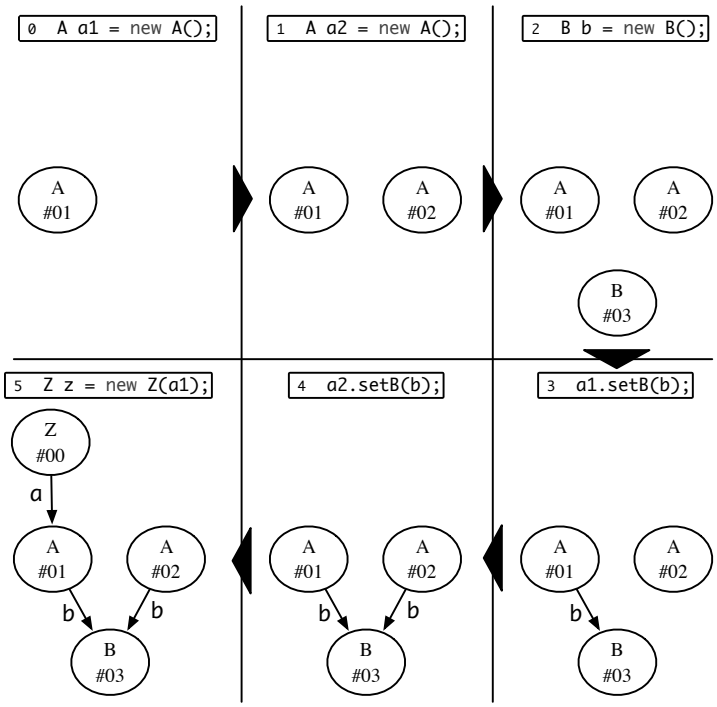


Figure 4.3. Incremental construction of the memory model for the example in Figure 4.2.

stance fields. If the model already contains an edge that represents the field, then *DReaDs* removes it. If the value assigned to the field is not `null`, then *DReaDs* augments the model with a new edge from the field owner instance to the node representing the assigned value. Every time an edge is removed, *DReaDs* checks whether the original target node is still referenced by another node or by the entry point of the program. If the node is not reachable anymore, *DReaDs* removes it from the graph, simulating a garbage collector. This algorithm produces a sound representation of the objects in memory, and contributes to reducing the size of the model.

Figure 4.3 illustrates how the model is incrementally built during the execution of the different statements of the test case t in Listing 4.2 with respect to the example discussed above.

DReaDs addresses array structures as special instances that include a field for each offset in the array. Each edge between the array and any array element is labeled with the position index of the element. Thus, the above representation and handling generalize to arrays as well.

4.4 *DReaDs* Runtime Analysis

The *Runtime Analysis* module monitors the program execution to identify dynamic data flow abstractions and their propagation over a single execution trace (Algorithm 1, line 9). The module (1) intercepts memory events at runtime and maps them to (dynamic) definitions and uses, and (2) keeps track of the set of active definitions and reachable uses for any instant of the execution.

Mapping memory events to definitions and uses of instance state variables

The *Runtime Analysis module* of *DReaDs* maps memory events to definitions and uses of instance state variables. The module specializes the concept of class state variable introduced in Definition 2.1 to capture the concrete states of the object instances at runtime, and adapts the concepts of contextual definition, use and pair accordingly. In particular, it identifies definitions and uses of *instance state variables*, that is, definitions and uses of class state variables specialized distinguishing the object instance they belong to.

Definition 4.1 (*Instance state variable*)**.** An instance state variable is a pair $\langle instance\_id, field\_chain \rangle$, where $instance\_id$ is the identity of an object instance and $field\_chain$ is a chain of field signatures that navigate the data structure of the instance up to an attribute declared therein.

Definition 4.2 (*Contextual definition (use) of instance state variable*)**.** A contextual definition (use) cd (cu) of instance state variable of an object o is a tuple $(inst, \langle im_1, im_2, ..., im_n \rangle, l)$ where $inst$ is the instance state variable of o, l is the location of the actual instruction that modifies (references) the nested state of o, and $\langle im_1, im_2, ..., im_n \rangle$ is an execution trace where im_1 is the call site of the first method call leading to a modification (read) of the state of o, and im_n is the method containing l.

Definition 4.3 (*Contextual definition use pairs of instance state variable*)**.** A contextual definition use association for an instance state variable $inst$ is a pair (cd, cu) that associates a contextual definition cd of $inst$, with a contextual use cu of $inst$, which occur on the same objects instances and have at least one def-clear paths between them.

Definitions and uses of instance state variables are identified by *DReaDs* over the memory model described in the previous section. To this end, this module of *DReaDs* monitors the memory events at runtime, and for each event uses the model to compute the instance state variables that were impacted by that event.

The lookup of the model works as following: When a memory event occurs on an object o, *DReaDs* identifies the node n in the memory model corresponding to o, according the the identity of o. Then, starting from this node, it traverses the graph in a depth-first fashion to identify all the nodes n_0, ..., n_k that are directly or indirectly connected to n, and that represent all the objects o_0, ..., o_k that own a reference to the

object o. Each path from a node n_i to n, for $i = 0..k$, identifies a state variable that corresponds to an object o_i that includes o as part of its state. Since the memory model can contain cycles, *DReaDs* excludes paths that traverses the same edge multiple times. For instance, in the memory model shown in Figure 4.2, the path from $Z\#00$ to $B\#03$ identifies the instance state variable $\langle Z\#00, \texttt{Z.a.b}\rangle$ that represents the inclusion of the object $B\#03$ in the state of the object $Z\#00$.

DReaDs maps definition and use memory events to definitions and uses of instance state variables, respectively. Definition events correspond to changes of the internal state of an object, triggered by the assignment of a field (memory store events that assign a primitive value or set a reference). Use events correspond to accesses to the value of state variables in program statements, that is, memory load events that read a primitive value or retrieve a reference. Moreover, *DReaDs* records the execution trace at runtime to identify the context of methods invocations of definitions and uses.

For example, in Listing 4.2 the test t at line 8 instantiates a new object `Z` (with the instruction `Zz=newZ(a1)`), consequently executing the constructor of `Z` (lines 3–5 of Listing 4.1). When executing the assignment `this.a=a` at line 4 in the constructor of `Z`, *DReaDs* creates the node $Z\#00$, and the edge a in the model in Figure 4.2, and identifies the assignment as a definition event. Thus, after updating the model, *DReaDs* navigates the graph, and identifies the occurrence of the direct definition of the instance state variable $\langle Z\#00, \texttt{Z.a}\rangle$, and of the nested definition of the state variable $\langle Z\#00, \texttt{Z.a.b}\rangle$, with context `Z#00.<init>`. In the same way, executing the instruction `z.doSmt()` at line 9 of t, triggers a use event on $B\#03$ and leads to the identification of the use of the class state variables $\langle A\#01, \texttt{A.b}\rangle$, $\langle A\#02, \texttt{A.b}\rangle$ and $\langle Z\#00, \texttt{Z.a.b}\rangle$ with context `Z#00.doSmt()` $\rightarrow$ `A#01.getB()`.

Tracking active definitions, reachable uses and executed pairs

The *Runtime Analysis module* is also responsible for *tracking active definitions, reachable uses and executed pairs* of instance state variables. The module incrementally updates three sets of information: the *Active Defs (ADefs)* set of active definitions, the *Reachable Uses (RUses)* set of reachable uses, and the *Executed Pairs (EPairs)* set of executed definition use pairs.

Active Definitions ($ADefs[i]$) The set $ADefs[i]$ of active definition is the set of all the definitions of instance state variables that are active in an instant `i` of the execution, that is, that reached the current basic block. Every time *DReaDs* detects a new definition, it adds it to the set of active definitions. If the set of active definitions already contains a definition on the same instance variable, the old definition is removed from the set (killed) an replaced with the new one.

The $ADefs$ set is populated performing reaching definitions analysis over a single execution trace: $ADefs[i] = ADefs[i-1] \setminus KILL[i] \cup GEN[i]$, where $ADefs[i-1]$ are the definitions that were active in the previous instant of the execution, and

GEN and $KILL$ denote the sets of definitions that start and stop propagating in the current instant i. This equation is a specialisation over a single execution trace of the classic reaching definitions analysis equations, Equations 2.1 and Equations 2.2, discussed in Chapter 2.1.

Reachable Uses ($RUses[i]$) The set $RUses[i]$ of reachable uses contains the uses that are reachable in the instant i from the basic block entry. Every use that is not preceded, in the same block, by a definition on the same state variable is reachable and therefore added to the set. The set is reset at the exit of the block.

Executed Pairs ($EPairs[i]$) The set $EPairs[i]$ of executed definition use pairs contains the observed definition use pairs up to an instant i of the execution. Every time an use is added to the $RUses[i]$ set, it is matched with the active definitions in $ADefs[i]$ on the instance state variable to identify which pair was executed. This set is initialised at empty at the beginning of the execution of a test case, and reset at the end of the execution of each test case.

These sets are used by the next module of *DReaDs* to compute reaching definitions and reachable uses information for the program under analysis, and to register the executed definition use pairs.

4.5 *DReaDs* Generalization Across Traces

This module of *DReaDs* is responsible for generalising the information observed on the single object instances and on the different execution traces to compute reaching definitions and reachable uses information of *class state variables*.

In particular, after each execution step, *DReaDs* exploits the information computed with the previous module about active definitions and reachable uses of *instance* state variables, to update reaching definitions and observable uses of *class* state variables for the current basic block, and to track which pairs have been observed (Algorithm 1, line 10).

To this end, *DReaDs* (i) abstracts class state variables information from instance state variables information computed at runtime (ii) updates reaching definitions for the current basic block, (iii) updates observable uses for the current basic block, and (iv) updates executed definition use pairs information of the current execution.

Abstracting Class State Variables Information

In this step of the analysis *DReaDs* abstracts information on class state variables from information on instance state variables, that is, it combines the information observed on different instances of the same class and of different traces in single reports.

Abstracting a definition (use) of a class state variable from a definition (use) of an instance state variable consists of substituting the $instance\_id$ of the instance with

the $class\_id$ of the type of the instance, as observed dynamically. *DReaDs* abstract information on class state variables every time it enters and exits basic blocks.

This module updates three sets of information: *Generalized Active Defs (GADefs)*, *Generalized Reachable Uses (GRUses)* and *Generalized Executed Pairs (GEPairs)*.

The *Generalized Active Defs[i] (GADefs[i])* set contains the definitions of class state variables that reached the entry (exit) point of a basic block b in an instant i. *DReaDs* updates $GADefs[i]$ by abstracting active definitions of class state variables information from the set $ADefs[i]$ of active definitions of instance state variables whenever it enters and exits a basic block b at an instant i.

The *Generalized Reachable Uses[j] (GRUses[j])* set contains the uses of class state variables that were observed be reachable from the entry point of b in the instant j, and it is updated by abstracting reachable uses of class state variables information from the set $RUses[j]$ of reachable uses of instance state variables.

At the end of the execution of each test case t, *DReaDs* populate the set *Generalized Executed Pairs[t] (GEPairs[t])* by abstracting executed definition use pairs information of class state variables information from the set $GEPairs[i]$ of executed pairs of instance state variables.

Dynamic Reaching Definitions Analysis

DReaDs maintains at runtime the sets $ReachIn[b]$ and $ReachOut[b]$ of reaching definitions that enter and exit a basic block b, for each basic block of the classes under analysis. *DReaDs* rewrites the classic data flow equations of static reaching definitions analysis to exploit dynamically computed information.

When *DReaDs* enters in a basic block b in a specific instant i of the execution, *DReaDs* updates the set $ReachIn[b]$ by adding the definitions in $GADefs[i]$ to the set, according to the Equation 4.1:

$$ReachIn[b] = ReachIn[b] \cup GADefs[i] \tag{4.1}$$

Similarly, when *DReaDs* exit a basic block b in a instant $j > i$, it updates ReachOut using the Equation 4.2:

$$ReachOut[b] = ReachOut[b] \cup GADefs[j] \tag{4.2}$$

DReaDs initializes the sets $ReachIn[b]$ and $ReachOut[b]$ as empty sets at the beginning of the execution, and updates the sets during the execution of the test cases. At the end of the *DReaDs* analysis, the sets contain the reaching definitions that were observed in any instance of an object and in any execution trace of the program.

Dynamic Reachable Uses Analysis

Dynamic reachable uses analysis works similarly to dynamic reaching definitions analysis. For each basic block b, *DReaDs* computes the set $ReachUses[b]$ that includes the

uses of class state variables that have been observed in any block and in any execution trace.

When *DReaDs* exits a basic block b, it updates the set $ReachUses[b]$ adding the uses of $GRUses[i]$ to the set:

$$ReachUses[b] = ReachUses[b] \cup GRUses[i] \tag{4.3}$$

DReaDs initializes the *ReachUses* set is initialised as empty sets at the beginning of the execution, and updates them during the execution of the test cases.

Executed Definition Use Pairs Analysis

At the end of the execution of each test case, *DReaDs* stores the generalised information on the executed definition use pairs observed during that execution in the set $ExecutedPairs$.

DReaDs updates the $ExecutedPairs$ set at the end of the execution of each test t using the simple Equation 4.4:

$$ExecutedPairs = ExecutedPairs \cup GEPairs[t] \tag{4.4}$$

At the end of the execution of the last test case, *DReaDs* summarizes and reports the results of the analysis (Algorithm 1, line 13), and produces the sets *ReachIn*, *ReachOut* and *ReachUses* for each basic block of the classes under analysis, and in the set *ExecutedPairs* containing all the definition use pairs observed during the execution of the suite.

4.6 Prototype Implementation

We designed a prototype implementation of the *DReaDs* technique for the Java programming language, which we called *JDReaDs*.

JDReaDs requires as input a compiled Java program and a driver to execute it, either in the form of one or multiple classes with a main method, or as a JUnit test suite; and computes the data flow information collected while executing the program using the provided driver.

Figure 4.4 shows the logical components of *JDReaDs*, represented as boxes in the figure, and its inputs and outputs, represented as parallelograms. *JDReaDs* contains two main components, the *instrumentation* and the *runtime analysis* component.

The *instrumentation* component is responsible for instrumenting and executing the application under analysis, and is implemented relying on the DiSL framework for dynamic analysis [MVZ+12]. DiSL takes in input a set of instrumentation rules, and dynamically instruments the bytecode of the application according to the provided rules

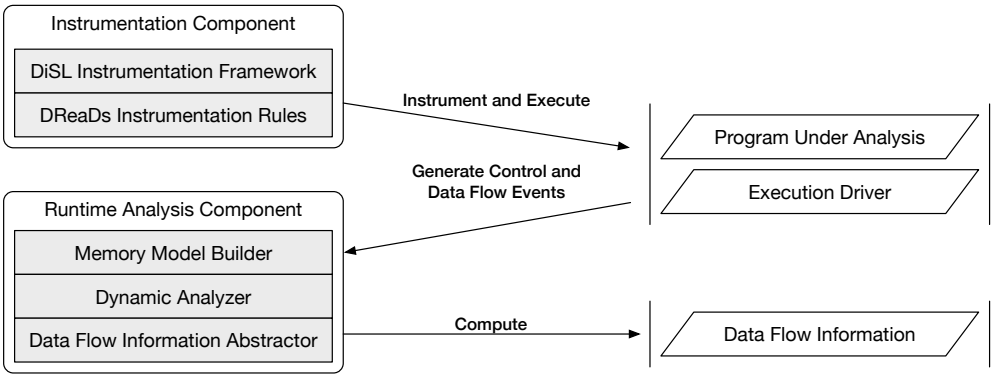


Figure 4.4. JDReaDs modules and workflow

while executing it. *JDReaDs* implements rules for instrumenting the application to trigger a series of events to both track the executed trace, that is, which basic block is executing, which method is called by a call point, etc, and data flow events, such as variable assignments, and reads of references.

The *runtime analysis* component processes the events identified with the *instrumentation* component at runtime. The *runtime analysis* component is composed of three sub-modules that implement the three high-level modules described in Chapter 4 and reported in Algorithm 1: a module for updating and managing the memory model, a module for dynamic tracking active definitions and reachable uses, and a module for abstracting and storing the observed data flow information.

Instrumentation Component

The instrumentation component of *JDReaDs* is responsible for (i) instrumenting the application and its execution driver to collect the necessary information for the analysis, and (ii) executing the (instrumented) application using the provided driver.

JDReaDs instrument the code using the DiSL framework and language that allows to specify instrumentation code as snippets, that is, code templates instantiated for each selected instrumentation site, and offers a series of markers and guards to specify the instrumentation points and scope. DiSL also provides helper classes to retrieve different information on the execution, such as line number, name of a field, etc, and a mechanism for passing information between snippets through the definition of synthetic variables.

We exploit these functionalities of DiSL to insert calls to an event handler class in different points of the original program and of its driver. We exploited DiSL markers to select the instrumentation points, the guards to limit the instrumentation to the classes of our interest, and the synthetic variables to collect detailed information on the objects that are de-referenced by an assignment, to properly register definition kills.

Listing 4.3. Instrumentation rules for tracking `putfield` bytecode instructions

```
1   @SyntheticLocal static Object owner;
2   @SyntheticLocal static Object oldVal;
3   @SyntheticLocal static Object newVal;
4
5   / PUTFIELD /
6   @Before(marker = BytecodeMarker.class, args = "putfield", guard = PutGuard.class,
7       scope = Properties.SCOPE, order = 100)
8   public static void beforePutField(LineNumberStaticContext lnsc, FieldStaticContext fsc,
9       MethodStaticContext msc, DynamicContext dc) {
10          owner = dc.getStackValue(1, Object.class);
11          oldVal = dc.getInstanceFieldValue(owner, fsc.getFieldOwner(), fsc.getFieldName(), ...);
12          newVal = dc.getStackValue(0, Object.class);
13  }
14
15  @AfterReturning(marker = BytecodeMarker.class, args = "putfield", guard = PutGuard.class,
16      scope = Properties.SCOPE, order = 100)
17  public static void afterRetPutField(LineNumberStaticContext lnsc, FieldStaticContext fsc,
18      MethodStaticContext msc, DynamicContext dc) {
19        EventHandler.instanceOf().onInstanceFieldPut(lnsc.getPrevLineNumber(),
20          dc.getThis(), fsc.getFieldOwner(), fsc.getFieldDesc(), fsc.getFieldName(),
21          owner, oldVal, newVal, fsc.isArray(), fsc.isPrimitive()
22          );
23  }
```

Listing 4.3 reports an example of code that contains the instrumentation rules for tracking the assignments of class fields. These rules insert instrumentation code before and after each `putfield` bytecode instruction, which is the bytecode instruction that assigns values to a field. Before each `putfield` (snippet `beforePutField(...)`) we retrieve the objects involved in the assignment, the field owner, the old value and the new value that will be assigned to the field, and we store them in a synthetic local variable. After each `putfield` then (snippet `afterRetPutField(...)`), we trigger the event on the event handler, communicating to the handler different information on the execution, such as the line of code, the field static name and type, and the concrete objects involved in the assignment that we stored in the synthetic variables.

