DaTeC: Contextual Data Flow Testing of Java Classes*

Giovanni Denaro† denaro@disco.unimib.it
Alessandra Gorla‡ gorlaa@lu.unisi.ch
Mauro Pezzè†‡ mauro.pezze@unisi.ch

†*University of Milano Bicocca*
20126, Milan, Italy

‡*University of Lugano*
CH-6904, Lugano, Switzerland

Abstract

Many mature development processes use structural coverage metrics to monitor the quality of testing. Recent studies suggest that commonly used control flow testing criteria poorly address state-based behavior of object oriented software. This paper presents DaTeC, a tool that provides useful coverage information of Java object states by implementing a novel contextual data flow testing approach.

1 Introduction

In object oriented software the interactions of methods can bring class instances to complex states. Recent research work indicates that classic structural criteria do not provide enough information about the coverage of state-based behavior, and extends data flow coverage criteria to adequately cover state-based behavior [2, 3, 1]. In previous work, we described an efficient structural technique based on contextual data flow analysis to test state-dependent behavior of classes that aggregate other classes as part of their state [1]. Our technique draws on the early approach proposed by Harrold and Rothermel [2], and shares the main concept of contextual def-use associations with Souter and Pollock [3], but with different goals; Souter and Pollock pursue exhaustive analysis of single methods, while we focus on state-based interactions between methods that can be independently invoked from outside the classes.

This paper presents *DaTeC* (Data flow Testing of Classes), a data flow testing prototype that computes state-related coverage of Java classes. *DaTeC* is based on our contextual data flow analysis that extends Harrold and Rothermel's approach by adding context information when analyzing method invocations [1]. *DaTeC* uses contextual data flow information to identify the sequences of method invocations that are involved in relevant state-based interactions, and identifies states that are not adequately tested by computing context data flow coverage.

2 DaTeC structure

DaTeC computes contextual data flow coverage of Java programs by executing the Java bytecode suitably instrumented to record the coverage of contextual def-use associations. Testers can execute their test suites and incrementally double-check object state coverage. Data about contextual data flow coverage are made available at different levels of granularity, from the finest grain level that presents all covered and not-yet-covered associations, to the coarsest granularity level that summarizes the amount of covered pairs for selected classes only.

Figure 1 shows the components of *DaTeC*. The *Data Flow Analyzer*, built on top of *Soot*[1], identifies contextual definitions and uses of class state variables according to the approach that we presented in [1]. Contextual definitions and uses associate classic definitions and uses with a context, that is the chain of (nested) method invocations that lead to the definition or the use. In this way, a contextual data flow analysis records definitions and uses of a state variable for a method, either if the method accesses the variable directly or if it invokes other methods that access it. Classic data flow analysis does not record the context, and consequently can miss definitions and uses that are filtered by other methods, thus missing relevant state information. For example, classic data flow analysis does not record the definitions and the uses in methods that use getter and setter methods of other classes to access their state, while contextual data flow analysis does. The *Bytecode Instrumenter* instruments the bytecode of the classes, to monitor the execution of def-use associations. It is implemented on top of Soot too: it instruments (1) method entry and exit points to trace the context at runtime, (2) statements that contain definitions and uses of instance variables to identify both executed def-use associations and the object identity (hash

*This work has been partially funded by the SNF project PerSeoS.

[1] http://www.sable.mcgill.ca/soot

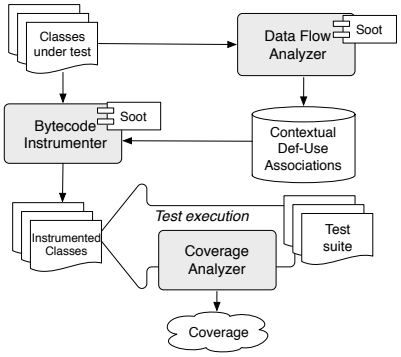


Figure 1. DaTeC components

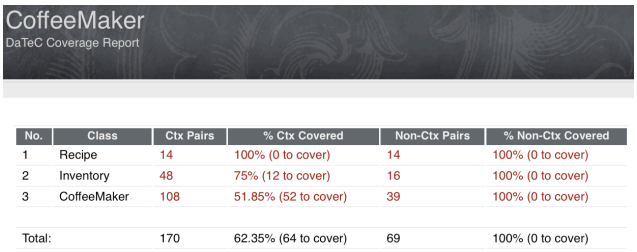
code). Recording the object hash code is important to correctly pair definitions and uses of instance variables of the same object, and thus record correct information on coverage. The *Coverage Analyzer* identifies at runtime the associations covered by the test suite, by tracing definitions to uses of instance variables of the same object.

