An Empirical Evaluation of Data Flow Testing of Java Classes

Technical Report No. 2007/03, January 2007

Giovanni Denaro[1] and Alessandra Gorla[2] and Mauro Pezzè[1,2] *

[1] University of Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione, Via Bicocca degli Arcimboldi 8, 20126, Milano, Italy
denaro@disco.unimib.it
[2] University of Lugano, Faculty of Informatics, via Buffi 13, 6900, Lugano, Switzerland
mauro.pezze@unisi.ch,alessandra.gorla@lu.unisi.ch

Abstract. This paper tackles the problem of structural integration testing of stateful classes. Previous work on structural testing of object-oriented software exploits data flow analysis to derive test requirements for class testing and defines contextual def-use associations to characterize inter-method relations. Non-contextual data flow testing of classes works well for unit testing, but not for integration testing, since it misses definitions and uses when properly encapsulated. Contextual data flow analysis approaches investigated so far either do not focus on state dependent behavior, or have limited applicability due to high complexity. This paper proposes 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 Introduction

Object-oriented programs are characterized by classes and objects, which enforce encapsulation, behave according to their internal state, inherit features from ancestors, separate normal from exceptional cases, and dynamically bind polymorphic types [1]. Object-oriented features discipline programming practice, and reduce the impact of some critical classes of faults, e.g., those that derive from excessive use of non-local information or from unexpected access to hidden details. However, they introduce new behaviors that cannot be checked satisfactorily with classic testing techniques, which assume procedural models of software [2]. In this paper, we focus on structural testing of state-based behavior, which impacts on both unit and integration testing of classes

* This technical report provides additional data about the experiments reported in the paper submitted to FASE '08.

State-based behavior is addressed by both functional and structural testing techniques. Many functional testing techniques derive test cases from finite state machines ([3–7]) or Statecharts ([8–11]). These methods generate test cases from specifications of single classes, and are well suited for unit testing. Other approaches consider communicating Statecharts ([12]), sequence diagrams ([13]) and collaboration diagrams ([14]), sometimes combined with class diagrams ([15]). These methods address the aspects of integration testing defined in the design specifications, but miss detailed design decisions that do not show up in specifications. Algebraic approaches propose an interesting solution to the oracle problem [16–18].

The most promising structural approaches to testing object oriented software exploit data flow analysis to implicitly capture state-based interactions. Harrold and Rothermel proposed data flow analysis for structural testing of classes in 1994 [19]. In their early work, Harrold and Rothermel define a class control flow graph to model data flow interactions among classes, and apply data flow analysis to identify flow relations of class state variables. The analysis supports well unit testing, but does not apply satisfactorily to integration testing of classes. In fact, when accesses to variables are properly encapsulated, standard data flow analysis does not identify chains of interactions that flow through different classes. At the same time, Harrold and Soffa proposed an efficient algorithm for inter-procedural data flow analysis that captures the propagation of definitions and uses through chains of invocations [20]. In 2003, Souter and Pollock proposed a contextual data flow analysis algorithm for object oriented software [21]. Souter and Pollock's algorithm distinguishes accesses to the same variables through different chains of method invocations, and thus captures inter-method interactions even when variables are encapsulated in classes. Differently from our work, the work of Souter and Pollock does not focus on state dependent behavior. Moreover, Souter and Pollock's algorithm is very accurate, but quite expensive ($O(N^4)$ in the size of the program). The complexity of the algorithm limits the scalability of the approach and the development of efficient tools.

This paper proposes an approach to state based integration testing of classes that properly extends the early approach by Harrold and Rothermel: We use contextual information about method invocations to analyze state-dependent behavior of classes that can aggregate other classes as part of their state. We share the main concept of contextual def-use associations with Souter and Pollock, 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. We extend the algorithm by Harrold and Soffa to include contextual information, thus obtaining an algorithm for contextual data flow analysis more efficient, albeit less accurate than the algorithm of Souter and Pollock. Our algorithm is quadratic in the size of the program in the worst case, and more efficient in many practical cases. Thus, it provides the background to define practical structural coverage criteria for integration testing of classes. The efficiency is obtained by ignoring details that are useful for analysis, but not relevant for defining testing criteria.

```
1  public class AClass {
2    public void aMethod(){...
3    ...
4  }
5  public class AGlobalObj {
6    static public AGlobalObj getInst()
         {...
7    public void aGOMethod(){...
8    ...
9  }
10 public class AnotherClass {
11   public void anotherMethod(){
12     AGlobalObj.getInst().aGOMethod();
13     ...
14   }
15   ...
16 }
```

```
17 public class YetAClass {
18   AClass aStateVar = new AClass();
19   public void yetAMethod1(AClass aPar){
20     AClass aLocalVar = new AClass();
21     ...
22     aLocalVar.aMethod();
23     aPar.aMethod();
24     aStateVar.aMethod();
25     AGlobalObj.getInst().aGOMethod();
26     ...
27   }
28   public void yetAMethod2(){
29     aStateVar.aMethod();
30     ...
31   }
32   ...
33 }
```

Fig. 1. Object interactions in Java

This paper classifies the different ways in which Java classes inter-operate (Section 2), describes an efficient contextual data flow analysis approach for object oriented software, discusses the complexity, and proposes structural coverage criteria for class integration testing (Section 3). It introduces a prototype implementation that we used for evaluating the suitability of the proposed approach (Section 4). It presents empirical data that support the applicability of the proposed coverage, and discusses the limits of the approach (Section 5). It surveys the main related work, acknowledges the contribution of previous research (Section 6). It concludes by summarizing the contribution of the paper, and by describing ongoing research work (Section 7).

2 Integration Testing of Classes for Java Programs

Java programs are collections of objects, which consume inputs, modify their states, and produce outputs through their mutual interactions. Integration testing aims to check the interactions of subsets of objects. Integration test cases are sequences of invocations of methods that belong to the considered subsets.

For Java objects to interact, at least one object must be able to access a reference to other objects. Java objects can interact through references stored in local variables of methods, parameters of methods, state variables of classes, and global objects. Here we briefly describe these different kinds of interaction, referring to the sample code shown in Figure 1:

Local variables: an object `obj1` can establish a link to another object `obj2` by instantiating `obj2` as local variable (for instance line 20 instantiates the object used at line 22 in Figure 1). This includes the case of objects referenced by temporary variable that are used in invocations without being stored in explicitly declared variables.

Parameters: an object `obj1` can establish a link to another object `obj2` by receiving a reference to `obj2` as input parameter of a method invocation (for instance line 19 receives a parameter that is used at line 23 in Figure 1).

State variables: an object `obj1` can establish a link to another object `obj2` by storing a reference to `obj2` as state variable (for instance in Figure 1line 18 defines a state variable that is used at lines 24 and 29).

Global objects: an object `obj1` can access a reference to another object `obj2` though a global registry, as prescribed for example by the singleton pattern [22] (see for instance lines 5-9, 12 and 25 in Figure 1).

Aliases can introduce additional interactions, objects can for example cache references to global information using their local variables. Classic alias analysis of procedural programs is being successfully extended to object-oriented programs [23, 24]. In this paper, we assume that most aliases have been statically solved before data flow analysis.

Completing the definitions proposed by Harrold and Rothermel, we consider four testing levels for object-oriented programs: *intra-method*, *inter-method*, *intra-class* and *inter-class* [19]. Intra-method testing considers methods in isolation, and does not deal with object interactions, thus it is not interesting from the integration testing viewpoint.