The instrumentation of other instructions related to the control and data flow of the application under analysis is handled similarly. Table 4.1 reports all the instrumentation points of *JDReaDs*, together with a short description.

Runtime Analysis Component

The runtime analysis component is composed of three sub-modules that implement the three high-level modules described in Chapter 4: *Memory Model Builder*, *Dynamic Analyzer*, and *Data Flow Information Abstractor* modules illustrated in Figure 4.4.

The modules react to the events triggered when executing the (instrumented) application, and process the events following the order described in Algorithm 1: Firstly the *Memory Model Builder* updates the memory model, then the *Dynamic Analyzer* updates

Table 4.1. Instrumentation rules of *JDReaDs*

Instrumentation Rule	Description
Before `putfield` After `putfield`	before and after each `putfield` instruction, intercepts assignments of fields
Before `getfield` After `getfield`	before and after each `getfield` instruction, intercepts accesses to field references
Before `arrayload` After `arrayload`	before and after each `arrayload` instruction, intercepts accesses to array elements, distinguishing the index
Before `arraystore` After `arraystore`	before and after each `arraystore` instruction, intercepts assignments of array elements, distinguishing the index
On Basic Block Entry On Basic Block Exit	before and after each basic block entry and exit, intercepts the current basic block.
On Test Entry On Test Exit Before Each Test Line	before and after each test method entry and exit, and before each test line. Intercepts the entry point of the execution.

the information on active definitions and reachable uses of instance state variables, and finally the *Data Flow Information Abstractor* generalizes the information on instances to classes, and updates the reaching definitions and reachable uses set for the current basic block.

The *Memory Model Builder* represents the memory model as a graph, and updates it lazily whenever a field or an array cell is referenced. To this end, it processes every `putfield` and `arraystore` event, updating the model according. The Memory Model Builder exposes utility methods to navigate the graph. These methods returns for each edge, all the nodes that were impacted by the modification of the edge, exploiting a cache mechanism to limit searches over the graph and improve the performances.

The *Dynamic Analyzer* module processes `putfield`, `getfield`, `arrayload`, `arraystore`, basic blocks, and test cases events. It queries the model to retrieve all the nodes impacted by a definition or use event, identifies the corresponding instance state variables, and updates a map of active definitions, reachable uses and executed pairs following the logic described in Section 4.4. The *Dynamic Analyzer* module computes the context of contextual definitions and uses by recording information on the execution trace of the application as the sequence of traversed basic blocks. The *Dynamic Analyzer* module associates each identified definition and use with the execution driver statement which generated it, for debugging and manual analysis purpose.

The *Data Flow Information Abstractor* module incrementally collects and stores the generalized data flow information for each traversed basic block. It processes basic block events: whenever the execution enters or exits a basic block, this module ab-

stracts the information of the active definitions and reachable uses made available by the previous module, and updates the reaching definitions and the reachable uses sets instantiated for each basic block according to the logic discussed in Section 4.5. This module reduces the memory consumption by indexing definitions and uses in a global cache, and using bit vectors over the cached indices to efficiently represent the sets of reaching definitions and reachable uses associated with the blocks. The module updates a set of generalized executed definition use pairs of class state variables. At the end of the executions, this module serializes the computed information in a final report.

JDReaDs works on every program written in Java up to version 1.7, but it currently does not guarantee sound results under certain circumstances: the *JDReaDs* implementation used in this thesis does not handle object cloning, field access via reflection, and unsafe field access. It also has a limited support to multi-threading.

Chapter 5

Static vs Dynamic Data Flow Analysis

This chapter investigates the hypothesis that data flow techniques used so far in data flow testing miss many relevant test objectives that derive from data flow relations entailed by a program. To this end, we compare the data flow relations computed with static data flow approaches with the ones observed while executing the program, for a set of open-source object-oriented programs. The experimental data discussed in the chapter show that data flow testing based on under-approximated static data flow analysis misses many data flow test objectives, and indicate that the amount of feasible but missed data flow test objectives is an obstacle stronger than of infeasible but detected data flow relations to the effectiveness of data flow testing.

In Chapter 3 we hypothesises that the data flow analyses that under-approximate the dynamic behavior of the application, commonly exploited in data flow testing so far, miss many data flow abstractions relevant to testing, thus compromising the effectiveness of the technique.

This chapter presents the results of an experimental evaluation of such hypothesis: We compared the set of data flow relations statically identified with an inter procedural data flow analysis with the data flow elements observed at runtime while executing the program, to estimate the amount of feasible data flow elements that are observed but not statically identified. We experimented with *JDReaDs*, a prototype implementation of the *Dynamic Data Flow Analysis* technique.

The experimental data discussed in the chapter indicate that (i) the current data flow testing approaches based on under-approximated static analyses miss many data flow test objectives, and that (ii) the amount of missing objectives limits the effectiveness of data flow testing more than the amount of the statically identified infeasible data flow relations. These results grounded our idea of defining new data flow testing approaches based on dynamically computed data flow information.

Section 5.1 reports the research questions of the study presented in this chapter. Section 5.2 discusses the motivations and rationale of the experiments. Section 5.3 presents the design of the experiments. Section 5.4 analyzes the results. Section 5.5 discusses our findings. Section 5.6 reports the threats to the validity of the study.

5.1 Research Questions

The study addresses two main research questions.

The first research question is about the ability of an under-approximated static data flow analysis to identify a relevant set of data flow relations:

RQ1: To what extend does static data flow testing identify a set of data flow relations that approximates the data flow relations entailed by a class under test?

The second research question focuses on understanding the impact of approximations over data flow analysis, to understand whether approximations impact more on the number of infeasible or of missed test objectives.

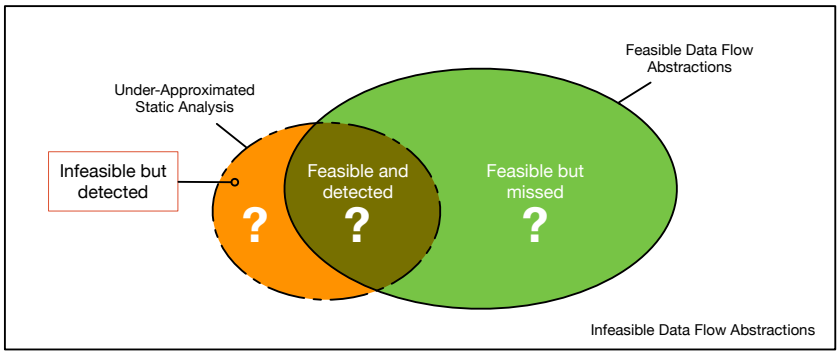
RQ2: To what extent is the outcome of a static technique affected by false-positive (infeasible but detected) or false-negative (feasible but missed) data flow relations?

5.2 Rationale

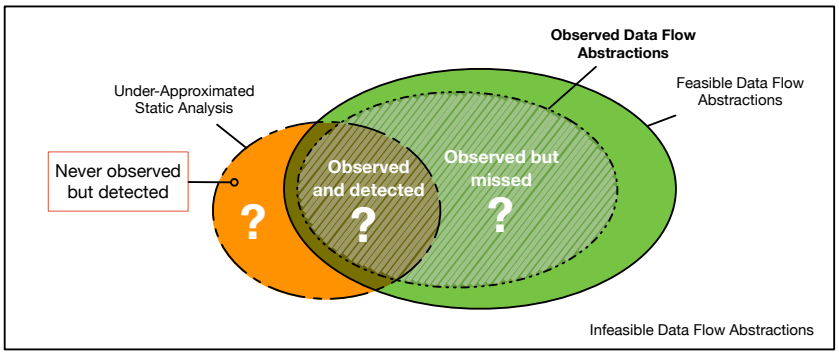
To investigate the research questions, we should compare the data flow information statically computed with the set of feasible data flow abstractions in the application, which requires to identify all the executable data flow abstractions in the code, and compare them with the data flow abstraction statically computed as illustrated in Figure 5.1 (a).

Computing the set of feasible data flow abstractions is undecidable in general, and too expensive to be feasible for non-trivial programs. In our experiment, we approximate the set of feasible data flow abstractions with the set of data flow elements that are dynamically observed with *JDReaDs* while executing the application. To maximize the information dynamically observed, and obtain a good approximation of the set of feasible data flow abstractions, we execute *JDReaDs* with thorough test suites generated both manually and automatically.

We compared the information that we derived statically and dynamically, and computed the three sets of information that are shown in in Figure 5.1 (b): the sets of observed and (statically) missed, observed and (statically) detected and of never observed but (statically) detected elements. We use these sets to estimate the limitations of static analyses in dealing with the dynamic constructs of languages. The amount of observed and missed elements compared to the amount of observed and detected ones would illustrate the suitability of under-approximated static data flow analysis to



(a) Relation between the data flow abstractions identified with a static under-approximated data flow analysis, and the feasible data flow abstractions.



(b) Relation between the data flow abstractions identified with a static under-approximated data flow analysis, and the data flow abstractions observed while executing the program.

Figure 5.1. Relation between the data flow abstractions identified with a static under-approximated data flow analysis, the observable data flow abstractions, and the feasible data flow abstractions.

approximate the set of feasible data flow elements. The amount of never observed but detected elements would indicate the impact of infeasible elements of these analyses on the data-flow testing techniques.

5.3 Experimental Settings

The main independent variable of the study is the technique applied to identify the data flow relations of the sample classes, that is either the static data flow analysis, or the dynamic mechanism *JDReaDs* that computes data flow information at runtime. The depended variable of each observation is the sets of data flow relations both statically and dynamically identified for a class under analysis. Other sources of variability include the classes and the test cases used in the experiments.

In the study, we instantiate the static data flow approach with *DaTeC*, a consolidated tool for data flow testing of object-oriented software that has been used in existing work in literature [DGP08, DGP09, DPV13]. *DaTeC* embodies an inter-procedural reaching definition analysis, and computes contextual data flow information that comply with the definition gave in Section 2.2.2. *DaTeC* computes test objectives for intra and inter-class testing for Java programs, and its original implementation does not include any form of alias analysis to resolve polymorphic types and reference aliasing. To the best of our knowledge, we are not aware of other publicly available tools for inter-procedural data flow analysis that compute contextual data flow information of object-oriented programs.

We compared the information computed using *DaTeC* with the data flow information observed at runtime using *JDReaDs*.

In the study, we focus data flow information relevant for inter-class testing, which corresponds to the set of contextual definitions of class fields that are (may be) executed within a class method, and (may) reach the exit of that method. Both *JDReaDs* and *DaTeC* compute this information as part of the respective reaching definition analyses. Hereafter we refer to these particular sets of reaching definitions as the `defs@exit` of a class method. We measure the `defs@exit` for a class by aggregating the `defs@exit` measured for the methods of the class.

We evaluated the precision of static analysis by comparing the `defs@exit` computed with *DaTeC* with the `defs@exit` observed at runtime with *JDReaDs*. Two definitions computed with *DaTeC* and observed at runtime are the same if they denote an assignment of the same variable made at the same code location through the same chain of methods invocations (context). We implemented some conservative choices in our strategy to obtain a fair comparison. When matching the static and dynamically computed definitions in chains of method invocations, we implement the most aggressive matching between polymorphic types, whose signatures are identified by *DaTeC* and *JDReaDs* as the type declared in the code or the type of the instance observed at runtime, respectively. For instance, we consider that a method that the static analysis identifies as belonging to an abstract class, as belonging to the concrete type that is executed at runtime by the dynamic analysis, since abstract types are not instantiated at runtime. To avoid mismatching, the types observed at runtime match any compatible type statically identified by *DaTeC*. Moreover, *DaTeC* considers the access to a single element of an array as a general access to the array structure. Although at runtime we could distinguish definitions and uses of individual slots of arrays, this feature was disabled in *JDReaDs* to compute a fair comparison.

Table 5.1 lists the characteristics of the Java applications and the test suites that we considered in the study. We considered five open source applications (column *Application*), which range between 3,000 and 55,000 executable lines of code (column *Eloc*), and result to a sample of 1,531 classes (column *#Classes*) that we regard as a statisti-

Table 5.1. Subject applications

Application	Eloc	#Classes	#Tests	Coverage*
Jfreechart	55k	619	26484	0.93
Collections	13k	444	20604	1.00
Lang	11k	150	3254	1.00
Jtopas	3k	63	1562	0.98
JgraphT	6k	255	1009	1.00
	88k	1531	52913	

* Median value of Eloc coverage per class, computed with Cobertura

cally significant number of observations.

We computed the sets of `defs@exit` observed at runtime by executing the subject applications with sample test suites. Since our goal is to stress the possible limitations of the static technique, we tried to maximize the number of the data flow relations observed dynamically by synthesizing thorough test suites that we obtained by augmenting the test cases bundled with the subject applications with automatically generated test cases. We generated test cases with Randoop [PLEB07] and EvoSuite [FA11a, FA13], configuring them with time limits of 120 seconds per application and 180 seconds per class, respectively. In EvoSuite we used the branch fitness function. In Randoop we set a maximum length of 300 lines of code per generated test case.

These test suites produce high statement and branch coverage, as shown in Table 5.1 that reports the cumulative number of bundled and generated test cases that we ran for each subject application (column *Tests*), and the median value of the coverage rates (column *Coverage*) that indicates that the test suites cover a large portion of the classes.

To account for the variability associated with the random-nature of the techniques, Randoop and EvoSuite were run several times.

5.4 Experimental Results

Table 5.2 reports the total number of `defs@exit` computed by *DaTeC* and *JDReaDs* respectively (columns *Total*), and their distribution across the classes considered in our study (columns Q_1 = lower quartile, *Median* and Q_3 = upper quartile).

The data show that at runtime it was possible to observe about an order of magnitude more `defs@exit` relations than the ones statically identified with *DaTeC*, and on average up to 6 times more `defs@exit` than *DaTeC* per class. The data vary across the

Table 5.2. Amount of identified `defs@exit` with *DaTeC* and *JDReaDs*

	d^{DT}: `defs@exit` with *DaTeC*				d^{OB}: `defs@exit` with *JDReaDs*			
Application	Total	Q_1	Median	Q_3	Total	Q_1	Median	Q_3
Jfreechart	20,513	2	9	38	89,415	3	18	75
Collections	3,908	2	4	12	63,460	4	26	81
Lang	1,227	2	3	14	1,638	2	5	13
Jtopas	1,481	3	12	16	8,380	6	39	320
JgraphT	1,800	2	4	16	6,602	1	7	44
	28,929	2	6	18	169,495	3	17	68

projects: Collections registers the larger differences, while the numbers on Lang are similar. The data on the distribution per class show that *JDReaDs* identifies at least 3 times more `defs@exit` than *DaTeC* on the majority of the classes.

Using the approach described in the previous section, we quantified the amount of data flow relations that were distinctively identified by either technique. Table 5.3 reports the amount of `defs@exit` relations that *DaTeC* identified statically but have not been observed dynamically (column *never observed*) and the amount of `defs@exit` data flow relations that *JDReaDs* revealed dynamically *DaTeC* did not compute statically (column *statically missed*).

The data indicate that the imprecision of static techniques is dominated by the statically missed relations (false negatives) over the never observed relations (potential false positives): The data reported in the table indicate that static data flow analysis misses about 96% of the dynamically observed relations, while identifies about 23% of relations that are not exercised by the test cases.

The last two columns of the table further refine the data about the statically missed relations, by showing the amount of data flow information that is statically missed due to polymorphic references (column *impacted by polymorphism*), and handling of arrays (column *impacted by arrays*). The data indicate that the impact of polymorphic reference (67% in average) largely dominate over the impact of array handling (12% in average).

To characterize in further detail our estimations of the relative impact of the false positives and false negatives on the data flow testing of the classes, we inspected the data for each class. Figure 5.2 plots the distribution of the amount of data flow relations that are missed and observed either statically or dynamically across the classes in our sample. Figure 5.2-a indicates the amounts of data flow relations identified dynamically but not statically (*statically missed*), the ones identified statically but not dynamically (*never observed*), all the ones *observed dynamically* and all the ones *identified statically*; Figure 5.2-b plots the proportions of the statically missed relations over all the observed

Table 5.3. Difference sets of `defs@exit` identified with *DaTeC* and *JDReaDs*

Application	Never observed $\#(\in d^{DT} \wedge \notin d^{DR})$	Statically missed $\#(\in d^{DR} \wedge \notin d^{DT})$	Impacted by polymorphism	Impacted by arrays
Jfreechart	3,480 (17%)	85,079 (95%)	47,968 (54%)	11,900 (13%)
Collections	1,779 (46%)	62,169 (98%)	55,295 (87%)	5,580 (9%)
Lang	409 (33%)	1,122 (69%)	605 (37%)	1 (0%)
Jtopas	600 (41%)	8,026 (96%)	5,085 (61%)	2,018 (24%)
JgraphT	505 (28%)	6,080 (92%)	5,259 (80%)	35 (1%)
	6,773 (23%)	162,476 (96%)	114,212 (67%)	19,534 (12%)

ones, and of the never observed relations over all the statically identified ones. We observe that the static technique misses a very large amount (between the 56% and the 99%) of the relations that we observed at runtime for the large majority of the classes (Figure 5.2-b), while only a small portion of the statically identified relations remain unseen after executing the test cases. The distributions of the figures at the numerators of those proportions (Figure 5.2-a) indicate that there are at most 3 never observed relations per-class across the three fourths of the sample, and that *DaTeC* misses up to 60 `defs@exit` per-class in the three fourths of the classes with fewer missed relations.