3 DaTeC example

We present *DaTeC* through a simple Java control program for a coffee machine. The program is composed of a main class CoffeeMaker and two service classes Inventory and Recipe that store the ingredients and the portions needed for preparing the hotdrinks, respectively. The methods of CoffeeMaker access the instance variables of Inventory and Recipe through set and get methods. For example, method addInventory uses the setters and getters of class Inventory to modify the amount of available ingredients.

```
1  class CoffeeMaker{
2   public CoffeeMaker(){
3     ...
4    inv = new Inventory();
5   }
6   public boolean addInventory(int amtCoffee,int amtSugar){
7    if(amtCoffee < 0 || amtSugar < 0) return false;
8    /* FAULT! Next statement should use inv.getCoffee() */
9    inv.setCoffee(inv.getSugar() + amtCoffee);
10   inv.setSugar(inv.getSugar() + amtSugar);
11   return true;
12  }
13  public int makeCoffee(Recipe r,int amtPaid){
14    if(amtPaid < r.getPrice() || !inv.enoughIngredients(r))
15      return amtPaid;
16    inv.setCoffee(inv.getCoffee() - r.getAmtCoffee());
17    inv.setSugar(inv.getSugar() - r.getAmtSugar());
18    return amtPaid - r.getPrice();
19  }
20  ...
21 }
```

Method addInventory is faulty at line 9 since getSugar is called in place of getCoffee, a typical fault that can be induced by a syntax driven editor that auto-completes method names while the developer types. This fault can be revealed only by test cases that call method addInventory in a state


CoffeeMaker
DaTeC Coverage Report

No.	Class	Ctx Pairs	% Ctx Covered	Non-Ctx Pairs	% Non-Ctx Covered
1	Recipe	14	100% (0 to cover)	14	100% (0 to cover)
2	Inventory	48	75% (12 to cover)	16	100% (0 to cover)
3	CoffeeMaker	108	51.85% (52 to cover)	39	100% (0 to cover)
Total:		170	62.35% (64 to cover)	69	100% (0 to cover)

Figure 2. Report after testing CoffeMaker

different from the initial one, that is in a state with different values for coffee and sugar. Such state can be reached only by test cases that call method addInventory more than ones, since the first call of this method refers to zero amount of both coffee and sugar as set by the constructor. Classic coverage criteria do not require methods to be invoked in different states, and thus can be satisfied by test suites that do not reveal this fault. For instance, the following test cases do not reveal the fault, albeit providing 100% statement coverage of the code:

CoffeeMaker(); addInventory(-10, 20);
CoffeeMaker(); addInventory(10, 20); ... makeCoffee(R, 5)
CoffeeMaker(); addInventory(10, 20); ... makeCoffee(R, 1)

Similarly, classic data flow testing criteria can be satisfied by test suites that do not reveal the fault. In fact, they require test cases to cover the state of classes Inventory and Recipe that contain direct accesses to the state variables coffee and sugar, but not the state of class CoffeeMaker that accesses these state variables through setter and getter methods of the other classes.

Figure 2 shows the report produced by *DaTeC* after executing a test suite that satisfies both statement and non-contextual criteria, but does not reveal the fault. The test suite covers all non-contextual def-use pairs, but misses several contextual def-use pairs. Looking at the coverage report of class CoffeeMaker, the contextual pair that involves variable inv.sugar defined within method addInventory at line 10, and used within the subsequent invocation of the same method at line 9 is not covered. This is the pair that requires the execution of the faulty statement in a faulty state, and has to be covered by a test suite satisfying contextual criteria.

References

[1] G. Denaro, A. Gorla, and M. Pezzè. Contextual integration testing of classes. In *Proc. of FASE*, pages 246–260, 2008.
[2] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proc. of FSE*, pages 154–163, 1994.
[3] A. L. Souter and L. L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE TSE*, 29(11):1005–1018, 2003.