Inter-method testing considers interactions that derive from methods as they access other methods through given object references. For example when testing method `yetAMethod1` of the Java program in Figure 1, we test the interactions of class `YetAClass` with instances of class `AClass` at line 22 (instantiated as the local variable `aLocalVar`) and line 23 (accessed through parameter `aPar`).

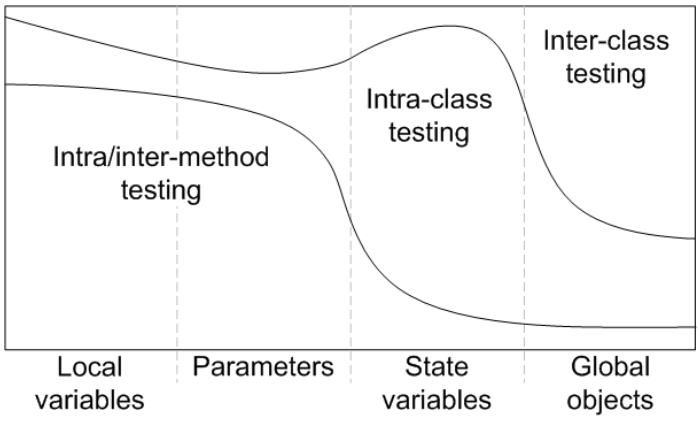

Fig. 2. Effectiveness of integration testing levels for different types of interactions

Intra-class testing considers interactions that derive from invoking many methods of the same class instance. For example, when executing an intra-

class test case that invokes methods `yetAMethod1` and `yetAMethod2` of class `yetAClass` (lines 24 and 29) in Figure 1, we test interactions between classes `AClass` and `yetAClass` through the state variable `aStateVar` (line 18).

Inter-class testing considers interactions that derive from invoking many methods of a set of class instances. For example, when executing an inter-class test case that invokes methods `anotherMethod` and `yetAMethod1`, we test the interaction between the classes `AnotherClass`, `YetAClass` and the singleton `AGlobalObj`, which is shared between the two methods (lines 12 and 25).

Testing levels focus on different types of interactions. Figure 2 summarizes (qualitatively) the effectiveness of the different testing levels. Intra- and inter-method testing focus on interactions within the scope of single methods, that is mainly through local variables and parameters. Intra-class testing focuses on interactions between methods of the same class, mostly through state variables. Inter-class testing focuses on interactions between classes, mostly through shared or global objects.

The technique proposed in this paper focuses on interactions among classes that derive from the aggregation relation. Thus, we consider the relations among state variables that are most relevant in the context of integration testing (the critical aspects of the area identified by "state variables" in Figure 2.)

3 Contextual Data Flow Testing of Classes

A class is a (possibly) stateful module that encapsulates data and exports operations (aka methods). At runtime, a method takes a class instance (aka an object) as part of its input, and reads and manipulates the object state as part of its action. Thus in general, the behavior of the methods depends on the state of the objects, and the set of reachable states of the objects depends on the methods. The states of the classes are often composed of instances of other classes. To thoroughly test classes, we need to identify test cases, i.e., sequences of method calls, that exercise the relevant state dependencies of the methods in the reachable (composed) states.

Data flow analysis can assist testing of classes by both identifying the relevant dependencies between methods and capturing definitions and uses of state variables [19]. Section 3.1 briefly recaps the minimal background on data flow testing. Section 3.2 discusses the limits of classic data flow analysis in identifying chains of interactions that flow through different classes. As a consequence, classic data flow analysis misses interactions through structured variables properly encapsulated in classes, which are extremely relevant during integration testing. Section 3.3 introduces our approach that leverages previous work on data flow testing of classes, making it amenable for the purpose of class integration testing.

```
1  public class Msg {
2    private byte info;
3    public Msg(){info = 0;}
4    public void setInfo(byte b){info=b;}
5    public byte getInfo(){return info;}
6  }
7  public class Storage {
8    private Msg msg;
9    private byte stored;
10   public Storage(){
11     msg = new Msg();
12     stored = 0;
13   }
14   public void setStored(byte b){
15     stored = b;
16   }
17   public byte getStored(){
18     return stored;
19   }
20   public void recvMsg(Msg m){
21     byte recv = m.getInfo();
22     msg.setInfo(recv);
23   }
24   public void storeMsg(){
25     byte b = msg.getInfo();
26     setStored(b);
27   }
28 }
```

Fig. 3. A sample Java program

3.1 Def-use associations

Given a program variable v, a standard (context-free) def-use association (d, u) is a pair of program locations, definition (d) and use (u) locations for v, where v is assigned a new value at d, and has its value read and used at u. Testing a def-use association (d, u) executes a program path that traverses first d and then u, without traversing statements that define v between d and u (the subpath between d and u is *def-free*). Def-use associations lead to many testing criteria [25].

Methods of a class interact through local variables, parameters, state variables and global objects. Def-use associations capture such interactions by identifying methods that use values set by other methods. For example the def-use association for variable `stored` at lines $(15, 18)$ of Figure 3 captures the dependency between methods `getStored()` and `setStored()`.

Def-use associations that involve local variables and parameters characterize method interactions through these elements, and are exploited by many tools, e.g., Coverlipse [26], and are not further considered in this paper.

3.2 Contextual def-use associations

Contextual def-use associations extend def-use associations with the *context* of the method invocations. A context is the chain of (nested) method invocations that leads to the definition or the use [21][3]. A contextual def-use association for a variable v is a tuple (d, u, cd, cu), where (d, u) is a def-use association for v, and cd and cu are the *contexts* of d and u, respectively. This concept is illustrated in Figure 1: The context-free def-use association for variable `stored` at lines (15,18) corresponds to two contextual def-use associations ($\rightarrow$ indicates method calls):

[3] In presence of multiple invocations from the same method, context may or may not distinguish the different invocation points. For the goals of this paper we do not need to distinguish different invocation points in the same method.

(15, 18, `Storage::setStored()`, `Storage::getStored()`) and

(15, 18, `Storage::storeMsg` → `Storage::setStored()`,`Storage::getStored()`)

In general, in absence of context information, def-use associations are satisfied by test cases that focus on trivial rather than complex interactions, while context information identifies a more thorough set of test cases. For example, the context-free def-use associations at lines $(15, 18)$ can be satisfied with a simple test case that invokes methods `setStored()` at line 14 and `getStored()` at line 17, while the corresponding contextual def-use associations illustrated above requires also the (more interesting) invocations of methods `storeMsg()` at line 24 and `getStored()` (line 17).

We clarify the concept through some common design practice: accessor methods and object aggregation.

Accessor methods are used to guarantee controlled access to state variables. A common design style is to define *get* methods (e.g., `getStored()`) to access state variables, and *set* methods (e.g., `setStored()`) to define state variables. Context-free definitions and uses of variables with accessors are always located within the accessor methods themselves, by construction. Thus, all state interactions that involve these variables are characterized by the few (context-free) def-use associations that derive from the accessor methods, while the many interactions mediated by the accessors are not captured. A test suite that covers all (context-free) def-use associations involving accessors would focus on trivial interactions only, missing the relevant ones. Contextual def-use associations distinguish between direct and mediated invocations of accessors, thus capturing also interactions between methods that access state variables through accessors. In the previous example the test cases derived from context-free def-use associations would focus on the trivial interaction between `setStored()` and `getStored()` only, missing the more relevant interaction between `storeMsg()` and `getStored()`, while the test cases derived from contextual def-use associations would capture all interactions through variable `stored`.