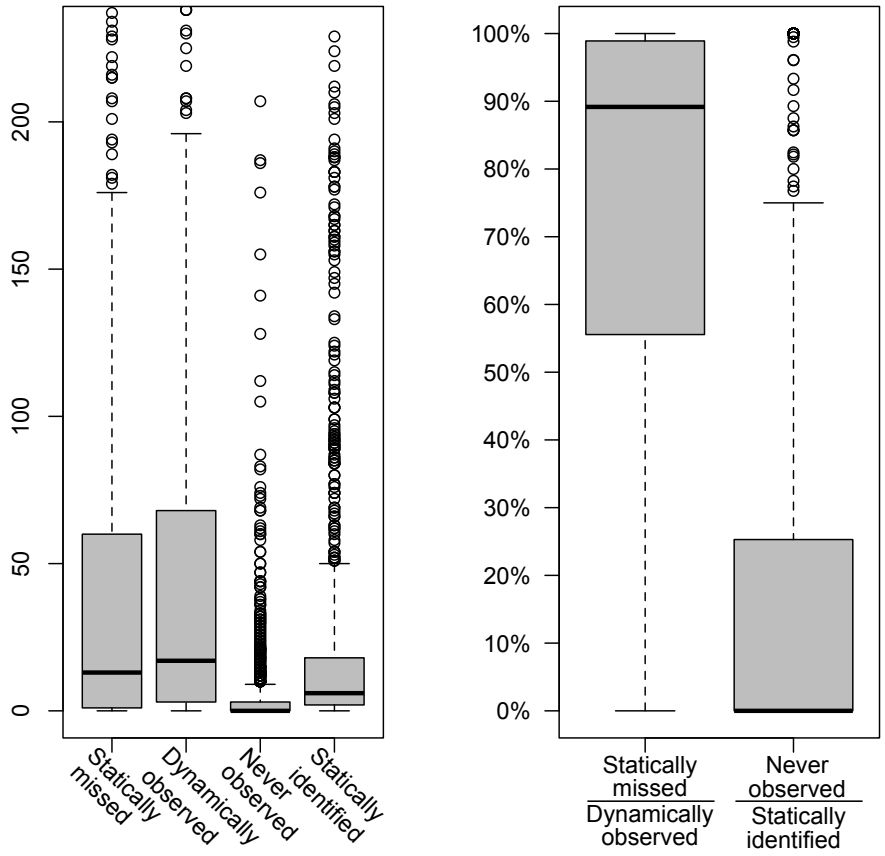
With Student's t-test, we found statistically significant support for the (alternative) hypotheses that the number of statically missed relations exceeds by a factor of 2.4 the number of occurring relations that are also statically identified ($p_i = 0.0293$), and exceeds by a factor of 4.2 the number of statically identified relations that are never observed ($p_i = 0.0468$) across the classes in our sample.

We executed the experiments on a OSX 10.7 MacBook Pro with 2.2 GHz Intel Core i7 and 8GB of RAM memory. We did not plan precise measures of the execution time, since they are not relevant to our experiment.

5.5 Discussion

With respect to the first research question **RQ1**, the results of the study indicate that an under-approximated static data flow analysis badly approximates the set of feasible data flow relations entailed by a program: The empirical data indicate that the set of data flow relations that can be statically identified by an analysis that under-approximate dynamic binding to gain scalability is largely incomplete.

This confirms our hypothesis on the limits of static data flow techniques to identify test targets relative to actual data flow relations, and suggests that to improve data flow testing it is necessary to handle the huge amount of data flow relations that hide when using the static techniques. In other words, if we base data flow testing on a static data



(a) (b)

Figure 5.2. Distributions of the statically missed over the dynamically observed `defs@exit`, and of the never observed over the statically identified `defs@exit`

flow technique which under approximate the dynamic behavior of software, we must be aware that we may miss a considerable amount of data flow relations that shall be accounted as test objectives.

With reference to the second research question about the impact of false-positive and false-negative relations (**RQ2**), this study indicates that the false negatives (statically missed feasible data flow relations) weight much more than the false positives (statically identified infeasible data flow relations) on the imprecision of the static data flow techniques.

The results question what has been suggested in literature so far, which infeasibility is the main source of problems for data flow testing. On the contrary, they indicate that the problem of handling missed information that depends on complex characteristics of languages, difficult to manage using static analysis, is of primary importance to improve data flow testing. This is counterintuitive with respect to what happens

with other structural criteria, like branch coverage, where the static coverage domain is an over-approximation of the possible test objectives, and therefore the challenge is to investigate the reachability of the not yet covered elements.

The results of this experiment ground the idea that the set of data flow relations identifiable with *Dynamic Data Flow Analysis* largely extends the data flow information that can be identified statically, and that *Dynamic Data Flow Analysis* can be use to complement the static approaches in the context of a data flow testing technique to identify more relevant test objectives. These findings also imply that the inconclusive results about data flow testing effectiveness discussed in literature refer to a set of data flow relations identified statically that do not represent the many data flow relations that occur in object-oriented programs. Finding a better way to identify the possible data flow relations in object-oriented programs opens the way to new results about the mutual effectiveness of data flow and structural testing criteria.

5.6 Threats to Validity

We are aware of threats to the internal, construct and external validity of our study. The internal validity can be threatened by a scarce control on factors that may influence the results. The construct validity requires that the operational implementation of the variables properly captures the intended theoretical concepts, and that the measurements are reliable. The external validity relates the generalizability of the findings.

The main *internal* threat in this study relies in the difficulty of measuring the consistency of our results across different implementations of static data flow techniques. In our study, we refer to the technique embodied in *DaTeC* that implements a mature and consolidated approach to data flow testing of object-oriented programs consistent with the approaches proposed in the main recent studies on this topic. We already commented on the range of design choices that can underlie the implementation of a static data flow technique: As many other static analyzers, *DaTeC* relies on both conservative choices, like choices concerning the feasibility of statements, paths or matching array offsets, and approximations due to the impossibility of accounting for all alias relations, as needed in particular to precisely solve polymorphic method calls. We are aware that implementations of the data flow technique different than *DaTeC* can lead to identify differently approximated sets of data flow relations. The current unavailability of other tools for inter-procedural data flow analysis of object-oriented programs inhibited us from extending our observations beyond *DaTeC*.

Another important *internal* threat to validity refers to how well `defs@exit` reaching definitions appropriately represent the objectives of data flow class testing. Classic data flow class testing addresses the interactions (*def-use* relations) between the methods of a class that can define and use the same class state variables, when invoked sequentially in a test case [HR94]. We observe that identifying the `defs@exit` reaching definitions

is a pre-requisite for a static technique to identify the def-use relations, since only the definitions that reach the method exit may propagate to uses in other methods. Thus, the set of `defs@exit` reaching definitions characterizes well the ability of identifying def-use interactions. Measuring directly the def-use interactions would lead to fewer interpretable results in our study. The reason is twofold. First, since *DaTeC* computes the def-use relations by pairing the `defs@exit` reaching definitions and the reachable uses of the methods, after computing either information separately, the imprecisions of the data flow analysis would be reflected with combinatorial confounding effects in the measurements of the def-use relations. Second, the def-use relations predicate on the combined execution of pairs of methods, and this increases the dependence of the dynamic data on the test suites, an effect that our study aims to minimize.

A threat to the *construct* validity refers to the dependency of the data flow relations that we observed dynamically in our study on the test cases. We have executed all the test cases bundled with the subject applications augmented with test cases generated with the most popular open source automatic test case generators. There is no indication that the bundled test cases have been built with data flow testing in mind, and neither the random nor the search-based tools used in the experiment address data flow criteria directly. Thus, we cannot exclude that our set of dynamic observations could be incomplete. However, because the study reveals a huge disproportion between the amounts of statically identified and dynamically observed data flow relations, we are confident that further (currently missed) dynamic observations would not significantly alter the current results.

Another important threat to the *construct* validity refers to the reliability of our measurements that depend on the reliability of the data computed with *DaTeC* and *JDReaDs*. We extensively tested and used *DaTeC* over the last years. We developed *JDReaDs* for this experiment, and we have tested it by manually inspecting the outcome produced on a sample of the classes considered in this experiment. *JDReaDs* however does not properly handle multi-threading and Java reflection. Therefore, we excluded classes that relies on those constructs from the experiment. An extension of *JDReaDs* to properly handle these constructs will likely further increase the precision gap between the two approaches, since also *DaTeC* does not implement any specific strategy for reasoning on threads, and does not handle Java reflection.

The main threat to the *external* validity concerns the limits of our subjects that include only open-source applications. Thus, we shall restrict the scope of our conclusions to open-source software. In general, we are aware that the results of a single scientific experiment cannot be directly generalized.

Chapter 6

Dynamic Data Flow Testing

In this section we introduce Dynamic Data Flow Testing, a new technique for automatically generating test cases to properly exercise the state based behavior of object-oriented systems captured with data flow relations. The technique promotes the synergies between dynamic analysis, static reasoning and test case generation. Dynamic Data Flow Testing relies on Dynamic Data Flow Analysis to compute precise data flow information for the program under test. It statically processes the dynamically computed data flow information to infer yet-to-be-executed test objectives that it uses to generate new test cases. Thanks to Dynamic Data Flow Analysis, Dynamic Data Flow Testing suffers less from the problems of current static data flow testing approaches, being able to both identify definition use pairs that depend on the dynamic behavior of the application, and limit the impact of infeasible elements.

This chapter presents *Dynamic Data Flow Testing*, a structural testing technique to automatically generate test cases for software systems characterised by a stateful behavior and a high degree of dynamism, such as object-oriented systems.

The technique builds on the rationale of classic data flow testing, following the idea that definition use pairs capture the state based interactions between objects. However, *Dynamic Data Flow Testing* does not follow the typical data flow testing framework that employs a static analysis to detect test objectives, but exploits dynamic analysis for steering the test generation process.

Dynamic Data Flow Testing exploits the dynamic data flow information computed on an initial set of test cases to infer new data flow relations and consequently new test objectives. Being inferred on dynamically computed data, the new data flow relations both include important information on the execution state that cannot be captured statically, and suffer less from the infeasibility problem than the data flow relations computed with static analysis. We iteratively generate test cases until we cannot identify new test objectives to be exercised.

The test cases obtained using *Dynamic Data Flow Testing* exercise the complex state

based behavior of object-oriented systems, which may not be well exercised while using other structural testing approaches.

The chapter is organised as following: Section 6.1 introduces the guiding principle in the design of *Dynamic Data Flow Testing*. Section 6.2 describes the workflow and the architecture of *Dynamic Data Flow Testing*, which is composed of a component that deploys *DReaDs* to perform dynamic data flow analysis (Section 6.3), a component that infers test objectives from the information computed dynamically (Section 6.4) and a component that generates test cases that satisfy the identified objectives (Section 6.5).

6.1 Objectives

Data flow testing techniques are well suited for testing object-oriented systems. By using inter-procedural data flow abstractions as test objectives, such as contextual definition use pairs of class state variables, data flow testing approaches select test cases that can exercise the state based behavior of classes, properly capturing the data dependencies between objects and methods [MOP02, SP03, DGP08].

The applicability and effectiveness of data flow testing is challenged by the imprecision of the static analysis used to identify the test objectives, that has a potentially huge impact on the success of a data flow testing approach, as discussed in Chapter 5.

In this chapter, we propose *Dynamic Data Flow Testing*, an original technique that re-thinks classic data flow testing to increase both its effectiveness and applicability.

The core idea of *Dynamic Data Flow Testing* is to overcome the problem of the imprecision of static analysis by using *Dynamic Data Flow Analysis*, which computes sound data flow information that better approximates the set of feasible data flow abstractions of the program, without under-approximating the dynamic and complex behavior of the application. Dynamically computed data flow information identifies the dependencies between objects that derive from the interplay between the static structure of the program and the dynamic evolution of the system, which are typically missed by static analysis, and that identify complex software behavior that can contain subtle failures and thus is important to be tested.

To identify new test objectives from information computed dynamically, *Dynamic Data Flow Testing* combines data flow information observed on different object instances and execution traces. In particular, *Dynamic Data Flow Testing* infers yet-to-be-executed definition use pairs by statically pairing definition and uses dynamically computed on different execution traces and on different instances of the same class. The definition use pairs identified in this way are computed on dynamically produced information, which guarantees the feasibility of the definitions and uses in the pairs. Thus, the newly inferred pairs suffer only from the infeasibility of the pair, but not of the constituting elements, reducing the infeasibility problem that affects static approaches.

Dynamic Data Flow Testing automatically generate test cases that satisfy the identified data flow relations. By providing an automated strategy to data flow test genera-

Algorithm 2 Dynamic Data Flow Testing(Program, TestSuite)

Require: Program: The program under test
Require: TestSuite: A test suite for the program under test
1: DynamicDFInfo = EMPTY
2: TestObjectives = EMPTY
3: NewTestCases = EMPTY
4: **repeat**
5: TestSuite = addNewTests(TestSuite, NewTestCases)
6: DynamicDFInfo = executeDReaDs(Program, TestSuite) ▷ section 4.3
7: TestObjectives = inferPairs(DynamicDFInfo) ▷ section 4.3
8: NewTestCases = generateTests(TestObjectives) ▷ section 4.3
9: **until** ¬isEmpty(NewTestCases) ∧ hasExecutionBudget()
10: **return** TestSuite

tion, *Dynamic Data Flow Testing* provides a solution to the problem of the complexity of generating test cases that satisfy data flow abstractions.

Finally, *Dynamic Data Flow Testing* defines a new approach to structural testing. Instead of computing the universe of test objectives upfront and then trying to satisfy them all, it incrementally discovers the interesting test objectives while generating test cases that satisfy them. The technique exploits the dynamic data flow information computed on an initial set of test cases to infer new test objectives, which are targeted by a test generator to produce new test cases. These new test cases are added to the pool of existing tests, and are iteratively analyzed with *Dynamic Data Flow Analysis*.

6.2 The Iterative Dynamic Data Flow Testing Approach

Dynamic Data Flow Testing exploits the synergies between dynamic analysis, static analysis and test case generation. The technique works iteratively, alternating dynamic analysis, static reasoning and test case generation to improve an original test suite with new test cases that execute dynamically computed data flow abstractions.

Algorithm 2 and Figure 6.1 introduce the workflow and the architecture of *Dynamic Data Flow Testing*. *Dynamic Data Flow Testing* includes three main steps: Dynamic Data Flow Analysis, Test Objectives Inference Analysis and Test Case Generation, represented in Figure 6.1 as circles.

Dynamic Data Flow Testing takes as input a program under test and a test suite, and iteratively executes the three steps (Algorithm 2, line 4). During each iteration, *Dynamic Data Flow Testing* augments the original suite with new test cases (line 8), and terminates after the first iteration that does not augment the test suite.

In each iteration, *Dynamic Data Flow Testing* executes the three steps, references in Algorithm 2 as `executeDReaDs`, `inferPairs` and `generateTests`. The Dynamic Data Flow Analysis step (`executeDReaDs`, line 6) executes the test cases and analyzes the execution traces to dynamically compute data flow information on the propagation of

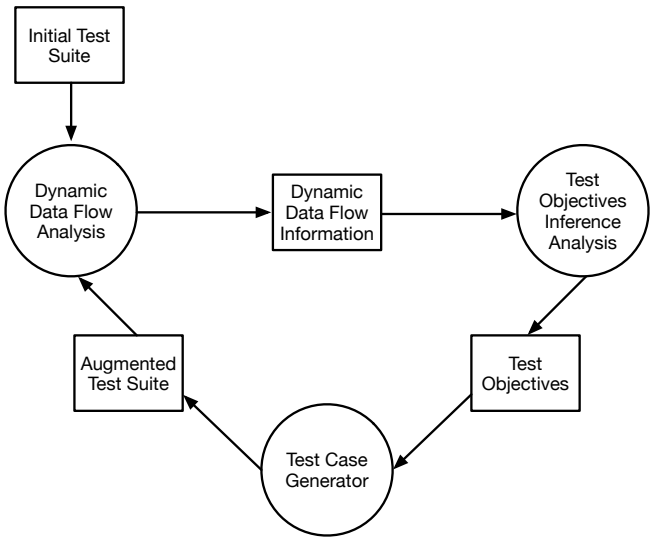


Figure 6.1. Workflow and Components of *Dynamic Data Flow Testing*

definitions and the reachability of uses (variable `DynamicDFInfo`, line 6).

The Test Objectives Inference Analysis step (`inferPairs`, line 7) infers not yet executed definition use pairs from the dynamic data flow information computed so far (variable `TestObjectives`, line 7). In particular, *Dynamic Data Flow Testing* matches definitions and uses observed on different instances and objects to identify new definition use pairs to use as test objectives.

The Test Case Generation step (`generateTests`, line 8) derives a set of test cases that satisfy the test objectives identified in the second step (variable `NewTestCases`, line 8). *Dynamic Data Flow Testing* adds the new test cases to the initial suite (line 5) to start a new cycle. The execution traces produced by executing the new test cases provide new data flow information material, that could lead to new data flow information, and new test objectives.

Dynamic Data Flow Testing terminates when either *Dynamic Data Flow Analysis* does not identify new test objectives and therefore the technique does not generate any new test case, or the execution budget is over (line 9).

The working mechanism of *Dynamic Data Flow Testing* can be illustrated through a brief example on the code reported in Listing 3.4, which we previously used to discuss the limits of over and under- approximated static data flow analysis in Chapter 3. Starting with the basic test suite shown in Listing 6.1, *Dynamic Data Flow Testing* can incrementally discover all the feasible pairs shown in Table 6.1 and generate test cases that execute all the identified pairs. The generated test cases leads to the failure of the

Table 6.1. Feasible definition use pairs of class `Cut` of Listing 3.4

Pair	Variable	Def at	Through call chain	Use at	Through call chain
p1	Cut.v0	2	Cut.<init>	10	Cut.met2
p2	Cut.v0	6	Cut.met1	10	Cut.met2
p3	Cut.lev[L1].v1	29	Cut.met2→L1.doB	26	Cut.met1→L1.doA
p4	Cut.lev[L1].sub1[L2].v2	41	Cut.met1→L1.doA→L2.doA	44	Cut.met2→L1.doB→L2.doB

Table 6.2. Definitions and uses dynamically revealed against the execution of method sequences

Iteration	Executed Test Case ⋆	Executed Pair ∗	Observed definitions	Observed uses	Inferred Pair ∗	Generated Test Case ⋆
1	testMet1()		Cut.v0	Cut.lev.v1		
	testMet2()	p1		Cut.v0	p2	newTest1()
2	newTest1()	p2	Cut.lev.v1	Cut.lev.sub1.v2	p3	newTest2()
3	newTest2()	p3	Cut.lev.sub1.v2		p4	newTest3()
4	newTest3()	p4				

∗ definition use pairs identified as in Table 6.1.
⋆ test cases reported in Listing 6.1 and Listing 6.2.

assertion at line 11 of the class `Cut`, exposing a fault.

Table 6.2 summarizes the information produces in each iterations of *Dynamic Data Flow Testing*. Each row reports the iteration (column *Iteration*), the executed test cases (column *Executed Test Case*), the new executed pairs in the iteration (column *Executed Pair*), the definitions and uses observed on the execution traces (columns *Observed definitions* and *Observed uses*, respectively), the definition use pairs that can be inferred based on the observed information so far (column *Inferred Pair*) and the new test case generated at the end of the iteration to satisfy the inferred pairs, if any (column *Generated Test Case*).

In the first iteration, *Dynamic Data Flow Testing* executes the existing test suite composed of the test cases `testMet1()` and `testMet2()`, reported in Listing 6.1. These test cases independently invoke the two methods `Cut.met1()` and `Cut.met2()` of class `Cut` of Listing 3.4.

By analyzing the execution traces of `testMet1()` and `testMet2()` with *DReaDs*, *Dynamic Data Flow Testing* identifies that the test `testMet1()` executes a definition of variable `Cut.v0` and a use of variable `Cut.lev.v1`, while `testMet2()` executes the pair $p1$ and uses variable `Cut.v0` (as reported in the lines *Iteration* 1 of Table 6.2).

While the definition and the use executed in the first test case does not identify any

Listing 6.1. Initial Simple Suite for Class `Cut` of Listing 3.4

```
1  @Test
2  public void testMet1() {
3      Cut cut = new Cut();
4      cut.met1();
5  }
6
7  @Test
8  public void testMet2() {
9      Cut cut = new Cut();
10     cut.met2();
11 }
```

Listing 6.2. Test Cases for Class `Cut` Generated by Dynamic Data Flow Testing

```
1  //covers p2
2  @Test
3  public void newTest1() {
4      Cut cut = new Cut();
5      cut.met1();
6      cut.met2();
7  }
8
9
10 //covers p3
11 @Test
12 public void newTest2() {
13     Cut cut = new Cut();
14     cut.met1();
15     cut.met2();
16     cut.met1();
17 }
18
19 //covers p4
20 @Test
21 public void newTest2() {
22 Cut cut = new Cut();
23     cut.met1();
24     cut.met2();
25     cut.met1();
26     cut.met2();
27 }
```

pair, the additional pair executed in the second test reveals a new pair $p2$ for variable `Cut.v0`, which identifies a data dependency between `Cut.met1()` and `Cut.met2()`. This pair is identified in the Test Objectives Inference Analysis phase of *Dynamic Data Flow Testing* by matching the definition observed in the execution trace of `testMet1()`, with the use observed in the trace of `testMet2()`.

In the test generator phase, *Dynamic Data Flow Testing* generates a new test case that executes the pair $p2$, for instance the test `newTest1()` reported in Listing 6.2, which subsequently invokes the methods `Cut.met1()` and `Cut.met2()` covering the pair.

The next runs execute the inferred and not-yet-executed pairs incrementally. In the second step, the new test case `newTest1()` is added to the set of tests to use for the analysis, and its execution leads to the identification of a definition on the variable `Cut.lev.v1`, which allows the inference of the pair $p3$ (line *Iteration* 2 of Table 6.2). The test case generated to execute pair $p3$ — the test `newTest2()`, reported in Listing 6.2 — is analyzed in the third iteration, reveals a new definition, and infers the pair $p4$, which is covered by the last test case `newTest3()` (line *Iteration* 3 of Table 6.2). The non-trivial combination of method calls executed by `newTest3()` leads to the failure of the assertion at line 11 of class `Cut`, exposing the fault.

Since the last test case does not reveal new definitions or uses to infer new pairs and generate new tests, the process terminates with the fourth iteration of *Dynamic Data Flow Testing* (line *Iteration* 4 of Table 6.2).

In this case the technique identifies all the feasible and important data flow relations without false positives, and guides the incremental generation of test cases to find the failure.

The following sections details the three steps of *Dynamic Data Flow Testing*. Section 6.3 presents how *Dynamic Data Flow Analysis* is used for the technique, Section 6.4 focuses on the static approach used to compute test objectives, and Section 6.5 discusses the automated generation of test cases from such goals.

6.3 Dynamic Data Flow Analysis

Dynamic Data Flow Testing exploits the *Dynamic Data Flow Analysis* (*DReaDs*) technique presented in Chapter 4 to compute precise data flow information from which *Dynamic Data Flow Testing* infers new test objectives. In the dynamic data flow analysis phase, *Dynamic Data Flow Testing* executes the program with the current test suite, and produces the contextual data flow information observed while executing the program, starting with an initial test suite. The contextual data flow information computed in this phase includes the set of reaching definitions, the set of reachable uses and the set of executed definition use pairs of the class state variables for each executed basic block.

In the *Dynamic Data Flow Analysis* phase, *Dynamic Data Flow Testing* selects the data flow information useful for inferring interesting test cases by excluding the data flow events that depend on aliasing, and that are triggered by accessing a variable through an escaped reference of some objects that are not under test. *Dynamic Data Flow Testing* excludes these data flow events because they can lead to definition use pairs that are not relevant for testing.

Consider, for example, the classes in Listing 6.3, the test case in Listing 6.4 and the memory model reported in Figure 6.2. Listing 6.3 comprises three classes under test `Z`, `A` and `B`; Listing 6.4 reports a test case *t_e* that execute the classes, and Figure 6.2 shows the memory model created by *DReaDs* when executing the test case *t_e*. The memory model in the figure represents the state of the objects in memory after executing the test case *t_e*.

Focus on the last line of the test *t_e* (line 9, `a2.getB()`). In the execution state represented with the memory model in the figure, the variable `a2` is an alias of the object `A#02`, and the invocation of the method `A.getB()` on `A#02` triggers a read event on the object `B#03`. Since `B#03` is part of the state of `A#02` according to the memory model shown in the figure, the read event on `B#03` generated by executing line `a2.getB()` is recorded by *DReaDs* as an *use* of the class state variable *A.b* through the invocation context `A.getB()`. However, `B#03` is also part of the state of `A#01`, which is part of the state of `Z#00`, as pictured in the memory model. Therefore, by navigating the model, *DReaDs*

```
1  class Z{
2    private A a;
3    Z(A a){
4      this.a=a;
5    }
6    void doSmt(){
7      print(a.getB());
8    }
9  }
10
11
12 class A{
13   private B b;
14   void setB(B b){
15     this.b=b;
16   }
17   B getB(){
18     return b;
19   }
20 }
21 class B{
22 }
```

Listing 6.3. Classes Under Test

```
1  @Test
2  public void t_e() {
3    A a1 = new A();
4    A a2 = new A();
5    B b = new B();
6    a1.setB(b);
7    a2.setB(b);
8    Z z = new Z(a1);
9    a2.getB();
10 }
```

Listing 6.4. Test *t_e* for classes `Z A B`

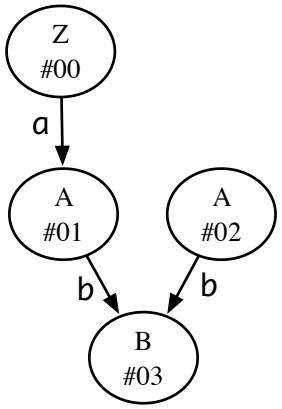


Figure 6.2. Memory Model obtained at the end of the execution of *t_e*

detects that the read event in line `a2.getB()` in the invocation context `A.getB()`, is a use of both the class state variables `A.b` and `Z.a.b`.

This use of `Z.a.b` within the context `A.getB()` has little relevance for generating test cases for class `Z`, because it does not identify any legal combination of methods of `Z`, and thus can divert the testing effort. *Dynamic Data Flow Testing* excludes these definitions and uses, identifying only the set of data flow elements relevant for testing.

Formally, *Dynamic Data Flow Testing* distinguishes relevant and non-relevant data flow elements by selecting definitions and uses according to the variables and the contexts. According to Definition 2.2, which defines contextual definitions and uses of class state variable as tuples $(var, \langle m_1, m_2, ..., m_n \rangle, l)$, where var is the variable name of a class c, $\langle m_1, m_2, ..., m_n \rangle$ is the sequence of method invocations that leads to the definition or use, and l is the line of code of the definition or use event, *Dynamic Data Flow Testing* identifies as non relevant definitions and uses where (i) Method m_1 belong to a class different from the class c that contains the defined (used) variable var, and (ii) the class c is not under test.

With respect to the example discussed above, *Dynamic Data Flow Testing* identifies the use of the variable `Z.a.b` with context `A.getB()` as non relevant for testing, because the used variable and the invocation context involve two different objects (i.e., a use of a variable of `Z` through the invocation of a method of `A`), and the use of `A.b` with context `A.getB()` as relevant, because both the variable and the invocation context involve the

same object A.

6.4 Inference of Test Objectives from Execution Traces

In the second step, *Dynamic Data Flow Testing* uses the data flow information computed dynamically in the first step to identify new test objectives that correspond to not-yet-executed data flow elements. *Dynamic Data Flow Testing* focuses on generating test cases that exercise the interactions of methods through state variables, by considering contextual definition use pairs of class state variables as test objectives.

Dynamic Data Flow Testing identifies new definition use pairs by pairing definitions and uses that have been observed on different instances and execution traces, but never executed together as a pair. *Dynamic Data Flow Testing* (i) uses the output of *Dynamic Data Flow Analysis* to compute summaries of reaching definitions and reachable uses for each method, and (ii) statically matches the information of different methods to infer new pairs, while excluding the pairs already covered.

In the *Test Objectives Inference* phase, *Dynamic Data Flow Testing* re-organizes the information produced with *DReaDs* by computing for each method m both the set *defs@exit* of definitions that reach the exit of m along at least one execution, and the set *uses@entry* of uses reachable from the entry of m along at least one execution:

Definition 6.1 (defs@exit)**.** The set $defs@exit_m$ of class method m is the set of the definitions of class state variables that are defined within m and have reached the exit of m in at least one execution, according to the information computed dynamically by *Dynamic Data Flow Analysis*.

Definition 6.2 (uses@entry)**.** The set $uses@entry_m$ of a class method m is the set of uses of class state variables that are defined within m and have been reached from the entry of m in at least one execution, according to the information computed dynamically by *Dynamic Data Flow Analysis*.

Then *Dynamic Data Flow Testing* infers new test objectives by computing the set of pairs $\langle defs@exit_{m'}, uses@entry_{m''}\rangle$ for all methods m' and m'', obtained by matching the definitions and the uses of the same class state variables, and by removing the already executed pairs from the set.

The set of pairs $\langle defs@exit_{m'}, uses@entry_{m''}\rangle$ is computed by statically matching *defs@exit* and *uses@entry*, and can include infeasible pairs. However, differently from classic static data flow analysis, the new test objectives are computed from dynamically produced information, which includes only feasible definitions and uses by construction. Thus the approach suffers only from the possible infeasibility of the pairing, but not of the single elements.

Dynamic Data Flow Testing terminates when it cannot infer any new test objectives.

6.5 Test Case Generation

In the third step, *Dynamic Data Flow Testing* derives new test cases that execute the newly identified test objectives. *Dynamic Data Flow Testing* is not bounded to a specific approach for generating the new test cases. In our experiments, we automatically generated the test cases relying on a *search based* test case generation strategy.

A search based test case generation approach exploits some meta-heuristic algorithms to automatically generate test cases that satisfy a given set of objectives. These algorithms iteratively search or evolve a population of individuals, test cases in our case, by exploiting on a fitness function that measure the distance of the individuals from the optimal solution.

Current search based approaches exploit fitness functions that measure the structural adequacy of the individuals to generate test cases that satisfy control and (static) data flow criteria. We survey such work in Chapter 8.

We implemented the *Dynamic Data Flow Testing* Test Case Generation step by defining a fitness function that extends existing search based techniques to satisfy contextual definition use pairs. We design the fitness function to work with evolutionary algorithms that use genetic algorithms to evolve a population of test cases. Below we introduce search based testing and detail our proposed approach.

Search Based Testing

Search Based testing casts the problem of test generation as a search problem, and applies efficient search algorithms such as genetic algorithms (GAs) to generate test cases [McM04]. The approach is not bounded to a specific criterion, but can be adapted to different criteria by defining new heuristics. Genetic algorithms incrementally evolve an initial population of candidate individuals by means of search operators inspired from natural evolution. The crossover between pairs of individuals produces two offspring that contain some genetic material from both parents, and the mutation of individuals introduces new genetic material. The representation and search operators depend on the test generation problem at hand, for instance sequences of method calls for testing classes [Ton04].

The fitness function measures how good individuals are with respect to the optimization target, and the better the fitness value is, the higher the probability of an individual for being selected for reproduction, thus gradually improving the fitness of the best individual with each generation, until either an optimal solution is found or some other stopping condition, for instance a timeout, is reached.

The fitness function executes the program under test with an input at a time, and measures how close this input is to reach a particular structural entity chosen as the optimization target. For example when optimizing either statement or branch coverage, the structural entity chosen as target is a node of the control flow graph of the application.

When optimizing for targeting a node of the control flow graph, the heuristic to measure the distance of a test with respect to a node is both the *approach level*, which measures the distance of the execution trace of the current test case from the target node and the *branch distance*, which estimates how close a particular branch was to evaluate to the desired outcome [McM04].

Approach level and branch distance are combined in the following function, which is commonly used as a fitness function to target a node of the control flow graph of a program:

$$nodeDistance = \text{approach level} + \text{normalize(branch distance)} \tag{6.1}$$

This is a minimising fitness function, that is, the target is covered if this function evaluates to 0. The branch distance is calculated for the branch where the control flow diverged, that is the point of diversion measured by the approach level. Since the approach level is an integer number, the branch distance is commonly normalized in the range $[0,1]$ using a normalization function, such that the approach level always dominates the branch distance.

Search Based Dynamic Data Flow Testing

We extended search based testing techniques [FA11a, BLM10] by implementing a fitness function that targets the contextual definition use pairs computed in the former two steps.

We define a fitness function that targets contextual definition use pairs by (i) defining a fitness function to target contextual definitions and uses, and (ii) extending the fitness function for targeting pairs by exploiting the fitness function for the single elements. The reader should notice that we focus on search based techniques that target the optimization of individual tests for individual coverage goals.

Contextual Definitions and Uses

We represent contextual definitions and uses as node-node fitness functions according to the categorization proposed by Wegener et al. [WBS01] that is, we represent a contextual definition or use as a series of nodes of the inter-procedural control flow graph, and use some standard fitness metrics such as Equation 6.1 to measure the distance from a node.

Given a contextual data flow element (either a definition d or a use u) $e = (var, \langle m_1, m_2, ..., m_n \rangle, l)$, as defined in Definition 2.2, where var is the variable name of a class c, $\langle m_1, m_2, ..., m_n \rangle$ is the sequence of method invocations that leads to the definition or use, and l is the line of code of the definition or use event; our fitness function represents e as tuple of nodes of the control flow graph $\langle n_1, n_2, ..., n_m \rangle$ that maps $\langle n_1, n_2 ..., n_{m-1} \rangle$ to the call points $\langle m_1, m_2, ..., m_n \rangle$, and n_m to l:

$$e = \langle n_1, n_2, ..., n_m \rangle \tag{6.2}$$

We define a fitness function that targets subsequent tuples of nodes by exploiting the standard fitness metric reported in Equation 6.1 to guide the optimization towards each node. The fitness returns 0 if and only if the input traverses each node in the correct order, otherwise it returns the sum of the distance to reach each uncovered node.

Formally, given an execution trace t and a contextual data flow element (either a definition or a use) $e = \langle n_1, n_2 ..., n_m \rangle$, we define the fitness function that target e as follows:

$$contextualElementFitness(\langle n_1, n_2 ..., n_m \rangle, t) = \sum_{i=1...m} \text{fitnessSingleNode}(n_i, t) \tag{6.3}$$

$$fitnessSingleNode(n_i, t) = \begin{cases} nodeDistance(n_i, t) \\ \text{if } i = 0 \vee nodeDistance(n_{i-1}, t) = 0 \\ 1 \\ \text{if } nodeDistance(n_{i-1}, t) > 0 \end{cases}$$

Thus we define a minimization fitness function, as in Equation 6.1.

Contextual Definition Use Pairs

To cover a definition use pair, we shall define an optimization that reaches both the definition and the use in this order. We defined such fitness function by combining two instances of Equation 6.3, for the definition and for the use, respectively. To properly cover the pair, we need also to ensure both that there are no killing definitions between the source definition and the target use, and that the definition and the use are covered on the same object instance.

Dynamic Data Flow Testing satisfy both requirements by using the fitness function $contextualElementFitness$ of Equation 6.3 to steer the optimisation towards the definition of the pair. If *Dynamic Data Flow Testing* reaches the definition, it analyzes the execution trace generated by the test to identify the sub-traces that are both kill-free with respect to the definition and that use the same object. Finally, *Dynamic Data Flow Testing* retrieves the best value for computing the final fitness of the pair by applying Equation 6.3 on the uses of each of the identified sub-traces. Below we describe the approach in detail.