Object aggregation indicates the use of an object as part of the data structure of another object. Since it is a good practice to encapsulate the state of an aggregated object within its methods, definitions and uses of the internal state variables are located within these methods. State interactions that involve internal variables of aggregated objects are then characterized by context-free def-use associations that involve methods of the aggregated object only. Test cases that cover context-free def-use associations focus on single objects and not the aggregated ones, thus missing complex and usually semantically more relevant interactions, while contextual def-use associations identify interactions of simple as well as aggregated objects and lead to a more thorough set of test cases. For example the context-free def-use association for variable `info` at lines $(4, 5)$ in Figure 3 characterizes both the interaction between methods `setInfo()` and `getInfo()` in class `Msg`, and the interaction between methods `recvMsg()` and `storeMsg()` in class `Storage` (which aggregates a `Msg` object as part of its state).

Table 1. Definitions and uses computed for the sample program in Figure 3.

Method	Line (state var)	Context
Defs@exit		
Msg::Msg	3 (info)	Msg::Msg
Msg::setInfo	4 (info)	Msg::setInfo
Storage::Storage	12 (stored)	Storage::Storage
Storage::setStored	15 (stored)	Storage::setStored
Storage::storeMsg	15 (stored)	Storage::storeMsg→Storage::setStored
Storage::Storage	11 (msg)	Storage::Storage
Storage::Storage	3 (msg.info)	Storage::Storage→Msg::Msg
Storage::recvMsg	4 (msg.info)	Storage::recvMsg→Msg::setInfo
Uses@entry		
Msg::getInfo	5 (info)	Msg::getInfo
Storage::getStored	18 (stored)	Storage::getStored
Storage::recvMsg	22 (msg)	Storage::recvMsg
Storage::storeMsg	25 (msg)	Storage::storeMsg
Storage::storeMsg	5 (msg.info)	Storage::storeMsg→Msg::getInfo

3.3 Deriving contextual associations

We compute state-based def-use associations by first identifying contextual definitions and uses that reach method boundaries, and then pairing definitions and uses of the same state variables across methods. In this phase, our main contribution is the redefinition of the first step of classic data flow algorithms: Differently from Harrold and Rothermel, we compute contextual information of definitions and uses, and thus we capture important interprocedural properties of object oriented software; We adapt the classic Harrold and Soffa's interprocedural algorithm to contextual definitions and uses in the context of object oriented software; We borrow the definitions of contextual information by Souter and Pollock, but we use a more efficient algorithm that focuses on definitions and uses of state variables that reach method boundaries as illustrated in the next paragraphs.

We present the algorithm through the example of Figure 3. In the first step, we statically analyze the methods of the classes in isolation, and we compute two sets of data, *defs@exit* and *uses@entry*. The *defs@exit* set includes all definitions of state variables that can reach the end of the method, i.e., for which there exists at least a (statically identified) def-free path from the definition to an exit of the method. The *uses@entry* set includes all uses of state variables that can be reached from the entry of the method, i.e., for which there exists at least one def-free path from the entry of the method to the use. For each element in the two sets, we record location, related state variable, and context information. Table 1 shows the *defs@exit* and *uses@entry* sets for the code in Figure 3.

In the second step, we match the information computed in the first step, by combining definitions in *defs@exit* and uses in *uses@entry* that relate to same state variables. In this way, we compute the set of contextual def-use associations for the class under analysis. Table 2 shows the complete set of def-use associations for the classes in Figure 3.

Table 2. Def-use associations for the sample program in Figure 3.

Class (state var)	Association	Def context	Use context
Msg (info)	(3, 5)	Msg::Msg	Msg::getInfo
Msg (info)	(4, 5)	Msg::setInfo	Msg::getInfo
Storage (stored)	(12, 18)	Storage::Storage	Storage::getStored
Storage (stored)	(15, 18)	Storage::setStored	Storage::getStored
Storage (stored)	(15, 18)	Storage::storeMsg→Storage::setStored	Storage::getStored
Storage (msg)	(11, 22)	Storage::Storage	Storage::recvMsg
Storage (msg)	(11, 25)	Storage::Storage	Storage::storeMsg
Storage (msg.info)	(3, 5)	Storage::Storage→Msg::Msg	Storage::storeMsg→Msg::getInfo
Storage (msg.info)	(4, 5)	Storage::recvMsg→Msg::setInfo	Storage::storeMsg→Msg::getInfo

Our data flow analysis implements intra-procedural analysis according to the classic reaching definition algorithm [27, 28]. We then compute inter-procedural relationships by elaborating the inter-procedural flow graph (IFG), as proposed by Harrold and Soffa [20]. We extended the algorithm to propagate the context information on the control-flow edges that represent inter-procedural relationships. The details of the extensions are illustrated in Figure 4: we refer to the pseudocode from [20] and highlight the points where modifications are required to include context information.

Context tracking in presence of recursive calls and programs with recursive data structures requires specific handling. Recursion of method calls may generate infinitely many new contexts, which may cause the algorithm to diverge. To avoid divergence, at each node of the IFG our algorithm distinguishes only one level of nested calls, by merging contexts that contain repeated subsequences of method calls. Recursive data structures define aggregations of possibly infinite state variables, which may generate unbounded sets of definitions and uses. Our algorithm expands recursive data structures up to one level of recursion. While limiting the depth of recursion and recursive data structures may threaten the validity of the analysis, it should not impact significantly on integration testing, since one level of depth is enough to test at least once all interactions between distinct modules.

The extended algorithm works with the same number of propagation steps as the original algorithm, and thus has a temporal complexity of the same order of magnitude as the original one ($O(n^2)$ worst case complexity.) Space complexity slightly increases because of the extra space for memorizing context information, and because definitions and uses that reach IFG nodes through different contexts are now tracked as distinct items.

Both the algorithms for computing intra- and inter-procedural information are rapid data flow problems, thus although their worst-case complexity is quadratic in the size of the analyzed program, the complexity is linear in most practical situations ([27, 28, 20]). (Recall that Souter and Pollock's algorithm is $O(n^4)$.) In our experiments, we completed static data flow analysis in few minutes, even for large programs (see Section 5 for details.)

```
procedure propagate(N, E)
input  N: set of node types to be processed
       E: set of edge types to be processed

BEGIN
  WHILE dataflow changes DO
    FOR each node n of type N DO
      FOR each node s that is a successor over E of n DO
        let e in E be the edge (n -> s)
        //begin extension
        IF (n is a call node) and (s is the entry node of procedure P) THEN
          IN'_use[s] = addContext(P, IN_use[s])
        ELSE
          IN'_use[s] = IN_use[s]
        //end extension
        OUT_use[n] = OUT_use[n] U IN'_use[s]
        IN_use[n] = OUT_use[n] U UPEXP[n]
      OD
    OD
  OD
END propagate




procedure addContext(C, INSET): OUTSET
input  C: the new context
       INSET: set of propagated uses
output OUTSET: modified set of propagated uses

BEGIN
  OUTSET = copy of INSET, where context is updated by C
  prune recursive contexts for items in OUTSET
  return OUTSET
END addContext
```

Fig. 4. Details of our *context-sensitive* extension of the algorithm from [20]. The extensions apply to the procedure *Propagate* in the inte-procedural data flow analysis algorithm from [20](refer to [20] for the role of *Propagate* in the algorithm). With no loss of generality, the figure shows how we extend *Propagate* to track context information, in the case of backward propagation of uses. Refer to [20] for the version of *Propagate* for forward propagation of definitions.