Dynamic Data Flow Testing executes a test case and identifies the trace t that distinguishes the called object instances:

$$t = \langle (n_1, o_1), (n_2, o_2), ..., (n_m, o_m) \rangle \tag{6.4}$$

where n_i represents a node in the inter-procedural control flow graph of the class under test, and o_i is a unique ID that identifies the instance on which the node was executed. We denote a sub-trace of t from (n_i, o_i) to (n_j, o_j) (inclusive) as $t_{i,j}$, with $i \leq j$.

Given a trace t and a data flow event $e = \langle n_1, n_2 ..., n_m \rangle$ (either a definition d or a use u, as defined above in Equation 6.2), we define a data flow event that have been executed in the trace t as $e(t) = \langle (n_1, o_1), (n_2, o_2), ..., (n_m, o_m) \rangle$, where n_i identifies the nodes of the inter procedural control flow graph executed in t that identify the definition or the use, and o_i identifies the object instances on which each node was executed in t.

Given a trace t, a contextual definition $d(t) = \langle (n_1^d, o_1^d), (n_2^d, o_2^d), ..., (n_m^d, o_m^d) \rangle$ and a contextual use $u(t) = \langle (n_1^u, o_1^u), (n_2^u, o_2^u), ..., (n_n^u, o_n^u) \rangle$ of the same variable v, the pair $\langle d, u \rangle$ is executed in the trace t iff the chains of method invocations of both the definition and the use start from the same object instance, and the value that is defined and read belongs to the same object instance, that is, iff $o_1^d = o_1^u$, and $o_m^d = o_n^u$.

Given a trace t, a definition $d(t) = \langle (n_1^d, o_1^d), (n_2^d, o_2^d), ..., (n_m^d, o_m^d) \rangle$ observed on t, and a target use $u = \langle n_1^u, n_2^u, ..., n_n^u \rangle$, we define $utraces(t, d(t), u)$ the set sub-traces of t which are kill-free with respect of $d(t)$, and that contains a full or partial match of the use u. In detail, we define $utraces(t, d(t), u)$ as a set $(t_{i,x}, t_{i,y}, ...t_{i,z})$ of sub-traces of t such that:

- each sub-trace $t_{i,j}$ starts from the definition $d(t)$, that is, the index i corresponds to the position of (n_m^d, o_m^d) in the trace t,
- for each sub-trace $t_{i,j}$, j is either the position of the next occurrence of a killing definition $d'(t)$ of the same variable and on the same objects of $d(t)$, or the end of the execution trace if there is no further definition.
- each sub-trace $t_{i,j}$ contains either a chain of nodes-ids that correspond to a partial march of u, that is, a chain of nodes $\langle (n_1, o_1), (n_2, o_2), ..., (n_x, o_x) \rangle$ such that $\langle n_1, n_2, ..., n_x \rangle \subset \langle n_1^u, n_2^u, ..., n_n^u \rangle$ and $o_1 = o_1^d$; or a chain of nodes-ids that correspond to a full match of u, that is, a chain of nodes $\langle (n_1, o_1), (n_2, o_2), ..., (n_n, o_n) \rangle$ such that $\langle n_1, n_2, ..., n_n \rangle = \langle n_1^u, n_2^u, ..., n_n^u \rangle$ and $o_1 = o_1^d$ and $o_n = o_n^d$.

Given a set of sub-traces $utraces(t, d(t), u)$, we define the set $utraces^*(t, d(t), u)$ as the set of sub-traces in $utraces(t, d(t), u)$ from which we remove the nodes that correspond to partial or full matches of the use u on objects instances different from the ones that appears in $d(t)$. In detail, let $utraces^*(t, d(t), u)$ be the set $(t_{i,x}^*, t_{i,y}^*, ...t_{i,z}^*)$ of sub-traces obtained removing from each $t_{i,j} \in utraces(t, d(t), u)$ all the chain of

nodes-ids $\langle(n_1, o_1), (n_2, o_2), ..., (n_x, o_y)\rangle$ such that $\langle n_1, n_2, ..., n_x\rangle \subset \langle n_1^u, n_2^u, ..., n_v^u\rangle$ and $o_1 \neq o_1^d$; and the chain of nodes-ids $\langle(n_1, o_1), (n_2, o_2), ..., (n_v, o_y)\rangle$ such that $\langle n_1, n_2, ..., n_v\rangle = \langle n_1^u, n_2^u, ..., n_v^u\rangle$ and $o_1 \neq o_1^d$ or $o_y \neq o_n^d$.

Thus the fitness of a test case with respect to a definition use pair (d, u) on trace t is defined as the fitness to reach the definition plus the maximum fitness to reach the use if the definition is not executed in t; or if the definition is reached, as the minimum fitness to reach the use as computed on the sub-traces of t that are both kill-free with respect to the definition and that interests the same object instances of the definition:

$$duFitness(d, u, t) = \begin{cases} \text{maxusefit} + contextualElementFitness(d, t) \\ \qquad \text{if } d \text{ is not covered by } t, \\ \min(contextualElementFitness(u, t_{i,j}) \\ \qquad | \; t_{i,j} \in utraces^*(d(t), t, u)) \\ \qquad \text{if } d \text{ is covered by } t \text{ by } d(t) \end{cases} \tag{6.5}$$

Where *maxusefit* is the maximum fitness value of $contextualElementFitness(u, t)$.

Chapter 7

Evaluation

We evaluated Dynamic Data Flow Testing by designing a prototype tool for the Java programming language that we refer to DynaFlow, and by experimenting with a set of open source programs. This chapter describes the experimental setting and discussed the evidence we collected. We show that the Dynamic Data Flow Testing strategy can indeed detect test objectives exploiting Dynamic Data Flow Analysis and generate new test cases to cover them. We compare the fault detection ability of Dynamic Data Flow Testing both with the initial the suite, and with the test suites generated randomly and for static data flow coverage. The experiments indicate that (1) the test cases identified with our approach can reveal failures that could go otherwise undetected with the initial test suite, (2) dynamic data flow testing is more effective than classic data flow testing, and (3) the effectiveness of the generated test suite depends more on the relevance of the identified test cases that on the size of the suite.

The empirical evaluation addresses the main research question: *"To what extent is it possible to enhance an initial test suite by exploiting dynamic data flow information?"*.

To answer this question, we implemented *DynaFlow*, a prototype implementation of *Dynamic Data Flow Testing* for Java programs, and executed a set of experiments on a benchmark of classes. We compare the test suites generated with *DynaFlow* with the original test suites, to evaluate the ability of *DynaFlow* to enhance the initial suite. We compare the *DynaFlow* test suites with the test suites generated with a state-of-art test generator based on static data flow analysis, to highlight the improvement of dynamic over classic data flow testing. Finally, we compare the *DynaFlow* test suites with large test suites generated randomly to verify that the good results of *DynaFlow* do not depend on the size of the test suite. We compare the test suites in terms of their ability to reveal failures, and we approximate this ability as the number of mutants killed by the test suites. The more mutants a test suite can kill, the more effective the test suite is in revealing the corresponding faults.

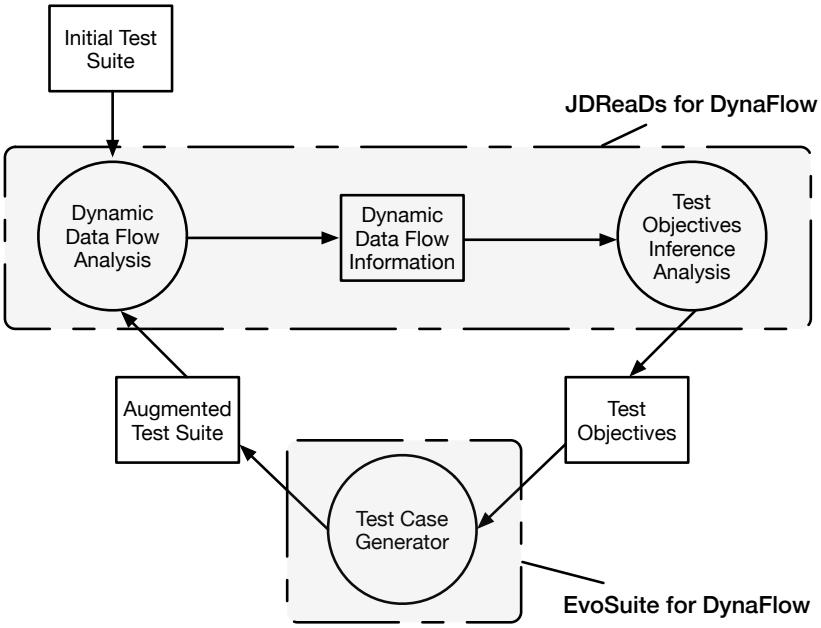


Figure 7.1. The architecture of *DynaFlow* and its relation with the workflow of Figure 6.1

Section 7.1 briefly describes the implementation details of *DynaFlow*, Section 7.2 presents the design of the experiment, Section 7.3 analyzes the obtained result, Section 7.4 discusses our findings, and Section 7.5 reports the threats to the validity of the study.

7.1 Prototype Implementation

We designed a prototype implementation of the *Dynamic Data Flow Testing* technique for Java programs, which we called *DynaFlow*. *DynaFlow* takes a JUnit test suite and a set of classes under test in input, and produces an enhanced test suite exploiting dynamic data flow information. *DynaFlow* computes also the data flow information — including observed and inferred definition use pairs — for each step of the execution of the technique.

DynaFlow is implemented relying on a custom version of *JDReaDs* for performing the dynamic analysis *and* the inference of the test objectives, and on a custom version of the EvoSuite tool for generating the test cases. We introduced *JDReaDs* in Section 4.6, while EvoSuite[1] is a mature search based testing tool for Java, originally developed by Fraser and Arcuri [FA11a, FA13]. EvoSuite generates test cases using genetic algorithms. It iteratively evolves an initial population of test suites by applying crossover

[1]http://www.evosuite.org

and mutation operators to the individuals. A distinctive characteristic of EvoSuite is the possibility of targeting the optimization of a whole test suite: While targeting the optimization of a whole suite, EvoSuite aims to optimise sets of test cases towards covering all goals described by a coverage criterion.

Figure 7.1 show the architecture of *DynaFlow* and its relation with the workflow of *Dynamic Data Flow Testing* described in Section 6.2. *DynaFlow* includes *JDReaDs* and EvoSuite. *DynaFlow* uses *JDReaDs* to (1) select the "relevant" definition use pairs as discussed in Section 6.3, and (2) perform the static matching between *defs@exit* and *uses@entry* to produce the test objectives. *JDReaDs* for *DynaFlow* serializes the data flow information, including the inferred test objetives, in a file, which is passed as input to our custom implementation of EvoSuite.

We extended EvoSuite for *DynaFlow* to target the test objectives passed by the analysis. We modified EvoSuite to process the test objectives passed by *JDReaDs*, we implemented the fitness function that we defined in Section 6.5, and we implemented a mechanism to keep track of data flow coverage, which records active definitions and kill events at runtime. We implemented many of these features on top of a version of EvoSuite that targets static data flow test objectives, which we developed in previous work [VMGF13].

In particular, we extended EvoSuite to encode *DynaFlow* definition use pairs as pairs of nodes of the interprocedural control flow graph, and to instrument the code locations corresponding to the nodes to record the execution of definitions and uses. EvoSuite has an option to insert the necessary instrumentation for tracking the invocation context of every definition and use event: We exploited this option to properly distinguish definitions and uses on their context. We also added a runtime analysis modules to register active definitions, kill events and the coverage of pairs.

On top of these modifications, we implemented the fitness of Equation 6.5 for targeting contextual definition use pairs. Our fitness exploits the modules of EvoSuite that computes the approach level and the branch distance given a target node and an execution trace. We also rely on the standard implementation of the crossover and mutation operators provided by the tool.

Our original fitness function targets the optimization of single tests for executing individual objectives. In EvoSuite for *DynaFlow* we also implemented a whole suite of fitness functions to exploit the performance and optimization of the tool. The resulting fitness function is the sum of the individual fitnesses of each goal, for each test in a test suite, as shows Equation 7.1, where S is the set of execution traces generated by the test suite, and P is the set of contextual definition use pairs to be used as test objectives.

$$\text{duSuiteFitness}(S) = \sum_{t_i \in S} \sum_{d_k, u_k \in P} duFitness(d_k, u_k, t_i) \tag{7.1}$$

We took advantage of the *archive* feature of evosuite to improve the performances of the approach [RVAF16]: EvoSuite archives the individuals (i.e., the test cases) that

satisfied a goal, and removes the satisfied goals from the set of test objectives yet to be covered. At the end of the execution, EVOSUITE returns a suite composed of all the individuals in the archive.

We performed all the experiments discussed in this section using EVOSUITE for *DynaFlow* configured to use the whole suite generation fitness function with the archive feature.

7.2 Experimental Setting

We evaluated *DynaFlow* on 30 Java classes extracted from some projects of the SF100 set of programs [FA12b]. The benchmark reflects the expected usage scenario of *DynaFlow*, which targets the testing of classes with complex state. We manually selected classes among the ones that both include one or more non-primitive fields and implement one or more methods that access or modify the value of such fields in a non trivial way.

We generated the initial suites with EVOSUITE for branch coverage, which is its default EVOSUITE configuration. This allows to limit the biases caused by the initial suite, and to intrinsically compare *DynaFlow* suites with test suites optimized for branch coverage.

Table 7.1 reports the relevant characteristics of the classes used in our experiments: the number of lines of code (*LOC*), the number of dependent classes as computed using the Dependency Finder tool[2] (*Reachable code – # classes*) and the sum of lines of code of the class under test and its dependent classes (*Reachable code – LOC*). The analysis domain of *DynaFlow* is the union of the class under test and its dependent classes, that is, the classes directly or indirectly called from the class under test. Thus the number of dependent classes and their LOCs indicates the size of the *DynaFlow* analysis domain. The last column reports the branch coverage obtained when executing the initial test suite generated with EVOSUITE for branch coverage.

For each subject class, we executed *DynaFlow* to enhance the initial test suite, with a maximum budget of three *DynaFlow* iterations. We generated two additional test suites for each class: a large suite generated randomly and a suite that covers the def-use pairs computed statically. We generated the large test suite using Randoop with a limit of 1000 test cases, and the static def-use test suite using EVOSUITE for static data flow testing [VMGF13].

We evaluated the effectiveness of the test suites as the amount of mutants killed when executing the suites. We use these data as a proxy measure of the amounts of failures that can be revealed by the test suites. We generated mutants for the classes under test and their dependent classes with the PiTest mutation analysis tool,[3] a tool commonly adopted in recent related work [IH14, GJG14].

[2] `http://depfind.sourceforge.net`

[3] http://pitest.org

Table 7.1. Benchmark classes

Class	Project	LOC	Reachable code		Branch Coverage
			# classes	LOC	
AttributeRegistry	freemind	371	68	15995	76%
ColorImage	jiggler	1273	43	11758	11%
HandballModel	jhandballmoves	814	110	9979	11%
Robot	at-robots2-j	417	109	7411	29%
ComplexImage	jiggler	872	20	7153	50%
MealList	caloriecount	388	30	4685	97%
GameState	gangup	472	23	3813	95%
DecadalModel	corina	295	15	3680	70%
BattleStatistics	twfbplayer	578	29	3665	83%
Hero	dsachat	349	19	3576	57%
FoodList	caloriecount	146	24	3464	100%
MoveEvent	jhandballmoves	247	22	3076	75%
Knight	feudalismgame	393	16	2471	34%
Challenge	dsachat	309	7	2370	22%
FieldInfo	fixsuite	367	14	2228	100%
Formation	gfarcegestionfa	104	13	2195	100%
ComponentInfo	fixsuite	276	13	2119	100%
ObjectChartData Model	jopenchart	252	10	1922	100%
ProductDetails	a4j	518	19	1783	90%
ProductInfo	a4j	96	7	1338	100%
HardwareBus	at-robots2-j	144	13	1310	100%
HL7FieldImpl	openhre	172	10	1078	95%
GroupInfo	fixsuite	179	6	1021	80%
StackedChartData ModelConstraints	jopenchart	164	5	910	69%
EmailFacadeImpl	bpmail	414	13	854	7%
MemoryRegion	at-robots2-j	48	7	839	100%
ListChannel	caloriecount	78	14	728	62%
InventorySavePet	petsoar	91	6	496	88%
HL7TableImpl	openhre	60	5	396	100%
DefaultDataSet	jopenchart	123	2	187	75%

7.3 Experimental Results

Tables 7.2 and 7.3 report the results of our experiments. For each subject class (first column of both tables), Table 7.2 indicates the number of definition use pairs (*number of test objectives*) identified by *DynaFlow* and the *number of mutants killed* by the generated test suites. Table 7.3 reports the size (*number of test cases*) of the generated test suites.

In Table 7.2, the columns *number of test objectives* report the number of definition use pairs executed (1) by the initial test suite generated with EvoSuite for branch coverage (column *Initially covered*), (2) after three iterations of *DynaFlow* (column *Covered* DynaFlow) and (3) not yet covered after the third iteration. In our experiment the enhanced test suite always executes a higher number of definition use pairs than the initial suite. In total, the enhanced test suites execute 44% more pairs than the initial suites.