4 Evaluation Framework

Contextual def-use analysis captures many non-trivial object interactions that depend on program state. Thus, covering contextual def-use associations should increase the possibility of revealing state-dependent faults and the confidence in the proper behavior of the program. However, complex interaction patterns may result in many contextual def-use associations, that may be expensive to compute. Moreover, the contextual information may increase the amount of infeasible associations, thus weakening testing criteria based on contextual def-use associations.

To estimate the actual costs of computing def-use associations, the amount of infeasible associations and the amount of faults that can be revealed only when covering contextual def-use associations, we built a prototype implementation that computes all contextual-uses coverage of Java programs.

Figure 6 and 5 illustrate the fundamental components of the **DaTeC** (Data flow Testing of Classes) prototype and the information flow to and from a classic testing environment. In a classic testing environment, a *test driver*, e.g., *JUnit*, executes a set of *test cases* for the *classes under test*, and produces a set of *test results*. The **DaTeC** prototype extends a classic testing environment to compute the coverage of contextual def-use associations.

The prototype is composed of a *data flow analyzer*, `DUAnalyzer`, that statically computes the contextual def-use associations for the class under test, a *code tracer*, `Instrumenter`, that inserts probes into the classes under test to identify the executed def-use associations, and a *coverage analyzer*, `DUCoverage`, that computes the "all contextual uses coverage". The *code tracer* has been implemented by modifying the `setUp()` and `tearDown()` methods of the `TestCase` class in JUnit. The first method is automatically executed every time a JUnit

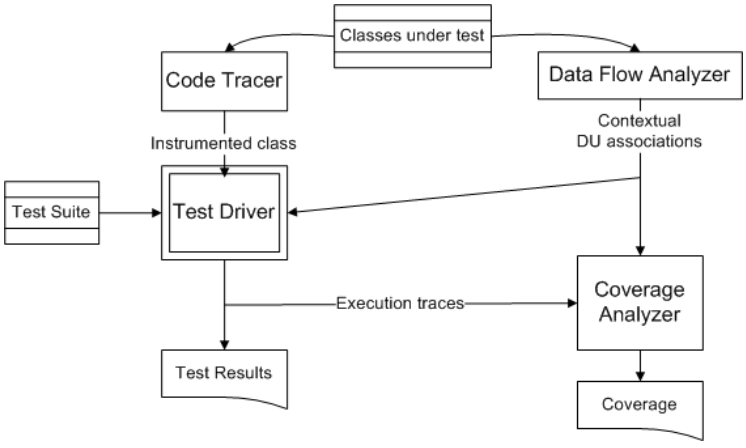

Fig. 5. The logical structure of the **DaTeC** prototype

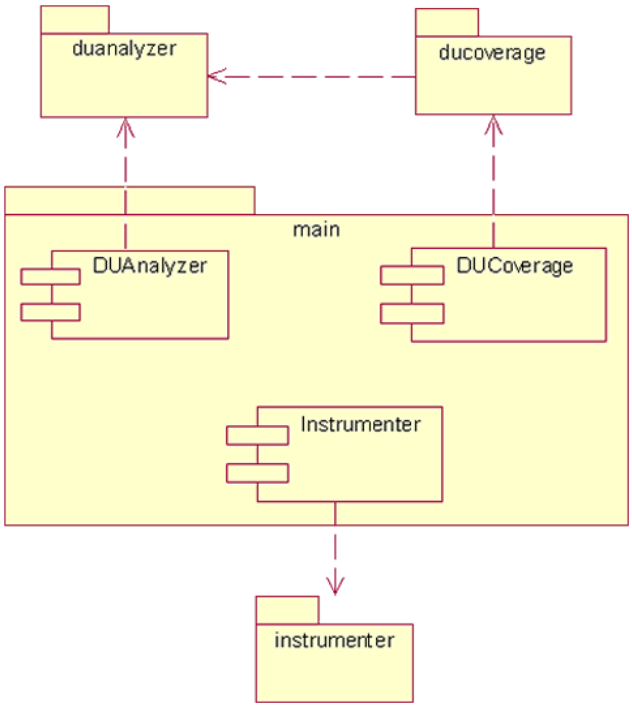

Fig. 6. The **DaTeC** component diagram

test case is run. The second one is invoked when the execution is over. Our modified JUnit library creates a different trace file for each test case that is executed. The content of a file is the execution trace of the test case. The application under test has to be instrumented according to the information provided by the data flow analyzer. Below we describe the data flow and the coverage analyzers.

4.1 Data flow analyzer

The *data flow analyzer* implements a classic reaching definition algorithm [27, 28]. The algorithm computes the definitions that may reach uses with a forward visit of a control flow graph. It uses a lattice of bit vectors where each bit corresponds to a definition to optimize computation space.

We instantiated the algorithm for analyzing Java bytecode. Our implementation relies on the JABA API (Java Architecture for Bytecode Analysis [29]) for exploring the control flow graphs of class methods. We adapted the traditional algorithm to compute the *uses@entry* and *defs@exit* sets for each method of the classes under test (the uses that are reachable from the method entry and the definitions that reach the method exit, as defined in Section 3.3.) We extended the algorithm to trace the context information of definitions and uses.

The temporal complexity of the algorithm is linear in the size of the input. In fact, we analyze each method at most once, since we handle invocations between methods at the invocation sites, by importing the information that results from the analysis of the called methods. Moreover, we traverse the nodes of the control flow graphs at most a constant number of times, since we visit the control flow graphs in *depth-first reverse post order* [28]. The upper bound for node traversal depends on the complexity of loops, which is usually low in object-oriented programs. In our experiments, we completed static data flow analysis in few minutes, even for large programs (see Section 5 for details.)

4.2 Coverage analyzer

The *coverage analyzer* computes the def-use associations covered by a set of program traces. A trace T is a sequence of program locations that correspond to a program execution. For each location, a trace records the class and the object involved in the execution at that point.[4] A trace $\overline{T}$ covers a def-use association $(\overline{d}, \overline{u}, \overline{cd}, \overline{cu})$ for a state variable $\overline{v}$ if and only if:

- the locations $\overline{d}$ and $\overline{u}$ occurs in $\overline{T}$ in this order;
- the variable $\overline{v}$ is instantiated within the same object;
- no killing definition for $\overline{v}$ occurs in $\overline{T}$ between $\overline{d}$ and $\overline{u}$;
- the trace $\overline{T}$ satisfies the contexts $\overline{cd}$ and $\overline{cu}$ at locations $\overline{d}$ and $\overline{u}$, respectively. A trace T satisfies a context c at a location l, if, in T, l is reached through a chain c of method invocations. Notice that if T satisfies c at l, it satisfies also all contexts obtained by considering the tails of c.

For example, referring to Figure 3, consider the trace $\overline{T}$ showed in Table 3, obtained by executing the method sequence *s=new Storage(); s.storeMsg(); s.getStored();* for an object s of class Storage which instantiates an object m of class Msg.

$\overline{T}$ covers the def-use association

⟨15, 18, Storage::setStored(), Storage::getStored()⟩

since it contains locations 15 and 18 that define and use variable *stored* of object s, without killing definitions in between.

$\overline{T}$ covers also the def-use association

⟨15, 18, Storage::storeMsg→Storage::setStored(),
Storage::getStored()⟩

since location 15 is reached through the invocation of method $storeMsg()$ at line 24.

The algorithm for coverage analysis is computation intensive. The trivial algorithm that scans all traces and checks all def-use associations for each trace

[4] Object identifiers are ignored for `static` methods.

class	method	line	object
Storage	Storage()	10	s
Msg	Msg()	3	m
Storage	Storage()	11	s
Storage	Storage()	12	s
Storage	storeMsg()	24	s
Msg	getInfo()	5	m
Storage	storeMsg()	25	s
Storage	setStored()	14	s
Storage	setStored()	15	s
Storage	storeMsg()	26	s
Storage	getStored()	17	s
Storage	getStored()	18	s

Table 3. Execution trace of the method sequence *s=new Storage(); s.storeMsg(); s.getStored();*

has a complexity linear in the product of the number of traces, the length of the traces, and the number of def-use associations. To reduce the execution time, we compare contexts incrementally, to avoid waste of memory for long traces, and we index the def-use associations by the traversed locations. In this way, we consider only the associations related to the locations in the trace currently analyzed, thus reducing the impact of the total number of associations on the complexity. In our experiments, we processed programs with traces containing up to 10^8 entries, and a total of up to 10^5 def-use associations without problems.

4.3 Scope of the prototype

Although the technique is applicable to general Java programs, the prototype does not currently handle exceptions and polymorphism; it does not statically analyze the code for aliases; and it treats arrays with some imprecision. In this subsection we briefly discuss the impact of these limits on the empirical results.

Exceptions require special treatment as discussed by Sinha and Harrold [30], Chatterjee and Ryder [31], and Chatterjee et al. [32]. Exceptions are part of the plans for the next release of our prototype. In the current release, the data flow analyzer ignores exception handlers and the coverage analyzer does not report the related coverage.

Polymorphism and dynamic binding can cause an exponential explosion of combinations. For example, Rountev et al. propose a technique to efficiently treat polymorphism based on class analysis [33]. The current prototype does not solve the bindings, but considers the ones indicated by the user.

As mentioned in Section 2, aliases widen the set of possible interactions. The current prototype relies on JABA for building control flow graphs annotated

with definitions and uses. We will include the JABA alias analyzer [24] as soon as released.

Data flow analysis, as most static analysis techniques, cannot handle well arrays, since array elements are often accessed though indexes whose values are computed dynamically. The Java dynamic initialization of arrays worsen the problem. Forgács and Hamlet et al. present some techniques for handling arrays efficiently [34, 35]. The current prototype approximates arrays as a whole, without distinguishing accesses to single element.

The limitations discussed in this section characterize most static analysis techniques, but none of them prevents the applicability of data flow analysis.

5 Empirical data

We identified three main aspects that characterize the efficiency of the technique proposed in this paper: ability of dealing with large programs (scalability), amount of infeasible def-use associations (feasibility) and ability of revealing failures (effectiveness).

Program name	Vers.	Description	Analyzed elements	SLOC	No. of classes	State vars
JEdit	4.2	Programming text editor	Whole	92,213	910	2,975
Ant	8	Java build tool	Whole	80,454	785	4,081
BCEL	5.2	Bytecode engineering library	Whole	23,631	383	929
Lucene	2.0	Text search engine library	Whole	19,337	287	1,013
JTopas	4	Java tokenizer and parser tool	Whole	5,359	63	196
NanoXML	5	XML parser	Whole	3,279	25	39
Siena	2	A wide-area event notification service	Whole	2,162	27	66
JUnit	3.8.1	Regression testing framework	junit.framework	650	12	22
Coffee Maker	/	Simple representation of a Coffee Machine	Whole	279	3	14

Table 4. Features of the analyzed applications.

5.1 Scalability

The size of the programs that can be analyzed may be limited by the complexity of data flow and coverage analysis. As discussed in Section 4, data flow analysis is linear in the size of the code, while coverage analysis depends on both the length of the analyzed traces and the amount of def-use associations.

To appreciate the impact of the complexity of data flow and coverage analysis, we analyzed a set of sample open-source programs of increasing complexity with

	Max assoc. for 90% of classes	Max assoc. for 95% of classes	Max assoc. per class	Max defs of a state-var per method	Max uses of a state-var per method	Exec time 1 in sec	Exec time 2 in sec	Exec time 3 in sec	Average time in sec
Jedit	44	130	255,608	251	879	66.935	65.707	79.763	∼70 s
Ant	91	198	11,337	200	1,990	60.61	60.857	60.945	∼60 s
BCEL	23	81	4,166	42	100	27.928	22.344	22.363	∼24 s
Lucene	74	170	43,557	90	2,930	17.879	17.821	17.988	∼17 s
JTopas	56	120	32,588	72	170	11.36	7.778	7.819	∼8 s
NanoXML	76	137	209	2	68	5.753	5.653	5.665	∼5 s
Siena	15	23	1,993	26	75	5.456	5.455	5.488	∼5 s
JUnit	21	30	39	2	23	4.593	4.588	4.658	∼4 s
CoffeeMaker	49	53	56	5	17	2.355	2.442	2.353	∼2 s

Table 5. Amount of data computed by the data flow analyzer for a set of sample programs

our prototype. Table 4 reports the features of the applications we analyzed. Most of them are available at the Software-artifact Infrastructure Repository [36] (Ant, JTopas, NanoXML and Siena). Apache website provides two additional applications (BCEL and Lucene) and SourceForge website hosts the others. We performed a complete analysis of all the programs, except for JUnit, for which we have considered only the core package of the application (`junit.framework`).

Columns SLOC, No. of classes and State vars of Table 4 report the size of the applications, expressed in number of classes (including non public classes), source lines of code (computed with SLOCCount), and number of state variables. Table 5 reports the data flow analysis results for all the analyzed programs. In the first columns of this table, we report the number of definitions, uses and associations for single methods and classes. Data are given per class and method, since in this paper we focus on integration testing, and we are thus interested in the amount of information relative to the classes under test and not to the programs as a whole.

The table suggests that in general the amount of associations per class does not depend on the complexity of the program, but on the amount of definitions and uses per method (see the columns *Max defs of a state var per method* and *Max uses of a state var per method*). For example, Lucene is smaller than BCEL, but the amount of definitions and uses per method as well as the amount of associations are higher in the smaller program.

The number of associations for a single class can be very high: it is more than 255,000 in the worst case (JEdit, column *Max assoc. per class*). However, most classes are characterized by a limited amount of associations: up to 44 if we do not consider the 10% of the classes with the highest number of associations, and up to 130 if we ignore only 5% of the classes with the highest number of associations (columns *Max assoc. for 90% of classes* and *Max assoc. for 95% of classes*.) In the cases considered so far, the classes with the highest number of associations are algorithm intensive classes, *tokenizers* or *parsers* for Jedit, Lucene and JTopas, which use state variables for storing intermediate results of the computation. Most of these classes are automatically produced with parser

generators, and are of little interest from the class testing viewpoint. Thus, we can obtain good results even ignoring the associations of these classes. As shown in the example of Section 4, a single test case may cover several associations, thus we can cover the associations with a number of test cases smaller than the amount of associations.