We report the increase distribution per class in the boxplots in Figure 7.2. In particular, we observe a median increase of 83%, with the first quartile being 27%, the third

Table 7.2. Experimental Results: Number of Test Objectives and Killed Mutants

Class	Number of test objectives			Number of killed mutants			
	Initially covered	Covered *DynaFlow*	Not covered	EvoSuite -branch	EvoSuite -data flow	Randoop	*DynaFlow*
AttributeRegistry	331	375	129	125	143	0	145
ColorImage	230	441	1439	165	107	34	270
HandballModel	98	99	160	395	264	213	423
Robot	91	188	129	43	32	60	63
ComplexImage	105	185	354	126	75	11	165
MealList	23	108	101	125	20	49	162
GameState	2133	2203	198	150	154	57	203
DecadalModel	198	556	381	66	79	34	69
BattleStatistics	317	380	95	294	175	93	334
Hero	44	66	20	166	113	0	177
FoodList	14	64	34	35	28	14	51
MoveEvent	74	194	90	86	15	0	102
Knight	76	89	106	134	99	135	166
Challenge	54	103	66	89	59	5	132
FieldInfo	13	50	24	36	40	31	56
Formation	10	49	8	17	23	11	18
ComponentInfo	42	123	31	64	77	55	69
ObjectChartData Model	34	38	87	105	0	55	141
ProductDetails	108	228	30	314	294	295	498
ProductInfo	113	281	15	246	61	215	289
HardwareBus	45	57	10	27	21	14	29
HL7FieldImpl	24	46	6	64	74	0	75
GroupInfo	32	113	40	85	26	81	98
StackedChartData ModelConstraints	102	179	136	142	7	110	209
EmailFacadeImpl	34	56	3	41	44	0	61
MemoryRegion	44	54	9	53	38	0	58
ListChannel	23	44	0	9	27	8	51
InventorySavePet	12	20	3	16	5	24	31
HL7TableImpl	13	19	0	34	34	35	38
DefaultDataSet	9	14	2	12	25	25	23
Sum	4446	6422	3706	3264	2159	1664	4206

All the experiments have been repeated 6 times with different seeds to govern the random mechanisms of the test generation. The 90th confidence interval was 5.5% of the measured value on the average, with a maximum of 11.5%. The 95th confidence interval was 6.5% of the measured value on the average, with a maximum of 13.7%.

162%, and the minimum and maximum increments 1% and 391%, respectively. Thus, the iterative process of *DynaFlow* can both identify new test objectives and generate new test cases to execute them.

The test objectives that are identified but not executed by *DynaFlow* amount to 37% of all the identified pairs. Some of these test objectives are identified at the third iteration step for which no test generation attempt has been taken, others are not executed due to known limitations of EvoSuite [FA12b], yet others are possibly infeasible.

The last four columns of Table 7.2 (columns *Number of killed mutants*) shows the number of mutants killed by the generated test suites. The *DynaFlow* enhanced test

Table 7.3. Experimental Results: Number of Test Cases

Class	Number of test cases			
	EvoSuite -branch	EvoSuite -data flow	Randoop	*DynaFlow*
AttributeRegistry	19	19	1000	48
ColorImage	19	33	1000	32
HandballModel	36	63	1000	60
Robot	20	60	1000	41
ComplexImage	16	23	1000	37
MealList	21	11	1000	61
GameState	7	53	1000	51
DecadalModel	23	38	1000	72
BattleStatistics	20	34	1000	62
Hero	14	24	1000	80
FoodList	16	50	1000	46
MoveEvent	20	48	1000	120
Knight	23	65	1000	201
Challenge	19	15	1000	58
FieldInfo	3	15	1000	18
Formation	5	37	1000	46
ComponentInfo	8	34	1000	46
ObjectChartData Model	11	39	1000	76
ProductDetails	13	58	1000	105
ProductInfo	4	8	1000	67
HardwareBus	5	15	1000	16
HL7FieldImpl	8	38	1000	28
GroupInfo	9	34	1000	53
StackedChartData ModelConstraints	10	4	1000	39
EmailFacadeImpl	12	12	1000	25
MemoryRegion	7	9	1000	48
ListChannel	1	10	1000	26
InventorySavePet	8	33	1000	21
HL7TableImpl	5	7	1000	13
DefaultDataSet	9	20	1000	23
Sum	391	909	30000	1619

All the experiments have been repeated 6 times with different seeds to govern the random mechanisms of the test generation. The 90th confidence interval was 5.5% of the measured value on the average, with a maximum of 11.5%. The 95th confidence interval was 6.5% of the measured value on the average, with a maximum of 13.7%.

suite (column *DynaFlow*) consistently kills more mutants than the test suite generated by EvoSuite for branch coverage (column EvoSuite-*branch*) that was used to seed *DynaFlow*. In total, *DynaFlow* kills 942 (29%) more mutants than the original test suite.

Figure 7.3 reports the increase distribution per class. We observe a median increment of 27%, with the first quartile being 12%, the third 48%, and the minimum and maximum increments 5% and 467%, respectively. These results indicate that the new test cases generated by exploiting the dynamic data flow information effectively

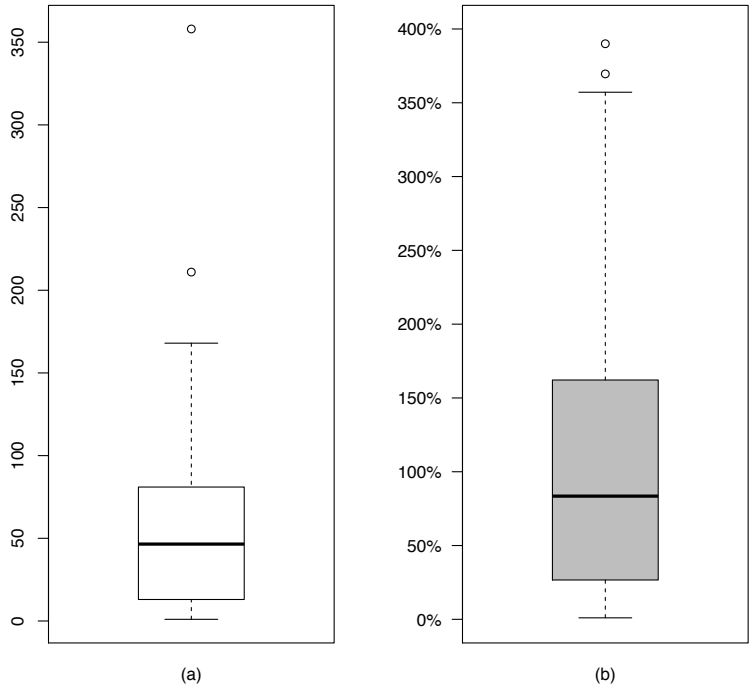


Figure 7.2. Distribution of the increase of executed definition use pair by the enhanced test suites: (a) distribution of increase per class, (b) distribution of the percentage increase per class.

enhance the initial test suite in terms of fault detection.

Column EVOSUITE-*data flow* reports the mutants killed by test suites generated with EVOSUITE for static data flow coverage. *DynaFlow*'s test suites kills more mutants that the ones generated for static data flow coverage for most classes, with an average of twice as many killed mutants. Per class, we observe a median increment of number of mutants killed with *DynaFlow* over EVOSUITE for static data flow coverage of 69%, with the first quartile being 32%, and the third 152%. This result indicates that dynamic data flow analysis selects better test objectives than static data flow analysis. Counter-intuitively the data in the table indicate that EVOSUITE-branch slightly outperforms EVOSUITE-data flow, differently from the results we obtained in previous experiments [VMGF13]. This may be caused by the fact that this benchmark includes complex classes that are difficult to analyze with static analysis.

Table 7.3 (columns *Number of test cases*) reports the amount of test cases generated with the different approaches, and indicates that *DynaFlow* generates larger test

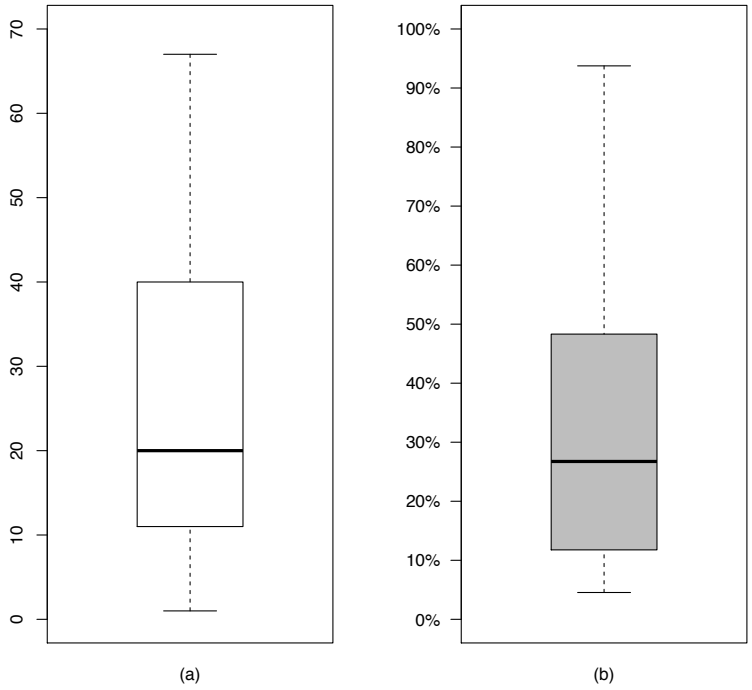


Figure 7.3. Distribution of the increase of killed mutants by the enhanced test suites. (a) reports the distribution of the increase per class, while (b) the distribution of the percentage increase per class.

suites than the other approaches, thus raising the question of the importance of the size of the test suites. To check the impact of the site size on the test effectiveness, we compared the results of *DynaFlow* test suites with the results of very large test suites randomly generated with Randoop. The results of the effectiveness of these large suites are reported in the column *Randoop* of Table 7.2.

We can see that Randoop is much less effective than any of the other techniques, despite the very large size of the test suites generated with Randoop (1000 test cases for each suite). In total, the *DynaFlow* test suites kill 2.5 times more mutants than Randoop test suites. We observe that Randoop kills zero or few mutants for some classes. This happens for classes that require a complex initialization or a complex interaction with other classes to be thoroughly exercised. This indicates that the larger amount to mutants killed with *DynaFlow* test suites does not depend on the size of the suite, but on the quality of the test cases.

The dynamic analysis component of *DynaFlow* generated the test objectives exe-

Listing 7.1. Example of mutants killed only by *DynaFlow* test cases

```
1  class BattleStatistics {
2      private AllCombatantSidesCounter swaps = new AllCombatantSidesCounter();
3      public int totalSwaps(final CombatantSide side) {
4          return swaps.getSideValue(side); // Mutant: return 0;
5      }
6  }
7
8  class AllCombatantSidesCounter {
9      private Map<CombatantSide, Counter> perSideCounters = new HashMap<>();
10     void incrementSide(final CombatantSide side) {
11         if (side == null) {
12             for (final Counter counter : perSideCounters.values()) {
13                 counter.increment();
14             }
15         } else {
16             perSideCounters.get(side).increment();
17         }
18     }
19
20     public int getSideValue(CombatantSide side) {
21         int sum = 0;
22         if (side == null) {
23             for (Counter counter : perSideCounters.values()) {
24                 sum += counter.getValue();
25             }
26             return sum; // Mutant: return 0;
27         }
28         return perSideCounters.get(side).getValue();
29         // Mutant: return 0;
30     }
31 }
```

cuting the test cases of each class within 10 seconds in most of the cases, and within 45 seconds in few worst cases. The test generation step of *DynaFlow* had a maximum budget of 5 minutes per iteration, with performances fully acceptable in the context of automated test case generation.

7.4 Discussion

The results presented in the former sections indicate that *Dynamic Data Flow Testing* can augment an initial test suite with test cases that reveal faults that would otherwise go undetected, and thus positively answer our research question. To support our hypothesis that dynamic data flow analysis can identify interactions among methods that are difficult to find otherwise, we manually inspected the mutants killed only by the *DynaFlow* test cases and investigated their nature.

Indeed, we found out that most of the mutants killed only by *DynaFlow* test cases are characterised by particular combinations of method calls that are triggered by test

objectives that involve interactions through dynamically instantiated state variables. These interactions require the methods of the class under test to be executed in multiple invocation contexts and with different values of the (nested) class state variables. *DynaFlow* can identify such interactions dynamically, and generate test cases that kill the corresponding mutants, while the other approaches cannot identify these interactions and thus fail to generate the required test cases.

Listing 7.1 shows two classes and a set of mutants that are killed only by *DynaFlow* test cases. The listing shows the class under test `BattleStatistics` that owns a reference to class `AllCombatantSidesCounter` through its `swap` field. The method `totalSwaps()` in class `BattleStatistics` invokes the method `getSideValue()` in class `AllCombatantSidesCounter`.

Three mutants, obtained with the PiTest operator that substitutes the return statements in method `totalSwaps()` and `getSideValue()` with `return 0`, are indicated with comments in the figure. In our experiments, only *DynaFlow* test cases kill these three mutants. This happens because *DynaFlow* targets the interactions on the subfields of variable `swap` that are nested in the state of the class under test: *DynaFlow* requires the two uses of the state variable `BattleStatistics.swaps.perSideCounters` at lines 23 and 28 to be coupled with observed definitions of that state variable. For example, *DynaFlow* explicitly requires the use at line 23 to be executed after the definition within the method `incrementSide()` at line 16 that modifies the content of the map `perSideCounter`. This increases the chances of returning a `sum` different from 0 and exposing the mutant at line 26. None of the other approaches identifies test objectives that capture this combinations of method calls, and thus may not generate test cases that exercise such combinations.

7.5 Threats to Validity

We conclude the presentation of the experimental validation results by acknowledging the threats that may limit the validity of our experimental results, and briefly discusses the countermeasures that we adopted to mitigate such threats.

Threats to internal validity may derive from the evaluation setting and execution, and depend on the *DynaFlow* prototype and EvoSuite. Although we tested carefully the *DynaFlow* prototype, we cannot exclude the presence of faults. We are aware that the limitations of EvoSuite could prevent the execution of feasible definition use pairs, and thus our results may be a pessimistic approximation of the effectiveness of *DynaFlow*. Current work on improving EvoSuite could reduce this limitation, potentially increase the effectiveness of our prototype, and improve the accuracy of the results. The randomness nature of EvoSuite could also impact on the results, to reduce such impact we repeated each experiment multiple times.

The selection of the initial test suite may also affect the results. We selected the initial suite automatically to avoid biases due to manual generation, and we used Evo-

SUITE for branch coverage, being a common tool and a common setting for the tool. We conducted some experiments with different initial test suites and we did not reveal major differences in the obtained results. Thus we decided to perform the main core of the experiments with initial test suites generated automatically.

Threats to construct validity involve how we measure the effectiveness of *DynaFlow*. We approximate the fault detection capability as the amount of killed mutants that we generated with the PiTest tool. Approximating fault detection in terms of killed mutants is common practice in current research projects and is widely accepted as a reasonable proxy measure. PiTest has been adopted in recent work [IH14, GJG14], and we modified it to prevent undesired approximations of the results. We plan to repeat our experiments with different mutation analysis tools and with real program faults.

Threats to external validity may derive from the selection of the benchmark classes. We mitigated this thread by randomly selecting classes from a well known corpus (SF100).

Chapter 8

Related Work

Dynamic Data Flow Testing relates to Data Flow Analysis and Testing, Automated Test Case Generation and Dynamic Data Analysis. We surveyed data flow analysis and testing in Chapters 2 and 3. This chapter describes the state of the art in Automated Test Case Generation and in Dynamic Data Analysis, focusing on techniques for object-oriented systems.

In this thesis we present a data flow test case generation approach that benefits from a new dynamic application of data flow analysis. Our work relates to data flow analysis and testing techniques, which we discussed in Chapters 2 and 3, but also to techniques for automatically generating test cases for object-oriented systems, and techniques for dynamic data analysis. In this section we discuss the main automated test case generation techniques, and briefly survey the work on dynamic data analysis.

8.1 Automated Test Case Generation

In the last decades, researchers have been working extensively towards the automation of test case generation and selection [Ber07]. By automating the tedious and error prone activity of selecting a set of test cases, researchers aims to reduce the cost of software quality assurance, which constitutes a large part of the cost of modern software development.

In this section, we survey the most important results that have been achieved in this area and discuss the main challenges that still need to be addressed. We focus the discussion on the techniques for testing object-oriented systems which employs static and dynamic software analysis to steer the test generation, for they are more closely related to *Dynamic Data Flow Testing*. We discuss existing work on automated test case generation classifying them on their background techniques: Model Based Testing, Random Testing, Symbolic Execution, Search-based Testing and hybrid approaches [ABC+13].

8.1.1 Model-Based Testing

Model Based Testing techniques use models of the expected behavior of software to systematically test an application. These techniques exploit models that describe the structure of the input space to automatically or semi-automatically generate test inputs and to define adequacy criteria, and rely on the discrepancies between the models and the program behavior to identify boundaries and error cases.

Model Based Testing techniques exploit a large variety of models that can represent the system behavior and structure at different levels of abstraction. Formal models provide a formal description of the system, and can typically be used to define fully automated test generation techniques. Semi-formal models require some human judgement to generate test cases and map the model entities on the actual software. Recent advances in the automatic generation of behavioral models from program executions has opened a new way to fully automate model based testing techniques.

A considerable amount of work targets the generation of test inputs, the selection of test scenarios, and the generation of test oracles from formal models and specification [HBB+09]. Finite State Machines (FSM) and Labelled Transition Systems (LTS) have been used to model the software behavior, and then define adequacy criteria on the paths represented in the models to select test scenarios [PY07, Tre08]. FSM can be extended with information on the input domain to enable automated test input generation and to exclude infeasible interactions [NMT12, HKU02].

Some FSM-based techniques are closely related to *Dynamic Data Flow Testing*. Hong et al. and Gallagher et al. define a Model Based Testing technique that combines FSM models and data flow testing to test the interactions within and between classes in object-oriented systems [HKC95, GOC06]. Hong et al. technique models the behavior of a single class as a finite state machine, transforms the representation into a data flow graph that explicitly identifies the definitions and uses of each state variable of the class, and applies data flow testing to produce test case specifications that can be used to test the class. Gallagher et al. extended their approach to target the testing of multiple classes, to exclude (some) infeasible paths from the models, and proposed to integrate an automated test input generator in the technique (however, for evaluating the technique they partially relied on manual testing to generate the test cases). With respect to these approaches, *Dynamic Data Flow Testing* does not rely on software specifications, directly addresses the dynamic constructs of object-oriented languages, and it is fully automated.

Other Model Based Testing approaches use formal and algebraic specification to both select test objectives and automatically generate test data [SC96, DF94]. To ease the applicability of formal specifications, which are typically complicate and expensive to apply, some work proposed user-friendly specifications suitable for automating test case generation [Mey92, GGJ+10]. Well-designed specifications suitable for automatic test generation are seldom available, and currently discouraged by popular development processes such as Agile ones.

A variety of techniques uses UML models to select test scenarios and inputs for testing functional and non-functional properties of software. For instance, model based testing approaches extract test scenarios from sequence chart [NZR10, DH11], statecharts [OA99], and use case diagram [KKP+08]. Recent UML-based techniques includes the testing of non-functional properties of software [TRLB+14].