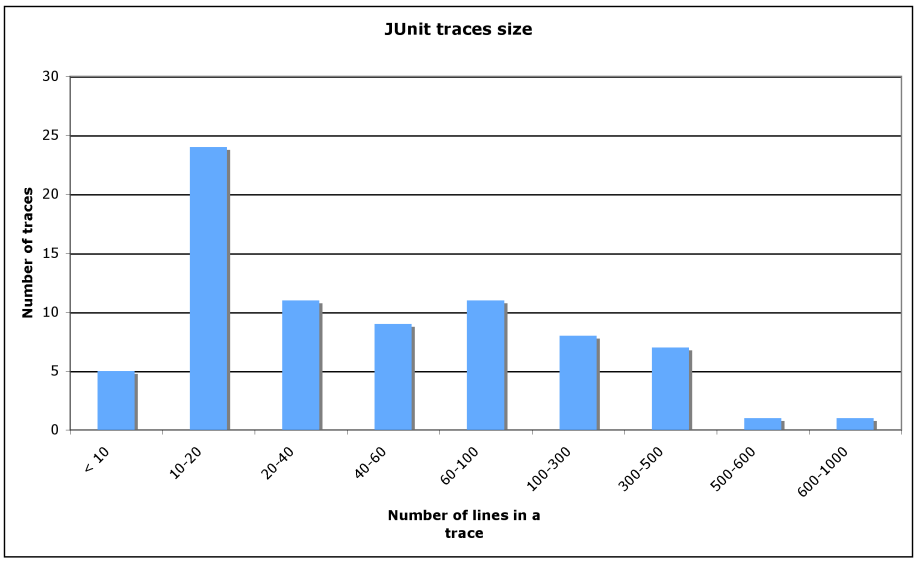

Fig. 7. Size of JUnit traces

We ran data flow analysis three times on each program. The execution times are reported in the last columns of table 5. The results indicate that the technique can scale up well.

We performed coverage analysis for JUnit and Lucene. Figure 7 represents the frequency of the generated traces grouped per size (number of lines). Most of the traces are small, between 10 and 20 lines, and all of them are shorter than 1000 lines. JUnit traces have been analyzed in less than 1 second.

The coverage analysis of Lucene produced interesting results. Since the test cases provided with the source code are longer than the ones provided with JUnit, the generated traces are longer, too. Figure 8 represents the frequency of Lucene traces grouped per size. Most of the traces are between 10^4 and 10^5 lines long, but some traces have more than 10^8 lines. We could analyze all of them in at most 46 minutes. In Figure 9 we plot the time required to analyze traces of different size. As we have already said in section 4, the computation time depends on both the number of lines in a trace and the number of def-use associations involved. Thus, bigger traces usually take more time to be analyzed, but sometimes the analysis of small and more complex traces is longer.

All data have been computed on the server of our laboratory (a Dell PowerEdge 2900 server with two 3.0 GHz Dual-Core Intel Xeon processors and 8 GB of RAM).

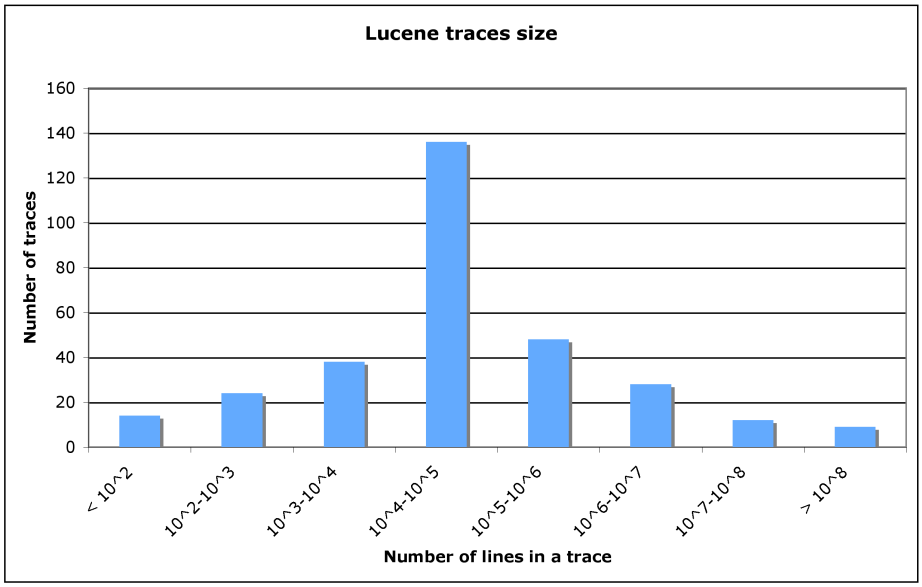

Fig. 8. Size of Lucene traces

5.2 Feasibility

Coverage criteria may not be effective when they statically identify too many infeasible elements, thus resulting in a highly variable coverage. To evaluate the impact of infeasible associations, we inspected all def-use associations derived from JUnit and some representative classes of Lucene. JUnit presents a total of 102 def-use associations, and only 3 of them are infeasible. Thus 96% def-use associations are feasible in JUnit. Three associations are infeasible because they involve uses in unreachable code, and none is infeasible due to data flow limitations.

Complete information on JUnit is provided in the top part of table 6. The first columns list the analyzed classes and their features. Column *Assoc.* represents the number of def-use associations in a class, while *Original cov.* and *Tot cov.* report the amount of associations covered by the original test suite and by the augmented one. Column *Infeasible* lists the number of infeasible associations in a class.

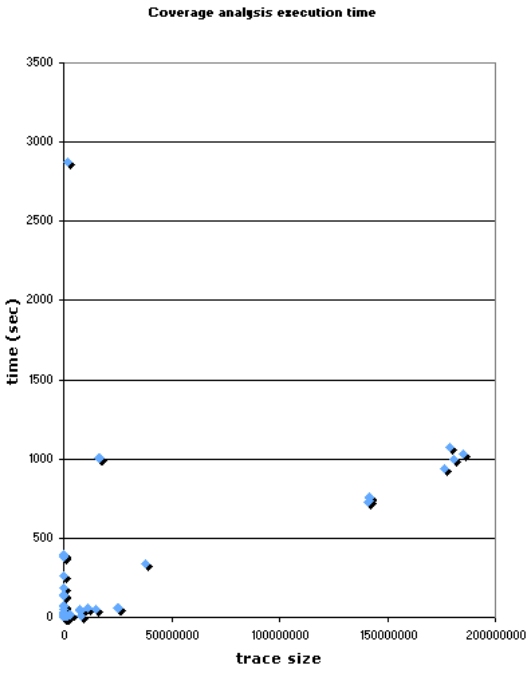

Fig. 9. Coverage analysis execution time

In Lucene we performed a complete analysis of the small package `document`. The 9 classes of Lucene that we inspected contain a total of 354 def-use associations, 74 (20%) of which are infeasible. All these infeasible def-use associations derive from the impossibility of executing all associations in all possible contexts, since class `Field` invokes some methods with a subset of parameter values that restrict the amount of code that can be executed. Complete information on Lucene is listed in the bottom part of the table 6

5.3 Effectiveness

We studied the ability of exposing failures, by comparing the performance of standard test suites with test suites that guarantee coverage of feasible def-use associations. We ran the experiments on the 3.8.1 distribution of JUnit, since it is distributed with a standard test suite. We first augmented the test suite provided with the distribution with additional test cases that guarantee 100% coverage of the feasible def-use associations. We then compared the effectiveness of the standard and the augmented test suites, by measuring the ability of revealing a set of seeded faults.