8.1.2 Random Testing

Random testing approaches generate test inputs by randomly sampling the input space of the program under test. Random testing is completely unbiased, can easily be applied and generates a large number of test cases. However, due to the size and complexity of the input space of an application, is unlikely for pure-random testing to be effective in revealing program faults. Recent advances in random testing research partially solved this problem by complementing random approaches with dynamic and static analysis techniques that drive the random search towards either interesting or never observed areas of the input space. Feedback directed random techniques can cover a significant part of the program under test with a minimum cost.

Random approaches have been proven to be a viable solution for automatically generating a large number of test cases with a minimum generation cost [CS04, GKS05, PLEB07, CLOM08]. The advantages of Random Testing are that it neither requires any domain and specific knowledge to generate test cases, nor employs computationally expensive techniques like Symbolic Execution. However, pure-random testing approaches seldom achieve high structural coverage, and typically perform a shallow exploration of the application control flow. This is because, without any restrictions on the input domain, the probability of executing a specific code element is very low [OP97]. Furthermore, random testing tends to generate a huge amount of invalid test cases, that either do not compile, or execute the program in unintended ways raising unwanted exceptions.

Recent research work mitigates these issues by combining random testing with some mechanisms to improve the smartness of the random search.

Feedback-directed random testing techniques employ a feedback mechanism to direct the test generation towards more effective test cases [PLEB07]. These approaches incrementally generate test cases, executing them at each step to identify valid and invalid sequences of statements. During the generation of test cases, if a test case contains a sequence of statements that leads to a compilation error or an unexpected exception, the sequence is discarded and substituted with a new (randomly generated) one — until a maximum test length or a time budget is reached.

The original feedback random testing approaches have been extended with dynamic and static analysis to further help the random search [AEK+06, TXT+09, ZZLX10, Zha11, MAZ+15]. Palulu improves the search using dynamic analysis to infer a call sequence model from a sample execution, and then follows that model to create test cases [AEK+06]. RecGen and MSeqGen rely on statically built data flow models to

identify pairs of method invocations that may interact on the same variables, and then weight the random search to select these pairs with higher probability [TXT+09, ZZLX10].

Palus and GRT uses both dynamic and static analysis to improve the random generation [Zha11, MAZ+15]. Both uses static analysis to identify method dependences that derive from accesses to common fields. Palus uses a dynamic step to captures sequences of method calls from the system execution traces, and then generalizes them in a call sequence model to be extended randomly [Zha11]. GRT refines and integrate the information computed by static analysis with a set of more detailed information observed dynamically while executing the program [MAZ+15]. *Dynamic Data Flow Testing* shares with these techniques test objectives that identify interactions between methods, and the idea to help the search relying on dynamic information, but considers a novel class of test objectives that are difficult to compute or address with any other existing technique.

Adaptive random testing (ART) techniques follow a different rationale to improve the effectiveness of random testing. Empirical studies have shown that failure-causing inputs tend to form contiguous regions in the input space [CCMY96]. Building on this, ART techniques maximize the distance between test inputs according to a certain metric of the state space, aiming at generating test cases that evenly spread across the input domain [CLM05]. Different distance metrics have been investigated in literature, targeting both single input programs and object-oriented systems [CLOM08]. A further improvement of the approach exploits method preconditions to further guide object selection [WGMO10].

Despite the amount of work on the topic, Arcuri and Briand demonstrated that, given a fixed time budget, the overhead caused by the computation of the distance metric makes ART less effective than pure-random approaches, questioning its practical effectiveness [AB11]. Another general disadvantage of random testing is that random generated test cases are in general particularly difficult to interpret, consequently a considerable effort is required to understand them and to write meaningful oracles.

8.1.3 Symbolic Execution

Test generation techniques that rely on symbolic execution execute the program with symbolic input values, and compute expressions on the input values that indicate the conditions to execute a given path [Kin76]. Symbolic executors rely on automated theorem provers to generate test cases that cover the corresponding program paths. Unfortunately the number of paths in a program is typically infinite and exhaustive symbolic execution is impossible. Testing techniques based on symbolic execution can explore the execution space systematically (up to a finite amount) and can thus address rare corner cases, but suffers from the difficulty of automatically solving complex path conditions, as the ones often generated while symbolically executing non trivial programs. Moreover, despite the improvements in automated theorem proving obtained in

the last decades, which allowed the implementation of robust symbolic execution tool (e.g. KLEE [CDE08] and JPF [KPV03]), some program constructs cannot be analyzed precisely and efficiently. Nevertheless, Symbolic Execution is still a very active research field and found some important industrial applications [CS13].

The original approaches to symbolic execution were proposed in the 70s and 80s and address sequential programs with simple inputs of a limited set of primitive types [BEL75, Cla76, Kin76, RHC76]. This is mostly because of limitations of automated theorem provers in dealing with complex path conditions and some program types (for instance, floating point arithmetics). Recent advances of theorem provers and the rapid increase of performances of modern hardware allowed the definition of new techniques which scale to more complex programs, and in particular to object-oriented systems.

Generalized Symbolic Execution (GSE) improved over the original approaches allowing the analysis of programs with complex data structures as inputs and a multi procedural and multi thread structure [KPV03]. GSE exploits lazy initialization to handle recursive data structures and methods with multiple parameters, and employs a theorem prover whose built-in capabilities allow handling non determinism such as multi-threading. GSE extends Symbolic Execution and Model Checking to Java, and has been developed within the Java PathFinder (JPF) model checker and symbolic executor (JPF–SE) [VPK04, APV07]. Other examples of symbolic techniques applicable on data structures in object-oriented code include *Symstra*, JBSE, Bogor/Kiasan, and *Symclat* [dPX$^+$06, XMSN05, DLR06, BDP13]. However, despite the improvements in automated theorem proving, some program constructs still cannot be analyzed precisely and efficiently.

An efficient way to overcome part of the limitations of symbolic execution is offered by Dynamic Symbolic Execution (DSE). DSE interweaves concrete and symbolic execution: DSE starts by executing a random input and generating the path condition for the executed path. Then, it negates a branch condition of the path constraint collected on the concrete execution path, and submits the new formula to a solver to obtain a new input that characterize test cases that both reach not-yet-covered branches and discover new paths. DSE also replaces symbolic values with concrete ones every time the theorem prover cannot deal with them. This allows DSE to partially overcome the limitations of classic symbolic execution, at the expense of some precision [TdH08]. DSE strategies for object-oriented systems have been implemented in tools like jCUTE (Java) and Pex (.NET) [SA06, TdH08]. *Dynamic Data Flow Testing* shares with DSE the idea of alternating dynamic and static analysis to use observed information to overcome the limitations of pure static approaches.

These techniques typically target the exploration of the path structure of the methods, but are not driven by any inter-procedural test objectives that explicitly identify the interesting method sequences, which are selected systematically up to a certain length. Some techniques rely on static and dynamic analysis to select interesting method sequences to execute with symbolic inputs. Thummalapenta et al. proposed MSeqGen

that statically mines sequences of method calls from the source code and use PEX to instantiate complex objects [TXT+09]. Thummalapenta et al. proposed Seeker that encodes the class state as a set of conditional branches, and synthesizes method sequences by combining static and dynamic analyses to execute the target class state [TXT+11]. The test generation approach of Martena et al. relies on static data flow analysis to derive a set of interesting method sequences to be symbolically analyzed [MOP02].

Su et al. propose symbolic techniques to satisfy data flow relations within methods or procedures [SFP+15]. No other testing technique based on symbolic execution directly targets the inter procedural data flow relation which occurs between methods.

8.1.4 Search Based Testing

Search based testing approaches exploit *meta-heuristic* search algorithms to generate test cases, casting the problem of test generation as a search problem. Meta-heuristic search techniques define heuristics to find *sufficiently good* solutions to optimization problems which are (typically) hard or impossible to solve in polynomial time. Search based testing approaches use a fitness function to estimate how well individuals fit an optimization target, and select the individuals to be reproduced according to their fitness. By varying the fitness function, search based testing techniques can target different optimizations. Several Search Based Testing approaches have been defined in literature to generate test cases that satisfy different structural criteria, and to target both functional and non functional properties [McM04]. Although search based approaches do not guarantee to obtain the optimal solution, they have been proven to be very effective and scalable in practice [FA12b].

Adapting a meta heuristic search to a specific testing problem requires a set of key decisions: (i) encoding test cases so that they can be manipulated by the algorithm, (ii) selecting a suitable meta-heuristic algorithm for the search, (iii) tuning the meta-heuristic for the specific problem, (iv) defininining a fitness function to guide the search towards the test objectives. The literature contains a wide variety of approaches that propose different solutions for each of these decisions.

The first requirement to cast the problem of test generation as a search one is the definition of an encoding of the test cases. For many applications, the tests can be represented simply using an input vector, represented either with real values or string of binary digits [McM04]. Special encodes have been proposed for representing test cases of object-oriented systems: In this case, tests are generally represented by linear sequences of method calls [Ton04].

The second element is the choice of the meta-heuristic search algorithm employed for the search. Different algorithms have been used in search based testing techniques, including but not limited to Hill Climbing, Simulated Annealing and Genetic Algorithms [Kor90, TCMM98, PHP99].

Search based techniques can be directed to alternative test goals by defining appropriate fitness functions. To date researchers investigated the application of search based

testing to automatically generate test cases with different targets. Many approaches have been defined for covering specific program structures, in particular to achieve high structural coverage according to a given coverage criterion [Rop97, FA13, VMGF13]. Other techniques focus on exercising some specific program feature as described by a specification [BW03], are used to generate a particular usage scenario that fit or disprove a given property [BPS03], or were defined to test non functional properties of the application [ATF09], for example for performing stress testing [BLS05]. Recent applications of search based technique includes also the testing of programs through their GUI [GFZ12], and testing of databases schemas [KMW13].

In the remaining of this section we will discuss the main approaches for structural testing of object-oriented systems, which are the most related to *Dynamic Data Flow Testing*.

Tonella firstly defined a search based technique based on genetic algorithms to generate unit tests of classes [Ton04]. Tonella represents the test cases as sequences of statements for class instantiations, method invocations and instantiations of objects to use as parameters, and developed a prototype *eToc* to generate test cases that satisfy branch coverage for Java programs.

Tonella's work highlighted some of the problematics of applying meta-heuristic techniques to object-oriented systems. When encoding the individuals as unbounded method sequences, the search tends to select individuals with an increasing length. If the size of the test is not directly taken in account, the test cases length can abnormally grows over time [FA11b]. Moreover, the construction of the test cases is complicated by the complexity of object-oriented programming languages. The state based behavior of objects complicate the search, and constructs like polymorphism or objects which require complex parameters make the encoding of the test cases and the optimization particularly challenging.

To improve coverage of Java classes that require complex string inputs, Shahbaz et al. extended Tonella's *eToc* approach with Web searches to retrieve examples of valid inputs to use during the search [SMS12]. Miraz et al. improved over Tonella's approach by proposing a technique that explicitly considers also the internal state of objects to drive the search [BLM10]. Their *TestFul* technique implements an hybrid evolutionary algorithm that alternate global search to cover the internal states of objects, and local search to satisfy the branches of methods.

The *whole test generation* technique of Fraser and Arcuri is implemented in EvoSuite, and largely improves the applicability and effectiveness of search based test generation for object-oriented systems [FA13]. EvoSuite uses an evolutionary technique which evolves all the test cases in a test suite at the same time, using a fitness function that considers all the test goals simultaneously. This is opposed to classic search based testing techniques which evolve each test case individually towards the satisfaction of one coverage goal at time. The whole test generation approach allows a better optimization of the search budget and limits the problem of the abnormal grown

of the test cases, leading to better scalability and results.

EVOSUITE has been original proposed for branch coverage, and has been extended to support many different coverage criteria, including statement, (static) data flow coverage and mutation testing [VMGF13, FA14].

Several work extends EVOSUITE to improve its effectiveness for testing classes. EVOSUITE integrates a seeding strategy that extract "genetic material" from the application under test to constitute the initial population of test cases to evolve. This allows EVOSUITE to re-use domain specific information extracted from the source code to build the test cases, easing the generation of test cases that requires complex values [FA12a]. Local search, multi-objective fitness functions and dynamic symbolic execution were also experimented within the technique to improve the ability of EVOSUITE in satisfying corner cases and test complex classes [GFA13, FAM15, RCV+15]. Finally, EVOSUITE integrate a strategy to generate regression oracles relying on mutation testing. It executes the generated test cases on original and mutated programs and calculates a reduced set of assertions that is sufficient to kill all the mutants [FZ12].

At the state of the art, EVOSUITE generates self-contained test cases for Java more effectively than any other automated techniques [FA13]. *Dynamic Data Flow Testing* relies on a customized version of EVOSUITE to generate test cases.

Hybrid Approaches

Hybrid techniques exploit the synergies between the different test case generation approaches to obtain better results and performances. Random testing, search based techniques and symbolic execution are largely complementary in their strengths: Random testing and search based techniques scale better, have good performance, and can quickly identify test cases that execute a significant part of the application behavior. However, they are not well suited to cover both corner cases and difficult-to-execute paths. Dually, Symbolic Execution can generate more precise and focused tests, but with high cost enhanced by the use of constraint solvers.

Hybrid techniques build on the observation that randomized and symbolic techniques can benefit from each other when combined together. Randomized approaches can quickly explore the application execution space, and symbolic execution can then be used to reach the few cases left behind.

Majundar and Sen defined a technique that interleaves random and dynamic symbolic execution to explore deeply and widely the program state space [MS07]. Other techniques enhance search based testing embedding DSE in the search algorithm itself. Baars et al. use DSE in the fitness evaluation step for achieving branch coverage [BHH+11], while Malburg and Fraser proposed to use small steps of DSE as mutation operator within EVOSUITE [MF11].

Galeotti et al. observed that integrating DSE into a meta-heuristic algorithm affects the performances of the search at the point that higher coverage in a few corner cases may come at the price of lower coverage in the general case [GFA13]. Galeotti et al.

address this problem with an adaptive approach that employ DSE as operator only whether the coverage problem is determined to be suitable for DSE.

A different line of research combines randomized and symbolic techniques for testing object-oriented systems. This work exploits a randomised approach to derive interesting sequences of method calls, and relies on symbolic execution to determine the method inputs that maximize the coverage. Inkumsah and Xie use a genetic algorithm to search for desirable method sequences, and then use symbolic execution to generate method arguments [IX08]. Garg et al. propose to use a feedback random testing approach to generate the sequences [GIB+13].

Other hybrid techniques includes application of DSE and genetic algorithms for strong mutation testing [HJL11], and of search based algorithms to guide path exploration of a DSE technique [XTdHS09].

8.2 Dynamic Data Analysis

With dynamic data analysis we refer to program analysis techniques that analyze data accesses and information flow dynamically traced during the program execution. This section discusses the approaches more related to *Dynamic Data Flow Testing* and *Dynamic Data Flow Analysis*, including both standard dynamic analysis approaches, and ad hoc solutions developed within more general testing and analysis techniques.

One well known dynamic data analysis problem involves the identification of the origin of data during execution. *Taint analysis* tracks the information flow of the input data. Selected inputs of the applications are labeled (or *tainted*) with a tag, and at runtime these labels are propagated such that any new values derived from a tagged value is marked with a tag as well. Taint analysis inspects the tags to determine whether a value (or an object) derives from a tainted input or not. Traditional implementations of taint tracking systems have been defined for different programming languages, and rely on modifications of the operating system [EKV+05, ZBWKM06], modifications of the language interpreter [AS10, CF07, EGC+10, GPT+11], access to source code [LC06, XBS06] or bytecode instrumentation [BK14].

Taint analysis founds a broad range of applications in program analysis and testing. Taint techniques have been used for privacy testing [EGC+10], fine-grained data security [AS10, CSL08, MPE+10, RPB+09], detection of code-injection attacks [HOM06, SAP+11] and enhanced debugging [GLG12].

Taint analysis approaches for privacy testing, data security and code injection attacks labels to data originating from untrusted sources and keep track of the propagation of the tainted data as the program executes, to detect at runtime when tainted data are used in illegal ways depending on the scope of the analysis. Other techniques improve debugging and slicing approaches by relying on taint information for identifying relevant subsets of input sources [KL88, Tip95]. In a context closer to *Dynamic Data Flow Testing*, some work uses taint analysis to improve dynamic symbolic execution for

test generation [GLR09, WWGZ11].

Other dynamic data analyses have been exploited for runtime verification, slicing and testing techniques.

Dynamic data flow analysis techniques has been proposed by Chen and Low, and Boujarwah et al. for runtime verification of C++ and Java programs respectively [CL95, BSAD00]. These techniques monitor data events during a program execution to detect data flow anomalies, such as usage of variables before their initialization, or killing definitions of values which were never used before the kill. The techniques have been implemented in tools for C++ and Java and used to find simple anomalies. They share the idea of *Dynamic Data Flow Analysis* of using runtime information to perform data flow analysis, but they focus on a different problem and employ a simpler strategies to analyze the program, without considering the nested state of objects.

Dynamically augmented program slicing techniques leverages data dependencies observed at runtime by dynamic data analysis that tracks definition, uses and definition use pairs [KL88, Tip95]. These dynamic analyses track the occurrence of variable reads and writes, at the intra-procedural level, and have not been extended to either inter-procedural reaching analysis or contextual data flow analysis like *DReaDs*.

Dynamic data analysis is used in the context of testing to synthesize unit test cases from system ones [SAPE05, ECDD06]. These approaches run system-wide tests, and synthetize unit test cases by suitably identifying and tailor components and behaviors. Dynamic data analysis is used to capture dynamic information about the fields that are read or written during the execution to decide which parts of the program states have to be extracted and reproduced in the automatically generated unit tests.

Lienhard et al. and Zheng et al. adapt dynamic data analysis to build memory graphs similar to the one implemented in *Dynamic Data Flow Testing* in order to support program comprehension and advanced debugging functionality [ZZ02, LGN08].