The 58 test cases of the standard test suite cover only 52% of the def-use associations (54 out of a total of 102 associations, 99 of which are feasible) and

JUnit							
Class name	SLOC	State vars	Methods	Assoc.	Original cov.	Tot cov.	Infeas
junit.framework.Assert	184	0	39	0	0	0	0
junit.framework.AssertionFailedEror	10	0	2	0	0	0	0
junit.framework.ComparisonFailure	44	2	2	18	18	18	0
junit.framework.Protectable	4	0	1	0	0	0	0
junit.framework.Test	5	0	2	0	0	0	0
junit.framework.TestCase	79	1	13	15	8	12	3
junit.framework.TestFailure	38	2	7	7	1	7	0
junit.framework.TestListener	7	0	4	0	0	0	0
junit.framework.TestResult	111	5	18	20	17	20	0
junit.framework.TestResult$1	5	0	2	0	0	0	0
junit.framework.TestSuite	168	4	23	42	10	42	0
junit.framework.TestSuite$1	5	0	2	0	0	0	0
JUnit Total	650	14	115	102	54	99	3
TOT INFEASIBLE ASSOCIATIONS: 1%							

Lucene							
Class name	SLOC	State vars	Methods	Assoc.	Original cov.	Tot cov.	Infeas
org.apache.lucene.document.DateField	49	1	7	6	0	6	0
org.apache.lucene.document.DateTools	130	1	7	4	0	4	0
org.apache.lucene.document.DateTools$Resolution	25	8	3	0	0	0	0
org.apache.lucene.document.Document	129	2	14	20	20	20	0
org.apache.lucene.document.Field	257	12	23	321	88	247	74
org.apache.lucene.document.Field$Index	31	6	1	0	0	0	0
org.apache.lucene.document.Field$Store	21	5	1	0	0	0	0
org.apache.lucene.document.Field$TermVector	37	7	1	0	0	0	0
org.apache.lucene.document.NumberTools	52	1	2	3	0	3	0
Lucene Total	617	43	59	354	108	280	74
TOT INFEASIBLE ASSOCIATIONS: 20%							

Table 6. Feasible associations in JUnit and Lucene.

76% of the statements. We augmented the default test suite with 19 additional test cases to cover all 99 feasible associations, and we reached a statement coverage of 87%. Neither of the suites find any fault, as expected, since JUnit is a mature tool, and the package `junit.framework`, which we analyze, is the core of the tool.

A qualitative inspection of the additional test cases indicates that the original test suite omits some relevant checks: The test cases of the original suite (1) do not check for the consistency of the initial state of most objects, (2) do not check some functionalities, and (3) do not check the behavior of some methods when invoked in states that violate the preconditions of the methods. For example, in JUnit, test suites can be given a name, but no original test case checks for the correct treatment of this case; moreover JUnit requires test cases to be defined in public classes, but the behavior in presence of test suites included in private classes is not checked.

To further assess the performance of the augmented test suite, we evaluated the original and the augmented suites with respect to a set of mutants generated according to the following rules:

- **Definition mutation**: for each definition of a state variable, we generated a mutant where the assigned value is the default value:
  - 0 if the variable is a number, 1 whether 0 is the actual value.
  - the negation of the actual value if the variable is a boolean.
  - null and another compatible value if the variable is a reference.
- **P-use mutation**: for each use of a state variable in a decision predicate, we replaced the comparison operator with its inverse.
  - $!=$ with $==$ and viceversa
  - $<$ with $>$ and viceversa.
  - $\leq$ with $\geq$ and viceversa.
- **C-use mutation**: for each use of a state variable in a command statement, we modified the used value.
  - We added 1 to the used integer values.
  - We used the negation of used boolean values.
  - We used a faulty method invocation that preserved the syntactic validity, in presence of used references. We replaced the called method with a different one (whenever possible) having a compatible return type when the reference was associated to a method call; we set a different method invocation with a return value compatible with the reference type when the reference was not associated to a method call (as in a return statement).
- for all indirect definitions and uses of a state variable (i.e., definitions and uses that take place through method invocations as identified by the context data), we replaced the input parameters of the called method with default values, using the same replacement rules as in case of direct definitions. In case of indirect definitions or uses through methods with no input parameters, we did not seed any fault.

The original suite identifies only 56 out of 83 seeded faults, the original suite augmented with 9 test cases to reach 100% coverage of the feasible statements identifies 65 seeded faults, while the test suite augmented to guarantee all def-use coverage identifies all 83 faults. Table 7 reports the results of this experiment.

Fault type	No. of seeded faults	Revealed in original	Revealed in augmented
Def mutation	32	16	32
P-use mutation	12	11	12
C-use mutation	39	23	39

Table 7. Mutations in JUnit.

5.4 Limitations and threats to validity

The main threats to the validity of the empirical results reported in this section derive from the limitations of the prototype used in the experimental work: the

language issues that are not addressed by the prototype (exception handling, polymorphism, and reference aliasing) might affect the precision of the computed sets of def-use associations, thus biasing our figures. Alias analysis algorithms, such us the ones proposed by Liang, Pennings and Harrold [24] and Milanova, Rountev and Ryder [23] should improve the precision of the results. We are currently including such algorithm in the next release of the prototype. Exception handlers may worsen the problem of infeasible associations, but we believe that they should be addressed independently.

The results on fault-detection effectiveness based on mutation analysis may be biased by the set of injected faults, even if they confirm the preliminary feedback from comparing different test suites. We are currently experimenting with known faults in publicly available applications to further confirm the preliminary results of mutation analysis.

Additional validity threats may derive from having experimented only with open-source software, which could not adequately represent software produced in industrial settings. We are working with our industrial partners to confirm the preliminary results obtained with open-source software.

6 Related Work

The problem of testing state-dependent behavior has been originally addressed in the domain of communication protocols [6, 5, 3, 4], and then further extended to cope with object oriented software [8, 2]. Most work on testing the state-dependent behavior of object oriented software has focused on deriving test cases from state-based specifications, often UML [14, 15, 9, 10, 37, 13, 12, 7, 11].

The most relevant code-based approaches to testing the state-dependent behavior of object-oriented software have exploited data flow analysis techniques. Harrold and Rothermel first, and Souter and Pollock later laid down the foundations [19, 21].

Harrold and Rothermel introduced a suitable data flow model of class interactions, and defined intra- and inter-class testing. We based our analysis on Harrold and Rothermel's model, and our prototype on the JABA library [29], which extracts the model from Java bytecode.

Souter and Pollock introduced contextual def-use associations for better characterizing the interactions within object-oriented software, and proposed an algorithm that analyzes method interactions by examining complete chains of method invocations. Our framework adapts the notion of contextual def-use associations defined by Souter and Pollock, to provide a framework for defining intra-class integration testing strategies.

Souter and Pollock instantiated their approach in the prototype tool TATOO [38] that analyzes inter-method interactions, and used this tool to explore the use contexts of different granularity. TATOO was not publicly available at the time

of writing, thus we could not make a direct comparison with our prototype. Based on what published in the literature [38, 21], our prototype seems to scale up better than TATOO: uses of TATOO are reported for programs up to 10 KLOC and 100 classes, while we have successfully analyzed programs up to 100 KLOC and 1,000 classes. On the other hand, our static analysis for building contextual def-use associations is less precise than Pollock and Souter's. Their algorithm analyzes each method separately for each context in which it can be invoked, and partially accounts for reference aliasing. We analyze methods only once, handling invocations between methods based on the raw information that results from the analysis of the called methods, and ignore aliasing for the moment.

Other tools that address data flow coverage for Java programs, e.g., Coverlipse [26] and JaBUTi [39] consider only intra-method interactions, thus they are not suitable for intra-class testing.

Several papers propose approaches to increase the precision of the data flow analysis for Java by accounting for reference aliasing [40, 41, 24, 33, 23, 32]. We are currently extending our prototype to include alias analysis, and improve the precision of the results.

The presence of libraries or components available without source code impacts on the precision of data flow analysis. Rountev et al. propose an approach to interprocedural dataflow analysis that relies on summary information provided with external libraries, and does not require access to the source code [42]. Since our analysis works at the byte-code level, we face this problem only in presence of native code in Java libraries.

Buy et al. combined data flow analysis with symbolic execution to automatically generate intra-class test cases for C++ programs [43].