Chapter 9

Conclusions

This thesis presents *Dynamic Data Flow Testing*, a novel approach to generate inter-procedural test cases for object-oriented software systems. Several studies have showed the effectiveness of data flow testing for exercising the state based interactions between methods and objects, but the applicability of data flow testing is limited by the difficulty of static data flow analysis to deal with the dynamic features of software systems.

In this work, we propose *Dynamic Data Flow Analysis*, a dynamic implementation of data flow analysis that computes sound data flow information while executing a program. By comparing data flow information collected with classic static analysis with information observed dynamically, we empirical observed that the data flow information computed with classic analysis of the source code misses a lot of information that corresponds to relevant behaviors that shall be tested.

In view of these results, we propose *Dynamic Data Flow Testing*, which exploits *Dynamic Data Flow Analysis* to identify precise and relevant data flow test objectives for a program under test. *Dynamic Data Flow Testing* combines dynamic analysis, static reasoning and test case generation. It uses *Dynamic Data Flow Analysis* to compute precise data flow information for a program under test, and processes the dynamic information to infer yet-to-be-executed test objectives, which are finally used by a test generation approach to produce new test cases. The test cases generated by *Dynamic Data Flow Testing* augment a suite with new executions that exercise the complex state based interactions between objects.

We implemented *Dynamic Data Flow Testing* in the *DynaFlow* prototype tool that generates JUnit test cases for Java programs. Our experiments indicate that *DynaFlow* generates test cases that reveal failures that could go otherwise undetected with state of the art structural and data flow testing approaches.

This work could represent the first step towards a new testing paradigm, which exploits dynamic information observed at runtime to effectively exercise interesting states and interactions that could cause subtle failures of programs.

9.1 Contributions

The main contribution of this thesis is an approach for automated data flow testing that exploits dynamically computed information to steer the generation of test cases towards relevant method interactions. The approach takes advantage of a dynamic implementation of data flow analysis to identify precise data interactions to use as test objectives. By exploiting the synergies between dynamic analysis, static reasoning and test case generation, we propose a new model of structural testing, which iteratively discovers new test objectives while testing the application.

The thesis makes the following specific contributions:

- **It proposes a dynamic implementation of data flow analysis for object-oriented systems to compute sound data flow information of class state variables.** *Dynamic Data Flow Analysis* (*DReaDs*) computes reaching definitions and reachable uses of the class state variables dynamically on the program traces. *DReaDs* captures memory events at runtime, and maps them to the involved class state variables by exploiting a model of the references between objects in memory. This allows the analysis to resolve aliasing and pointer references, computing a sound data flow information.

- **It provides a quantitative analysis of the limits of the current data flow testing approaches based on static data flow analysis.** By comparing dynamic data flow analysis with static data flow analysis, we empirically shown that the under-approximated static analyses commonly used within data flow testing approaches compute a set of data flow relations that badly approximate the set of feasible data flow elements. This negatively affects the effectiveness of data flow testing techniques based on static analysis, and ground our idea of using dynamic analysis to steer the test generation.

- **It proposes *Dynamic Data Flow Testing*, a testing technique for systematically generating effective test cases for object-oriented systems leveraging dynamic data flow information computed by *Dynamic Data Flow Analysis*.** *Dynamic Data Flow Testing* re-thinks data flow testing, as it defines a new paradigm that employs dynamic analysis to identify the test objectives for test case generation and selection. The technique exploits the possible interplays between dynamic analysis, static reasoning and automated test case generation to test the state based interactions of object-oriented systems.

- **It presents empirical evidence of the effectiveness of the proposed approach.** The *Dynamic Data Flow Testing* approach was implemented in the prototype tool *DynaFlow* for generating test cases for Java programs. *DynaFlow* has been used to generate test suites on several test subjects. The experiments validate the effectiveness of the *Dynamic Data Flow Testing* approach with respect to both

structural testing approaches based on the coverage of conditionals, and classic data flow testing techniques based on static analysis.

9.2 Future Directions

The work presented in this thesis opens several research problems. *Dynamic Data Flow Testing* constitutes the first step towards exploiting dynamic analysis techniques for generating test cases. Moreover, the *Dynamic Data Flow Analysis* technique could be applied to other problems beside data flow testing. We foresee the following future research directions:

- **Further exploiting the synergies between *Dynamic Data Flow Analysis* and automated test case generation.** The current implementation of *Dynamic Data Flow Testing* exploits *Dynamic Data Flow Analysis* to identify the test objectives and steer the test generation. Additional information collected using *Dynamic Data Flow Analysis* could further improve the automated test case generation. For instance, *Dynamic Data Flow Analysis* can report detailed coverage information of basic blocks and combinations of methods, identifying which blocks and combination of methods are not executed yet, and thus require more attention from the test generator. *Dynamic Data Flow Analysis* can also detect which definitions were never paired to any use during the executions, and used to prioritize the goals for test generation.
- **Extending *Dynamic Data Flow Testing* to tackle the oracle generation problem.** The current implementation of *Dynamic Data Flow Testing* generates test cases that relies on implicit oracles to expose faults. However, the information collected with *Dynamic Data Flow Analysis* could be exploited to either help testers in designing an effective test oracle, or automatically generating regression ones. We believe that data flow information can be useful for generating test oracles, and that it is interesting to understand how it can be exploited to tackle the problem of automated oracles generation.
- **Applying the *Dynamic Data Flow Testing* technique and testing model to dynamic-typed programming languages.** The problems of static analysis witnessed in this thesis are exacerbated in the context of programming languages that relies on a full dynamic typing mechanism. For instance, Python is more and more used to develop complex object-oriented applications with an very high degree of dynamism. These application are generally difficult to test systematically, since their behavior strongly depends on their runtime. Since *Dynamic Data Flow Testing* is particularly effective on programs which behavior is strongly dynamic, applying *Dynamic Data Flow Testing* on dynamically typed programming languages could ease the testing of these systems.

Bibliography

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275. ACM, 2011.

[ABC+13] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[AEK+06] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *Workshop on Model-Based Testing and Object-Oriented Systems*, M-TOOS '06, 2006.

[APV07] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '07, pages 134–138. Springer, 2007.

[AS10] Mohammad Reza Azadmanesh and Mohsen Sharifi. Towards a system-wide and transparent security mechanism using language-level information flow control. In *Proceedings of the International Conference on Security of Information and Networks*, SIN '10, pages 19–26. ACM, 2010.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, June 2009.

[BDP13] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the European Software Engineering Conference held jointly*

with the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '13, pages 411–421. ACM, 2013.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, 1975.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering*, FOSE '07, pages 85–103. IEEE Computer Society, 2007.

[BHH+11] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 53–62. IEEE Computer Society, 2011.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '07, pages 300–309. ACM, 2007.

[BK14] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '14, pages 83–101. ACM, 2014.

[BLM10] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. Testful: An evolutionary test approach for java. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '09, pages 185–194. IEEE Computer Society, 2010.

[BLS05] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1021–1028. ACM, 2005.

[BOP00] Ugo Buy, Alessandro Orso, and Mauro Pezzè. Automated testing of classes. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 39–48. ACM, 2000.

[BPS03] André Baresel, Hartmut Pohlheim, and Sadegh Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the conference on Genetic and Evolutionary Computation*, GECCO '03, pages 2428–2441. Springer, 2003.

[BSAD00] A.S Boujarwah, K Saleh, and J Al-Dallal. Dynamic data flow analysis for java programs. *Information and Software Technology*, 42(11):765 – 775, 2000.

[BW03] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of an automated parking system. In *Proceedings of the International Conference on Computer, Communication and Control Technologies and the 9th. International Conference on Information Systems Analysis and Synthesis*. IEEE Computer Society, 2003.

[CCMY96] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: Guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[CF07] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC '07, pages 463–475. IEEE Computer Society, 2007.

[CL95] T. Y. Chen and C. K. Low. Dynamic data flow analysis for c++. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 22–28, Dec 1995.

[Cla76] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491. ACM, 1976.

[CLM05] Tsong Yueh Chen, Hing Leung, and Keith Mak. Adaptive random testing. In *Advances in Computer Science*, volume 3321 of *Lecture Notes in Computer Science*, pages 3156–3157. Springer, 2005.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 71–80. ACM, 2008.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15, 1989.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 34(11):1025–1050, September 2004.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[CSL08] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50. ACM, 2008.

[DBG10] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '10, pages 215–224. IEEE Computer Society, 2010.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[DGP08] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. Contextual integration testing of classes. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, FASE '08, pages 246–260. Springer, 2008.

[DGP09] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. DaTeC: Dataflow testing of Java classes. In *ICSE Companion '09: Proceedings of the International Conference on Software Engineering (Tool Demo)*, pages 421–422. ACM, 2009.

[DH11] Haitao Dan and Robert M. Hierons. Conformance testing from message sequence charts. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '11, pages 279–288. IEEE Computer Society, 2011.

[DLR06] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '06, pages 157–166. ACM, 2006.

[DMPV15] Giovanni Denaro, Alessandro Margara, Mauro Pezzè, and Mattia Vivanti. Dynamic data flow testing of object oriented systems. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 947–958. IEEE Computer Society, 2015.

[DPV13] Giovanni Denaro, Mauro Pezzè, and Mattia Vivanti. Quantifying the complexity of dataflow testing. In *Proceedings of the International Workshop on Automation of Software Test*, AST '13, pages 132–138. IEEE Computer Society, 2013.

[DPV14] Giovanni Denaro, Mauro Pezzè, and Mattia Vivanti. On the right objectives of data flow testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '14, pages 71–80. IEEE Computer Society, 2014.

[dPX+06] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. *Proceedings of the International Conference on Automated Software Engineering*, 0:59–68, 2006.

[ECDD06] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 253–264. ACM, 2006.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'10, pages 1–6. USENIX Association, 2010.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '05, pages 17–30. ACM, 2005.

[FA11a] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[FA11b] Gordon Fraser and Andrea Arcuri. It is not the length that matters, it is how you control it. In *ICST*, ICST '11, pages 150–159. IEEE Computer Society, 2011.

[FA12a] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, pages 121–130. IEEE Computer Society, 2012.

[FA12b] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the International Conference on Software Engineering*, ICSE '12, pages 178–188. IEEE Computer Society, 2012.

[FA13] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[FA14] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[FAM15] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103(0):311–327, 2015.

[FW93] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19:774–787, 1993.

[FWH97] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[FZ12] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[GFA13] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '13. IEEE Computer Society, 2013.

[GFZ12] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, pages 67–77. ACM, 2012.

[GGJ+10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the International Conference on Software Engineering*, ICSE '10, pages 225–234. ACM, 2010.

[GHG07] Ahmed S. Ghiduk, Mary Jean Harrold, and Moheb R. Girgis. Using genetic algorithms to aid test-data generation for data-flow coverage. In *Proceedings of the Asia-Pacific Software Engineering Conference*, APSEC '07, pages 41–48. IEEE Computer Society, 2007.

[GIB+13] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for c/c++ using concolic execution. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 132–141. IEEE Computer Society, 2013.

[Gir93] Moheb R. Girgis. Using symbolic execution and data flow criteria to aid test data selection. *Software Testing, Verification and Reliability*, 3(2):101–112, 1993.

[GJG14] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 72–82. ACM, 2014.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[GLG12] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, pages 46:1–46:11. ACM, 2012.

[GLR09] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 474–484. IEEE Computer Society, 2009.

[GOC06] Leonard Gallagher, Jeff Offutt, and Anthony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification and Reliability*, 16(4):215–266, 2006.

[GPT+11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187. ACM, 2011.

[HA13] Mohammad Mahdi Hassan and James H. Andrews. Comparing multi-point stride coverage and dataflow coverage. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 172–181. IEEE Computer Society, 2013.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, February 2009.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proceedings of the International Conference on Software Engineering*, pages 232–242. IEEE Computer Society, 2003.

[Her76] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, 1976.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, ICSE '94, pages 191–200. IEEE Computer Society, 1994.

[HJL11] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222. ACM, 2011.

[HKC95] Hyoung Seok Hong, Yong Rae Kwon, and Sung Deok Cha. Testing of object-oriented programs based on finite state machines. In *Proceedings of the Asia-Pacific Software Engineering Conference*, APSEC '95, pages 234–241. IEEE, 1995.

[HKU02] R.M. Hierons, T.-H. Kim, and H. Ural. Expanding an extended finite state machine to aid testability. In *Proceedings of the International Computer Software and Applications Conference*, COMPSAC '02, pages 334–339. IEEE Computer Society, 2002.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 175–185. ACM, 2006.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '94, pages 154–163. ACM, 1994.

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 435–445. ACM, 2014.

[IX08] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '08, pages 297–306. IEEE Computer Society, 2008.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKP+08] Matthew Kaplan, Tim Klinger, Amit M. Paradkar, Avik Sinha, Clay Williams, and Cemal Yilmaz. Less is more: A minimalistic approach to uml model-based conformance test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '08, pages 82–91. IEEE Computer Society, 2008.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[KMW13] Gregory M. Kapfhammer, Phil McMinn, and Chris J. Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST 2013, pages 31–40. IEEE Computer Society, 2013.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '03. Springer, 2003.

[LC06] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC '06, pages 463–472. IEEE Computer Society, 2006.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 592–615. Springer, 2008.

[LH99] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '99, pages 199–215. Springer, 1999.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Proceedings of the International Conference on Compiler Construction*, CC '03, pages 153–169. Springer, 2003.

[LK83] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, 1983.

[LRW07] Konstantinos Liaskos, Marc Roper, and Murray Wood. Investigating data-flow coverage of classes using evolutionary algorithms. In *Proceedings of the conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1140–1140. ACM, 2007.

[MAZ+15] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '15, pages 212–223. ACM, 2015.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[MF11] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 436–439. IEEE Computer Society, 2011.

[Mir10] Matteo Miraz. *Evolutionary Testing of Stateful Systems: a Holistic Approach*. PhD thesis, Politecnico di Milano, 2010.

[MOP02] Vincenzo Martena, Alessandro Orso, and Mauro Pezzè. Interclass testing of object oriented software. In *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '02, pages 135–144. IEEE, 2002.

[MPE+10] Matteo Migliavacca, Ioannis Papagiannis, David M. Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. Defcon: High-performance event processing with information security. In *Proceedings of the USENIX Conference on Annual Technical Conference*, ATC '10. USENIX Association, 2010.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 416–426. IEEE Computer Society, 2007.

[MVZ+12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the International Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250. ACM, 2012.

[MW94] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[NMT12] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, pages 100–110. ACM, 2012.

[NT07] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '07, pages 215–224. ACM, 2007.

[NT08] Minh Ngoc Ngo and Hee Beng Kuan Tan. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*, 50(7-8):641–655, 2008.

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, 1984.

[NZR10] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. Mbsrt2: Model-based selective regression testing with traceability. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '10. IEEE Computer Society, 2010.

[OA99] A. Jefferson Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the International Conference on The Unified Modeling Language: Beyond the Standard*, UML '99, pages 416–429. Springer, 1999.

[OP97] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[OPTZ96] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software-Practice & Experience*, 26:165–176, 1996.

[OW91] Thomas J. Ostrand and Elaine J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV '91, pages 74–86. ACM, 1991.

[PHP99] Roy Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 75–84. ACM, 2007.

[PY07] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[RCV+15] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering*, SSBSE '15, pages 93–108. Springer, 2015.

[RHC76] C.V. Ramamoorthy, Siu-Bun F Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976.

[Rop97] Marc Roper. Computer aided software testing using genetic algorithms. *International Quality Week*, 1997.

[RPB+09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74. ACM, 2009.

[RVAF16] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, PP:to appear, 2016.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the International Conference on Compiler Construction*, CC'03, pages 126–137. Springer, 2003.

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '06, pages 419–423. Springer, 2006.

[SAP+11] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '11, pages 1053–1068. ACM, 2011.

[SAPE05] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '05, pages 114–123. ACM, 2005.

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22:777–793, 1996.

[SFP+15] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. Combining symbolic execution and model checking for data flow testing. In *Proceedings of the International Conference on Software Engineering*, ICSE '15, pages 654–665. IEEE Computer Society, 2015.

[SH07] Raul Santelices and Mary Jean Harrold. Efficiently monitoring data-flow test coverage. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '07, pages 343–352. ACM, 2007.

[SJYH09] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 56–66. IEEE Computer Society, 2009.

[SMS12] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *Proceedings of the International Conference on Quality Software*, QSIC '12, pages 79–88. IEEE Computer Society, 2012.

[SP03] Amie L. Souter and Lori L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018, 2003.

[TCMM98] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '98, pages 285–294. IEEE Computer Society, 1998.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex: White box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, pages 134–153. Springer, 2008.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128. ACM, 2004.

[Tre08] Jan Tretmans. Formal methods and testing. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer, 2008.

[TRLB+14] Federico Toledo Rodriguez, Francesca Lonetti, Antonia Bertolino, Macario Polo Usaola, and Beatriz Perez Lamancha. Extending uml testing profile towards non-functional test modeling. In *International Conference on Model-Driven Engineering and Software Development*, MODELSWARD '14, pages 488–497. IEEE Computer Society, Jan 2014.

[TXT+09] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202. ACM, 2009.

[TXT+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '11, pages 189–206. ACM, 2011.

[VMGF13] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '13, pages 370–379. IEEE Computer Society, 2013.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107. ACM, 2004.

[WBS01] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[Wey90] Elaine J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, 1990.

[WGMO10] Yi Wei, Serge Gebhardt, Bertrand Meyer, and Manuel Oriol. Satisfying test preconditions through guided object selection. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 0:303–312, 2010.

[WWGZ11] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transaction of Information System Security*, 14(2):1–28, 2011.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the Conference on USENIX Security Symposium*, SEC '06. USENIX Association, 2006.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '05, pages 365–381. Springer, 2005.

[XTdHS09] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '09, pages 359–368, 2009.

[YLW06] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the International Workshop on Automation of Software Test*, AST '06, pages 99–103. ACM, 2006.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 19–19. USENIX Association, 2006.

[Zha11] Sai Zhang. Palus: A hybrid automated test generation tool for java. In *Proceedings of the International Conference on Software Engineering*, ICSE '11, pages 1182–1184. ACM, 2011.

[ZZ02] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization*, pages 191–204. Springer, 2002.

[ZZLX10] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '10, pages 293–296. ACM, 2010.