7 Conclusions

Classic structural testing approaches do not adequately address subtle failures that may depend on state-dependent behavior of object-oriented classes. Data flow analysis techniques have been recently extended to capture dependencies from instance variables that determine the state of the classes, taking into account the context in which methods that access instance variables are invoked.

In this paper, we propose a framework that adapts and extends previous results to intra-class structural integration testing [19]. The main contributions of this paper are the adaptation of Souter and Pollock's context-sensitive analysis of object-oriented methods [21] to the incremental analysis of intra-class interactions, and a preliminary set of empirical data obtained though a prototype implementation of the approach on a set of open-source programs.

The analysis proposed in this paper can be performed incrementally on single classes, since definition and use sets can be combined at integration time,

while Souter and Pollock's analysis combines the points-to-graphs of the different methods for each method invocation, and thus gains precision at the expense of scalability. Souter and Pollock's approach applies well to the analysis of complete programs and single methods, while our approach fits better the analysis of the interactions of methods that can be invoked independently form outside of the class.

The empirical results confirm the complexity results and suggests that the technique proposed in this paper scales up well to mid-size programs (we analyzed programs of up to 100,000 lines of code and 1000 classes in few minutes). The results indicate also a high percentage of feasible def-use associations, thus sustaining the usefulness of the proposed structural coverage. Finally, the results suggest that the coverage proposed in this paper include relevant state-dependent behavior that are ignored by classic structural coverage criteria.

The release of the prototype used in the empirical evaluation and the complexity of some experiments limit the empirical results obtained so far. The prototype implementation is based on the current release of JABA that derives inter-class data flow models, but does not implement alias analysis yet. The analysis of infeasible def-use associations as well as the derivation of test suites that cover feasible associations require expensive inspection of program code and test traces. We are currently including well known alias analysis algorithms in the prototype to improve the precision of the results [24, 23]. We are also including techniques to deal with polymorphism, dynamic binding and exception handling [33, 32], and we are analyzing feasibility and coverage of additional large programs.

In this paper, we focused on intra-class testing, but we are currently extending the technique to inter-class testing, i.e., to methods that interact through shared objects, such as global objects and shared state instances. The main challenge of inter-class testing is understanding which of the shared associations that can be statically identified can also be dynamically instantiated. Too many infeasible shared associations lead to too many infeasible test requirements, and thus almost useless testing criteria. The technique proposed in this paper can easily identify shared associations, but we are currently investigating the amount of infeasible associations computed with the technique, to learn if and how to improve it.

References

1. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall (March 2000)
2. Pezzè, M., Young, M.: Software Test and Analysis: Process, Principles and Techniques. John Wiley and Sons (2008)
3. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE **84**(8) (August 1996) 1090–1123

4. von Bochman, G., Dssouli, R., Zhao, J.R.: Trace analysis for conformance and arbitration testing. IEEE Transactions on Software Engineering **15**(11) (November 1989) 1347–1356
5. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Transactions on Software Engineering **17**(6) (June 1991) 591–603
6. Chow, T.S.: Testing design modeled by finite-state machines. IEEE Transactions on Software Engineering **4** (March 1978) 178–186
7. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y.S., Song, Y.K.: Developing an object-oriented software testing and maintenance environment. Communications of the ACM **38**(10) (1995) 75–87
8. Binder, R.V.: Testing Object-Oriented Systems, Models, Patterns, and Tools. Addison-Wesley (2000)
9. Briand, L.C., Cui, J., Labiche, Y.: Towards automated support for deriving test data from UML statecharts. In: Proceedings of the International Conference on the Unified Modeling Languages and Applications. LNCS 2863, Springer (2003) 249–264
10. Briand, L.C., Labiche, Y., Wang, Y.: Using simulation to empirically investigate test coverage criteria based on statechart. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society (2004) 86–95
11. Offutt, A.J., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: Proceedings of the Internationcal Conference on Engineering of Complex Computer Systems, IEEE Computer Society (1999) 119–129
12. Hartmann, J., Imoberdorf, C., Meisinger, M.: Uml-based integration testing. In: Proceedings of the 2000 International Symposium on Software Testing and Analysis, ACM Press (2000) 60–70
13. Fraikin, F., Leonhardt, T.: SeDiTeC - testing based on sequence diagrams. In: Proceedings of the International Conference on Automated Software Engineering, IEEE Computer Society (2002) 261–266
14. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation. In: Proceedings of the International Conference on the Unified Modeling Language. LNCS 1939, Springer (2000) 383–395
15. Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability **13** (2003) 95–127
16. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology **3**(2) (April 1994) 101–130
17. Chen, H.Y., Tse, T.H., Chan, F.T., Chen, T.Y.: In black and white: an integrated approach to class-level testing of object-oriented programs. ACM Transactions on Software Engineering and Methodology **7**(3) (1998) 250–295
18. Chen, H.Y., Tse, T.H., Chen, T.Y.: TACCLE: a methodology for object-oriented software testing at the class and cluster levels. ACM Transactions on Software Engineering and Methodology **10**(1) (2001) 56–109
19. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press (1994) 154–163
20. Harrold, M.J., Soffa, M.L.: Efficient computation of interprocedural definition-use chains. ACM Transactions on Programming Languages and Systems **16**(2) (1994) 175–204

21. Souter, A.L., Pollock, L.L.: The construction of contextual def-use associations for object-oriented systems. IEEE Transaction on Software Engineering **29**(11) (2003) 1005–1018
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley (2000)
23. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology **14**(1) (January 2005) 1–41
24. Liang, D., Pennings, M., Harrold, M.J.: Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In: Proceedings of the ACM Workshop on Program Analysis For Software Tools and Engineering, ACM Press (2005) 6–12
25. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **SE-11**(4) (1985) 367–375
26. Kempka, M.: Coverlipse: Eclipse plugin that visualizes the code coverage of JUnit tests Open source project on SourceForge.net, http://coverlipse.sourceforge.net.
27. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press (1973) 194–206
28. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
29. JABA: Aristotele Research Group. Java Architecture for Bytecode Analysis (2005)
30. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering **26**(9) (2000) 849–871
31. Chatterjee, R., Ryder, B.G.: Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Department of Computer Science, Rutgers University (2001)
32. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Transactions on Software Engineering **27**(6) (June 2001) 481–512
33. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. IEEE Transactions on Software Engineering **30**(6) (2004) 372–387
34. Forgács, I.: An exact array reference analysis for data flow testing. In: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society (1996) 565–574
35. Hamlet, D., Gifford, B., Nikolik, B.: Exploring dataflow testing of arrays. In: Proceedings of the 15th International Conference on Software Engineering, IEEE Computer Society Press (1993) 118–129
36. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal **10**(4) (2005) 405–435
37. Briand, L.C., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing: A series of experiments. IEEE Transactions on Software Engineering **30**(11) (2004) 770–793
38. Souter, A., Wong, T., Shindo, S., Pollock, L.: TATOO: Testing and analysis tool for object-oriented software. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2031, Springer (2001)

39. Vincenzi, A.M.R., Maldonado, J.C., Wong, W.E., Delamaro, M.E.: Coverage testing of Java programs and components. Science of Computer Programming **56**(1-2) (2005) 211–230
40. Liang, D., Pennings, M., Harrold, M.J.: Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, ACM Press (2001) 73–79
41. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press (2001) 43–55
42. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Proceedings of the 15th International Conference on Compiler Construction. LNCS 3923, Springer (2006) 2–16
43. Buy, U., Orso, A., Pezzè, M.: Automated testing of classes. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM Press (2000) 39–48